SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings

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

Kexin Pei
Columbia University
kpei@cs.columbia.edu

Jun Yeon Won
The Ohio State University
won.126@osu.edu

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

ABSTRACT

Predicting function names in stripped binaries is an extremely useful but challenging task, as it requires summarizing the execution behavior and semantics of the function in human languages. Recently, there has been significant progress in this direction with machine learning. However, existing approaches fail to model the exhaustive function behavior and thus suffer from the poor generalizability to unseen binaries. To advance the state of the art, we present a function Symbol name prediction and binary Language Modeling (SymLM) framework, with a novel neural architecture that learns the comprehensive function semantics by jointly modeling the execution behavior of the calling context and instructions via a novel fusing encoder. We have evaluated SymLM with 1,431,169 binary functions from 27 popular open source projects, compiled with 4 optimizations (O0-O3) for 4 different architectures (i.e., x64, x86, ARM, and MIPS) and 4 obfuscations. SymLM outperforms the state-of-the-art function name prediction tools by up to 15.4%, 59.6%, and 35.0% in precision, recall, and F1 score, with significantly better generalizability and obfuscation resistance. Ablation studies also show that our design choices (e.g., fusing components of the calling context and execution behavior) substantially boost the performance of function name prediction. Finally, our case studies further demonstrate the practical use cases of SymLM in analyzing firmware images.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**;
• **Security and privacy** → **Software reverse engineering**.

KEYWORDS

Function name prediction; binary reverse engineering; transfer learning; calling context; execution behavior

ACM Reference Format:
Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. 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 (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560612

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560612

1 INTRODUCTION

Predicting function names in stripped binaries is extremely useful for many security applications such as malware analysis [84, 92], vulnerability identification [63, 85], binary code hardening [23, 74], binary code reuse [20, 95], program comprehension [26, 86], and decompilation [53, 65]. As a concrete example, security companies such as Mandiant [50] have invested significant resources in developing various reverse engineering tools for symbol recovery, function annotation, binary code matching, and binary code attribution, in order to help human analysts to analyze the functionalities of the malware and even trace back their origins more efficiently.

Unfortunately, constructing meaningful function names is an extremely challenging task. In particular, function names often provide a high-level summary of the approximate behavior of functions, as shown in Figure 1. Predicting the function name thus relies on understanding of what operations the function performs and the capability of combining and mapping them to a few keywords in human languages. However, various compiler idioms, optimizations or obfuscation passes, application binary interfaces (ABIs) across different operating systems, and hardware specifications of diverse architectures introduce an extremely diverse binary code representations even for the code with the same semantics. Reasoning about their execution behavior is thus nontrivial and prohibitively expensive. Moreover, the noisy nature of human language even exacerbates the problem. For example, studies have shown that different developers often name the same functions differently (e.g., with only 6.9% probability choosing the same name) [35] and with different choice of vocabularies, resulting in out-of-vocabulary (OOV) problems [54].

Interestingly, recent efforts (e.g., [38, 41, 50, 57]) have shown a promising direction by training Machine Learning (ML) models from a large set of binary function name mappings and automatically learning useful patterns in the binary code for function name prediction. However, their results on even simple binaries (e.g., without compiler optimizations) are suboptimal (around 40% from the state of the art), let alone the binaries with more aggressive optimizations or obfuscations that often break the spurious patterns learned from these models. For example, as shown in Figure 1, the function name `analogRead` should be attributed to the execution behavior of the body within the `else` block (line 4-12) instead of any other

```
1  uint32_t analogRead(uint32_t ulPin){
2    ...
3    if (pin == NC) uVar3 = 0;
4    else {
5      uVar2 = adc_read_value(pin);
6      uVar3 = (uint32_t)uVar2;
7      if (uVar4 != 0xc) {
8        if ((uint) uVar4 < 0xc)
9          return (uint)(uVar2 >> (0xcU - uVar4 & 0xff));
10       return uVar3 << (uVar4 - 0xcU & 0xff);
11     }
12   }
13   return uVar3;
14 }
```

Figure 1: A piece of decompiled code for function `analogRead` from the Gateway firmware image [2]. Understanding the behavior of the callee at line 5 is critical to predict the function name at line 1.

patterns. Yet, none of the existing approaches attempts to expose the dynamic behavior of the function to the ML model for function name predication. Without learning how binary program behaves dynamically, the ML models cannot reason precisely about function behavior and relate them to function name, while instead resort to other patterns, which might not persist when the code is compiled on different architectures or with different compiler optimizations.

On the other hand, there are also a number of attempts (e.g., [75, 77, 90, 91]) to learn execution-informed program representations for various downstream program analysis tasks, e.g., type inference, program repair, etc. With the training dataset augmented by execution traces, the ML models have direct access to observe how code would behave during runtime, and tend to more likely learn program semantics and map them to target labels. Unfortunately, the key limitation of these works is that *they are completely agnostic to the calling context*, i.e., none of them models program behavior beyond a single function. For example, Trex [77] skips function calls during tracing, and Stateformer [75] only considers the return value of callees. Failing to consider the calling context significantly restricts the actual functions' behavior accessible to the ML models. Consider the function shown in Figure 1, where the function name `analogRead` at line 1 is highly related to the behavior of its callee `abc_read_value` (line 5). Our studies in §5.5 suggest that existing ML models [77] fail to predict this function name meaningfully, and overall they are 7.9% less accurate compared to our approach that considers the calling context.

Our Approach. In this paper, we present SymLM, a novel neural architecture to learn *context-sensitive* function semantics for function name prediction. Based on the execution-aware pretrained models [77], SymLM learns the comprehensive function semantics by jointly modeling the execution behavior of the calling context and function instructions via a novel fusing encoder module. To address the ambiguity of natural language in function names, i.e., synonyms and out-of-vocabulary (OOV) words, we build *CodeWordNet* encapsulating the domain-specific distributed representations of function name tokens to measure the semantic distance between the predicted and ground-truth names.

Our extensive experiments demonstrate that SymLM is accurate: achieving 0.634 precision, 0.677 recall, and 0.655 F1 score across various architectures and optimizations. SymLM outperforms all state-of-the-art function name prediction works by up to 35.0% in F1 score. By learning context-sensitive function semantics, we show that SymLM is more generalizable and robust than existing works, outperforming the state-of-the-art by 295.5% in F1 score when evaluated against unseen binary functions. Additionally, the ablation studies further demonstrate the effectiveness of SymLM's design in incorporating calling context. For example, the semantics fusing encoder can improve the performance of SymLM by up to 36.2% in F1 score. We also apply SymLM on real-world IoT firmware and it successfully predicts the function names in the firmware images.

Contributions. Our paper makes the following contributions:

- We design a novel neural architecture, SymLM, to learn the comprehensive function semantics preserved in the execution behavior of the calling context and function instructions.
- We build the CodeWordNet module to measure semantic distance of function names with the domain-specific distributed representations and address OOV problems by preprocessing.
- We advance the state-of-the-art of function name prediction by outperforming existing works and show the generalizability, obfuscation resistance, component effectiveness, and the practical use case of SymLM. Our code and dataset are released at https://github.com/OSUSecLab/SymLM.

2 BACKGROUND AND MOTIVATION

2.1 Problem Definition

We define the semantics of a binary function $\mathcal{F}$ as $\mathcal{E}$, which is manifested by the execution behavior of a sequence of its instructions $\mathcal{I}$ and the calling context C. The calling context C denotes the set of functions that calls $\mathcal{F}$ (callers) C^+ and functions that $\mathcal{F}$ calls (callees) C^-. That is, $C = \{C^+, C^-\}$.

We introduce an embedding space E (consisting of multi-dimensional numerical vectors) to represent $\mathcal{E}$. Let $E_{\mathcal{E}}$, $E_{\mathcal{I}}$, and E_C denote the vector representations (in E) of $\mathcal{E}$, $\mathcal{I}$, and C, respectively. We define the function semantics $\mathcal{E}$ as the composition of the semantics of function instructions $\mathcal{I}$ and the calling context C:

$$E_{\mathcal{E}} = \phi(E_C, E_{\mathcal{I}}) \tag{1}$$

where $\phi(\cdot)$ is a composition function.

Given $E_{\mathcal{E}}$, we define the function name prediction task as a multi-class multi-label classification problem. In particular, we aim to learn a mapping function $\Gamma(\cdot)$ that maps $E_{\mathcal{E}}$ to the function name W, which consists of a set of words $W = \{w_1, w_2..., w_n\}$, where W is a subset of the function name vocabulary $\mathcal{V}$ ($W \subseteq \mathcal{V}$):

$$W = \Gamma(E_{\mathcal{E}}) \tag{2}$$

Here $\mathcal{V}$ consists mostly of English words and also other commonly developer-chosen terms such as abbreviations, types (e.g., `int`, `float`), numbers, and even misspellings.

2.2 Challenges

Learning to predict function names (i.e., the mapping Γ) requires modeling the semantics of both binary functions and their names and learning how they should align. We discuss five challenges in this task as follows.

C1: Limited Semantic Information in Stripped Binaries. Stripped binaries do not have symbol information, which otherwise provides strong hints for function name prediction [46, 60, 64]. For example, Host et al. [46] propose a set of association rules to infer the verbs in function names from the identifier names in function bodies. Liu et al. [64] demonstrate that good function names can be constructed from semantic information in function bodies. Moreover, the names of function parameters and return types have also been shown to be valuable for name prediction [60], but these valuable semantic information are often stripped in binaries.

C2: Variety of Compilation Settings and Obfuscations. The use of different compilation settings (e.g., compilers, optimization levels, and computer architectures) and obfuscations pose significant challenges to ML models to learn function semantics, as even with the same source code, binaries compiled with different compilation flags or obfuscated with compiler-based transformations can have completely different instructions and control flow graphs [31].

C3: Ambiguous Function Names and Various Naming Methods. Function names often consist of words in the natural language, which is by the ambiguous nature of human languages [25]. Part of the ambiguity stems from the use of morphological forms (e.g., synonyms, abbreviations, and even misspellings) [34], and they are found to be more ubiquitous in function names than in general written texts [52]. Meanwhile, software developers tend to choose shorter and more concise names [42]. Consequently, different developers can name the same function with totally different words. Feitelson et al. [35] even showed that the probability that two developers select the same name for the same function is only 6.9%. Although developers are recommended to name functions by full words [44], it has been found that abbreviations are popularly used for many reasons, such as abbreviations being as understandable as complete words [81]. Single letters are even found to be meaningful names when properly chosen [16].

Moreover, unlike written texts that divide words by standard punctuation marks, there are numerous naming conventions suggested by both industry [30] and academia [15]. The delimiters used by developers are diverse [33]. Based on our observations, developers can use capital letters, underscores, and even numbers to split words. Therefore, the ambiguity of function names and the diversity of naming methods make it difficult to predict function names.

C4: The Out-Of-Vocabulary (OOV) Issue. The OOV issue can result in poor performance for machine learning models. More specifically, models generate outputs based on the existing training vocabulary [39]. Without knowing the OOV classes and labels, they cannot predict meaningful function names. The model for function name prediction also suffers from this problem at both the whole name and individual word levels. At the whole function name level, our investigation on the binary dataset reveals that 21.7% function names have occurred only once. Although we can mitigate this problem by splitting names into individual words, the OOV ratio still remains relatively high, e.g., 5.8% for ARM binaries. During the investigation, we also found that this issue usually comes from the bad naming habits from developers such as the use of uncommon words. For example, Table 2 shows the categories of OOV words, in which 52.2% OOV words are the direct concatenation of multiple words or abbreviations, e.g., `sharefile` and `streq`. Resolving this issue requires an in-depth understanding and properly modeling of the function name words.

Table 1: Comparison with Closely Related Works in Addressing Challenges

System	Year	C1	C2	C3	C4	C5
Debin [41]	2018	✗	✗	✗	✗	✗
Punstrip [73]	2020	✗	✗	✓	✗	✗
NERO [27]	2020	✗	✗	✗	✗	✓
Trex [77]	2020	✓	✓	✗	✗	✗
NFRE [38]	2021	✗	✗	✓	✗	✗
StateFormer [75]	2021	✓	✓	✗	✗	✗
SymLM	2022	✓	✓	✓	✓	✓

C5: Calling Context Modeling. Calling context is the key semantic component of binary functions, which has been found to be helpful for function name prediction at source code level [60]. Therefore, to predict the function name, we need to extend modeling the behavior of the function itself to its calling context. However, none of the existing works considers the execution behavior of calling context, including the most recent works that have already made initial attempts to model the functions' execution behavior (§2.4). While NERO [27] constructs the function call graph and includes the information of function call sites as its model's input, it does not fully consider the behavior of the called functions.

2.3 Prior Efforts and Our Motivations

Predicting function names of binary code is not new, and there have been several attempts from the machine learning perspective recently. However, as shown in Table 1, previous efforts have not yet adequately addressed the aforementioned five challenges, which directly motivates us to propose SymLM.

More specifically, Debin [41] lifts the binary code into the BAP IR and designs a list of association rules derived from the DWARF entries. Then it trains models based on these association rules to predict the debug information. However, it does not consider the calling context and cannot learn function semantics from the rule-based features because they only contain local information, e.g., the relationship between operands in the same instruction. The same issues exist in Punstrip [73] which lifts binaries into the VEX IR and predicts function names based on a list of hand-crafted features, e.g., binary file sizes. In contrast, NERO [27] only considers the calling context, where it constructs call site graphs from call-related instructions, but ignores all other functions instructions. Therefore, NERO does not encode the complete semantics of functions. NFRE [38] proposes a structural instruction embedding method to standardize instructions. However, the result representations of instructions will miss their internal semantics. For example, with this proposed approach, instructions `push rbp` and `mov rbp, rsp` are converted to `INST_1325` and `INST_0b25`. Thus, existing works have not properly addressed **C1** and **C2**, and none of them can learn the full semantics of functions.

Moreover, Debin predicts function names at the whole name level, which does not consider **C3** and **C4**. Meanwhile, NERO divides function names into words, but does not propose any solutions to **C3** and **C4**. Punstrip and NFRE find synonyms by using existing synsets, generated from the general English corpus, which cannot

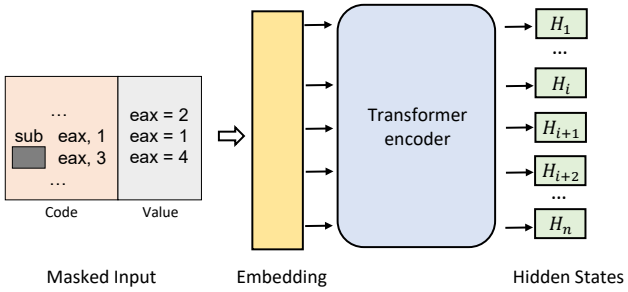


Figure 2: Microtrace-based Pretrained Model

identify morphological words specific to the software domain. And both works ignore the OOV word problems and fail to address **C4**. For **C5**, only NERO partially solves it by considering callee names and arguments, while others do not solve it.

2.4 Microtrace-Based Pretraining

Recently, TREX [77] was proposed to learn the instructions' execution semantics by pretraining the model with micro-execution trace values. TREX has shown the promise of the execution-aware modeling on the task of detecting semantically similar binary functions. Figure 2 shows the pretrained model of TREX. The model takes as input the code and values collected by microexecuting function instructions. For example, the values of the register `eax` are 1 and 4 before and after executing the second instruction. When given a masked operand in this instruction, TREX tries to reason the execution logic behind the calculation of `1?3=4`. In this case, by training to minimize the loss of prediction of the masked tokens, TREX is expected to learn the execution semantics of `add`. After pretraining, TREX generates semantic embeddings (the vector representation) of each instruction, which can be further used to detect binary functions with the similar execution behavior. Similarly, its followup work STATEFORMER [75] also learns the semantics of function execution behavior for type inference. However, TREX and STATEFORMER are not designed for function name prediction, and thus they cannot solve the corresponding challenges (**C3** and **C4**) as shown in Table 1.

Missing Calling Context. Unfortunately, TREX does not consider the side effect of function calls either, so it can only learn partial function semantics. The same problem also exists in STATEFORMER, which limits their usability of binary function semantics understanding while facing the challenge **C5**. To bridge the gap, we propose SYMLM, based on the microtrace-based pretrained model, to learn the full function semantics for function name prediction.

3 OVERVIEW

Figure 3 presents SYMLM's workflow on function name prediction, which consists of three key steps: (*i*) function semantics encoding to understand the comprehensive semantics of binary functions and encode them as structured representations, (*ii*) function name preprocessing and CodeWordNet training to model function name semantics by mitigating language ambiguity in the context of function names and reducing OOV words, and (*iii*) training and testing to learn the mapping from full function semantics embeddings to names and predict names. We describe each of the steps as follows.

Table 2: Categories of OOV Function Name Words

	Category	Ratio	Examples
1	Abbreviation concatenation	29.9%	statinfo, streq
2	Clean word concatenation	22.3%	sharefile, startpoints
3	Misspelling	14.6%	anewer, tac, sb
4	Clean word	12.1%	dependent, specifer
5	Abbreviation	7.0%	utils, pred
6	Inflection	9.6%	addresses, using
7	Digits in word	4.5%	add32, merge2

Function Semantics Encoding. In this step, we aim to train the model to generate the vector representation $E_{\mathcal{E}}$ that approximately encodes the function semantics $\mathcal{E}$. To this end, we propose a novel semantic fusing encoder module, which first encodes the semantics of the calling context C and function instructions $\mathcal{I}$ as E_C and $E_{\mathcal{I}}$, respectively, and then fuse them together as $E_{\mathcal{E}}$.

At first, SYMLM constructs inter-procedural control flow graphs (ICFGs) by disassembling the input binaries. In Figure 3, we show an example ICFG of a function (node 3), which has two callers (node 1 and 2) and two callees (node 4 and 5), where node 1-4 are the internal functions and node 5 is an external function of the input binary.

Next, SYMLM follows two major steps to generate $E_{\mathcal{I}}$: (1) it first uses the microtrace-based pretrained model (Figure 2) to generate embeddings of function tokens (we define this process as transformer encoding in Figure 3), and then (2) it downsamples the token embeddings to generate structured representations ($E_{\mathcal{I}}$) (see §4.1). For node 3 in Figure 3, we use E_3^{in} to specify its $E_{\mathcal{I}}$.

To produce E_C, SYMLM needs to encode the execution behavior of both the callees and callers. Since node 1, 2 and 4 are the internal functions, SYMLM uses the same approach of generating $E_{\mathcal{I}}$ to obtain the semantic embeddings E_1^{in}, E_2^{in}, and E_4^{in}. For the external function (node 5), whose instructions are often not accessible in binaries (e.g., the dynamically linked binaries), SYMLM generates its embeddings E_5^{ex} by looking up an embedding table (see §4.1). Finally, SYMLM integrates the embeddings of callees and callers to form E_C ($E_C = \{E_1^{in}, E_2^{in}, E_4^{in}, E_5^{ex}\}$) and generates the full function semantics encoding $E_{\mathcal{E}}$ based on Equation 1.

Function Name Preprocessing and CodeWordNet Training. We develop preprocessing steps and train CodeWordNet to tackle the problem of the ambiguous function names and OOV words. Specifically, SYMLM first tokenizes the whole function name into multiple individual word tokens. To address the OOV problem, we identify 7 categories of OOV issues (see Table 2) and develop multiple solutions to tackle each of them, described in the following.

First, during tokenization, we lemmatize and remove digits of words to address OOV categories 6 and 7 in Table 2. Second, we build a word segmentation model, based on the unigram language model [55], to segment the concatenated words of OOV categories 1 and 2 (see §4.2), and reduce the rest problems into OOV categories 3, 4, and 5. As the key reason for words in OOV categories 3, 4, and 5 is the use of morphological forms (e.g., synonyms, abbreviations and misspellings), we design the CodeWordNet module, which can generate distributed representations of words in the context of function names (see §4.3). With this module, SYMLM can measure the semantic distance of words, model function name semantics, and address OOV issues in categories 3, 4, and 5.

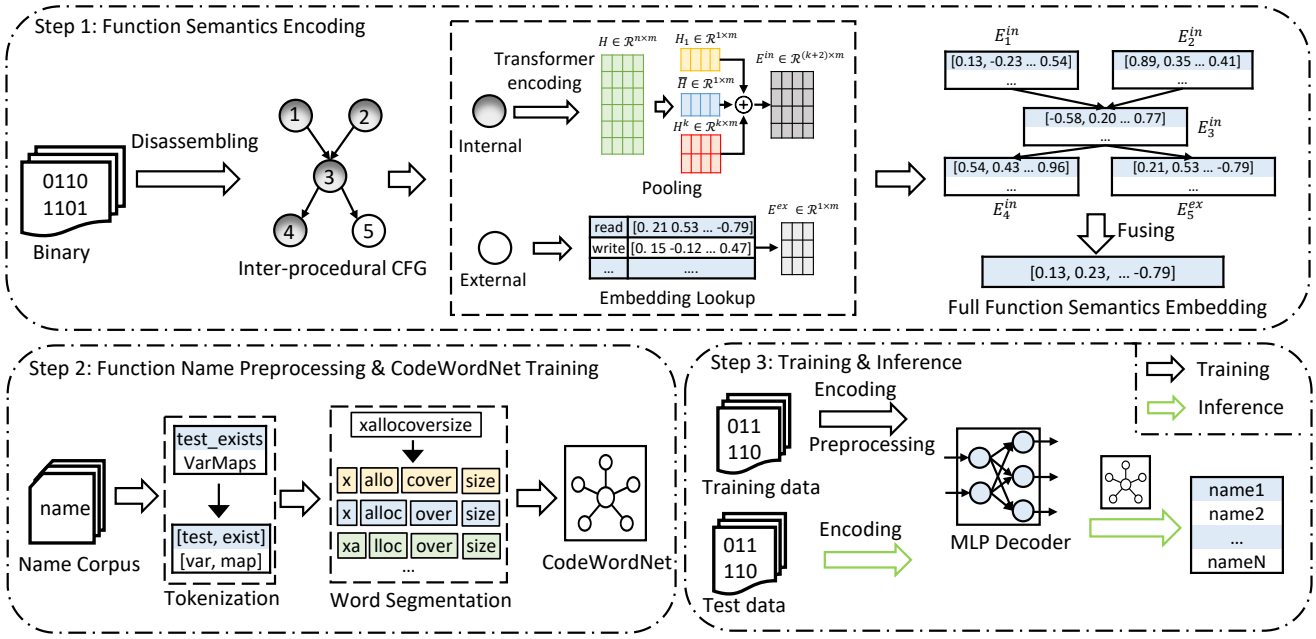


Figure 3: System Workflow

Training and Inference. To learn the semantic mapping Γ from function semantics to names (Equation 2), we train SymLM using the binaries with debug information, where the ground truth function names are parsed from symbol tables. For training, SymLM first encodes the full function semantics into embeddings, then preprocesses the ground truth, decodes the function semantic embeddings into predicted names by its multi-layer perception (MLP) decoder, finally gets trained by minimizing the prediction loss (§4.4). For inference, SymLM takes as input stripped binaries and predicts function name words based on the function semantics encodings, in which SymLM also leverages CodeWordNet to generate words that semantically match the input function semantics.

4 DETAILED DESIGN

4.1 Function Semantics Encoding

ICFG Construction and Input Generation. As discussed in §3, encoding the semantics of calling context requires understanding both callers and callees. To systematically analyze the control flows and determine function callers and callees, we set up to construct inter-procedural control flow graphs (ICFGs) from input binaries. More specifically, we first build a parser, based on the binary disassembler, to analyze input binaries. By iterating every binary function, the parser resolves the entry points of its callees and callers to build the ICFGs. Meanwhile, the parser prepares inputs for the function semantics encoding module to learn semantics for internal and external functions. For internal functions, five input sequences (including code, value, operand position, instruction position, and architecture sequences) are generated as the input for the macrotrace-based pretrained model (Figure 2). For external functions, only their names (if available) are collected for embedding lookup.

Internal Function Embedding. Internal function embedding is to encode the semantics of function instructions $\mathcal{I}$ as the vector representation $E_{\mathcal{I}}$, which is the key component of the function semantics. This process contains two steps.

First, SymLM generates semantic embeddings of every token in function instructions $\mathcal{I}$ by the microtrace-based pre-trained model. Specifically, as shown in Figure 2, the pre-trained model takes as input the parsed sequences, then creates input token embeddings with the embedding layer, and finally produces the hidden states with the transformer encoder. We use the hidden states $\{H_1, H_2, ..., H_n\}$ of the last layer as the semantic embeddings of the function instruction tokens.

Second, SymLM downsamples the token embeddings to generate structured representations $E_{\mathcal{I}}$. Unlike Trex which directly uses the mean of all tokens as function embeddings, we propose a specific pooling scheme for downsampling to achieve better performance. In particular, we observed that, in natural language processing tasks, the direct use of `[CLS]` token [29] embedding and the mean of all token embeddings as sentence embedding has been shown to be ineffective, which can cause 17.5% and 10.3% performance degradation respectively [59]. The root cause is that the BERT-based pretrained models generate anisotropic token embeddings [59]. The observations of the performance degradation motivate us to propose a new pooling scheme for generating $E_{\mathcal{I}}$ instead of directly using embeddings of `[CLS]` or the mean of all tokens.

Motivated by BERT-flow [59] and BERT-whitening [48], our embedding pooling scheme combines multiple downsampled embeddings, including `[CLS]` token embeddings, the mean of all token embeddings, and the token-position-insensitive embeddings, as function semantic embeddings $E_{\mathcal{I}}$. In particular, we denote H_1 as the `[CLS]` token embedding, since it is the first input token of the pretrained model [77]. Then the mean of all token embeddings can

by calculated by:

$$\overline{H} = \frac{1}{n} \times \sum_{i=1}^{n} H_i \tag{3}$$

For token-position-insensitive embedding generation, we sample the column-wise top-k elements from H:

$$H^k = argmax_{h \in H', H' \subset H, H' \in \mathbb{R}^{k \times m}} \{h\} \tag{4}$$

By concatenating the above embeddings, SymLM generates internal function embeddings as:

$$E_{\mathcal{I}} = \{H_1, \overline{H}, H^k\} \in \mathbb{R}^{(k+2) \times m} \tag{5}$$

In Figure 3, we visualize our proposed pooling scheme, where $E_{\mathcal{I}}$ is specified for internal functions as E^{in} to differentiate it from the embeddings of external functions (E^{ex}). Moreover, our proposed pooling scheme is highly effective, and our experimental results show that it improves SymLM's prediction performance by 35.4% (see §5.5 for more details).

External Function Embedding. External function embedding is to encode external function semantics. In real-world binaries, the function instructions of external functions are not always available, especially for dynamically linked binaries. Thus, we cannot use the microtrace-based pre-trained model and our proposed pooling scheme to generate embeddings for external functions. However, our key observation is that the semantics of external functions are usually fixed and well described by their names, such as Linux system calls [6]. Therefore, as shown in Figure 3, SymLM generates external function embeddings $E_{\mathcal{I}}$ (we specify $E_{\mathcal{I}}$ for external functions as E^{ex} in Figure 3) from an embedding loopup table, which is achieved by an embedding layer [78]. To build the lookup table, we collect a vocabulary of external function names from our training dataset. Based on this vocabulary, the embedding layer randomly initializes a dictionary to represent the semantics of the external function, which will be updated during model training to get the optimal embeddings for external functions. When the external function names are not available, we discard the effect of corresponding function calls.

Calling Context Embedding. As defined in §2.1, the calling context C includes callers C^+ and callees C^-. Here, we consider callers as internal functions (since we cannot know the external callers), so we can generate their embeddings by Equation 5. For callees, they can be either internal or external functions, which can be encoded with the ahead-mentioned methods. However, there is still a problem to be solved. That is, different binary functions have different numbers of callees and callers and the direct callees and callers can also have their own (indirect) callees and callers. One potential solution is to consider the complete function call effects. However, we observe that this solution dramatically affects SymLM's training and testing efficiency, as computers (or hardware accelerators) have limited memory and computational capacity. Alternatively, motivated by BinGo [21] and Asm2vec [31], we propose a selective calling context fusing method to solve the problem by two steps: (1) SymLM ranks callers and callees based on their frequency of calling or being called by the target function, and (2) SymLM concatenates the embeddings of top-n callers and top-n callees as calling context embeddings E_C without considering indirect calls:

$$E_C = \{E_{C_1^+}, ..., E_{C_n^+}, E_{C_1^-}, ..., E_{C_n^-}\} \tag{6}$$

where $E_{C_i^+}$ and $E_{C_j^-}$ are the embeddings of i-th caller and j-th callee generated by the aheadmentioned embedding methods.

Embedding Fusing. Embedding fusing is to generate the full function semantic embedding $E_{\mathcal{E}}$ by fusing the embeddings of function instructions and the calling context, $E_{\mathcal{I}}$ and E_C, based on Equation 1. Here, we fuse embeddings by concatenating them together:

$$E_{\mathcal{E}} = \{E_{\mathcal{I}}, E_C\} \tag{7}$$

4.2 Function Name Preprocessing

As analyzed in §2.2, the OOV issue can result in poor performance for function name prediction. A potential solution is to use Byte-Pair-Encoding (BPE) [83]. Unfortunately, it limits our choice of decoders into the sequential models (e.g., LSTM [43]), which are found to be ineffective (see §5.5). Alternatively, we propose a set of preprocessing approaches to address the OOV issue.

Function Name Tokenization. Function name tokenization is to split function names as individual words, which requires understanding of how function names are composed. Therefore, we manually investigated 776 names (randomly sampled from 388 binaries). Our key finding is that there are three major naming conventions: camelCase, PascalCase and snake_case, which split names by capital letters and underscores (_). Therefore, we tokenize function names by capital letters and underscores. However, as shown in Table 2, we still find many OOV words in tokenization results. With a manual investigation, we identify 7 categories of how OOV words are formed.

In particular, for category 7, we observe that developers often combine two words with one digit or put digits at the end of one word. Therefore, we split the words by eliminating the middle digit and removing the digits at the end of the words. For category 6 (inflection words), we follow two steps to lemmatize the OOV words into their original forms. First, we identify the tag of every word by part-of-speech (POS) tagging [87], as different tags of words have different inflection methods. For example, plural nouns are generated by adding `-s` or `-es`, and verbs are inflected by changing their tense with `-ing` or `-ed`. Second, we lemmatize words based on POS tags and convert the resulting words into lowercase forms. To this end, we process the OOV words in categories 6 and 7.

Word Segmentation. Word segmentation is to detect word boundaries and then segment words. As presented in Table 2, the majority (52.2%) of OOV words are in categories 1 and 2, and the root cause is that developers directly concatenate two words (or abbreviations). One potential solution is to directly use existing word segmenting tools, such as Word Ninja [28] that has 500+ stars on Github, or to adopt existing tools proposed by the natural language processing community (e.g., [82]). Unfortunately, none of them can effectively segment function names, because existing approaches and tools are for general written texts, while function names have domain-specific words and jargons.

To address this problem, we train an unigram language model [55] based on a large corpus of function names. Specifically, the unigram language model assumes that unigram tokens have

independent probability of occurrence, and hence the probability of the unigram token sequence $T = \{t_1, t_2, ..., t_n\}$ is the multiplication product of all its tokens:

$$P(T) = \prod_{i=1}^{n} p(t_i), \quad \forall t_i \in \mathcal{V}, \tag{8}$$

$$\sum_{t \in \mathcal{V}} p(t) = 1, \tag{9}$$

where $\mathcal{V}$ is the vocabulary. If $\mathcal{V}$ is given, the probabilities of unigram tokens can be estimated by maximizing the likelihood:

$$L = \sum_{s=1}^{|C|} log(P(T^s)) = \sum_{s=1}^{|C|} log(\sum_{t \in S(T^s)} P(t)) \tag{10}$$

where C is the corpus and $S(T)$ is a set of all possible segmentation candidates. To apply this model for function name segmentation, we first curate a large function name corpus from CodeSearchNet [49], which contains millions of functions. From CodeSearchNet, we successfully extracted 1,506,742 function names as the raw corpus. Next, we apply the previously introduced tokenization steps on the corpus to remove noises. With a pre-determined vocabulary size[1], we follow the heuristic steps [55] to train the unigram language model. Finally, the optimal word boundaries can be decided by:

$$t^* = \underset{t \in S(T)}{\text{argmax}}\, P(t), \tag{11}$$

With the proposed segmentation method, we successfully solve the OOV issues in categories 1 and 2 (Table 2). And our experimental results show that it can effectively mitigate most of OOV words along with our preprocessing approaches, e.g., the OOV ratio of our ARM-O0 test set is reduced from 10.9% to 1.7% (see §5.5).

4.3 Function Name Semantics Modeling

Although we have addressed 4 categories of OOV words (Table 2) by preprocessing, OOV words in categories 3, 4, 5 still exist and affect SymLM's performance. Moreover, even if some words have been seen by our model, it is still hard to predict names as the same ones given by developers. The root cause of this problem is the use of morphological words (e.g., synonyms, abbreviations, and misspellings) in function names. To identify morphological words, a manual approach can work for a small set of words, but it will fail for large datasets. To thoroughly address the problem, we propose *CodeWordNet*, a function name semantic encoding module.

CodeWordNet Training. We train CodeWordNet to generate distributed representations to encode the function name semantics. Specifically, we leverage three existing word embedding models: Continuous Bag of Words (CBOW) [67], Skip-Gram [67], and FastText [19], which have been widely used for generating distributed representations of program code (e.g., [12]). CBOW learns conditional probabilities of central words given context words in a fixed-size context window, while Skip-Gram learns conditional probabilities in the opposite way. FastText considers words at subword level, which helps it generate better word embeddings for rare words than the other models. We train CBOW, Skip-Gram and FastText models

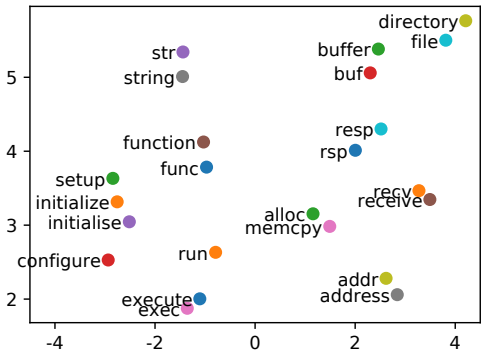


Figure 4: Visualization of Function Name Word Embeddings. The distance between words shows their semantic similarity.

by setting 3 for the window size[2] and enable negative sampling to reduce the cost of calculating softmax values [67]. Three models automatically converge to generate stable word embeddings.

Morphological Word Search. With CodeWordNet, we can search and find the moriphological words for the query words. Specifically, CodeWordNet first encodes every word w in the vocabulary $\mathcal{V}$ ($\forall w \in \mathcal{V}$) as the embedding vector $e(w)$. Then it finds the most similar words w ($w \neq w_q$) to the query word w_q by:

$$\underset{w \in \mathcal{V}}{\text{argmax}}\, cos(e(w), e(w_q)) = \underset{w \in \mathcal{V}}{\text{argmax}}\, \frac{e(w) \times e(w_q)}{||e(w)|| \times ||e(w_q)||} \tag{12}$$

where $cos(\cdot)$ is a cosine function. By applying Equation 12 to the three word embedding models, we obtain three sets of top-K most similar words for w_q. We visualize the results using t-SNE [88] to understand the effectiveness of our models. Figure 4 gives a visualization example of the CBOW word embedding. We find that the semantic similar words are close to each other, which indicates that the model can effectively measure semantic similarity by calculating the distance of function name words. We also find CBOW and Skip-Gram are good at learning synonyms and abbreviations, while FastText is effective at identifying misspellings.

Next, we combine all three similar word sets as the morphological word candidate set $M(w_q)$. In this process, we observe some noisy candidates, e.g., `smp` for `comp`. Motivated by prior works [24, 38], we propose a set of approaches to remove noisy candidates. First, we use the Damerau-Levenshtein distance [97] to calculate the edit distance of the strings between the query word w_q and the candidate word w. To mitigate length biases, we calculate the relative distance by:

$$d^r = \frac{d(w, w_q)}{max(||w||, ||w_q||)} \tag{13}$$

where $d(\cdot)$ is the Damerau-Levenshtein distance function and $||w||$ is the word length. Furthermore, we denoise the candidate set by only preserving candidate words w if any of the following rules is satisfied: (1) at least two of all three models agree that w is in the top-K[3] most similar word set of w_q, (2) $d^r < \frac{1}{3}$ and the first letters of w and w_q are the same, (3) w starts with w_q or w_q starts with w.

[1]We set the vocabulary size as 16K according to the suggested optimal vocabulary sizes: https://github.com/google/sentencepiece/issues/415

[2]The average number of words in function names is found to be 2.6 [35]

[3]In this work, we choose $K = 5$.

Finally, we follow the same manual validation approach as existing works [24, 38] to further remove the noise candidates from the candidate set $M(w_q)$. To this end, we successfully identify the semantic similar morphological word set and address the OOV words in category 3, 4, and 5 (Table 2). Our evaluations show that the use of function name preprocessing and CodeWordNet modules can effectively improve SymLM's performance by 16.7% in F1 score (see §5.5).

4.4 Training and Inference

As defined in §2.1, function name prediction is to map function semantics into names. In SymLM, this mapping is learned by training SymLM on binary function datasets.

Training on Binary Function Datasets. According to the problem definition in §2.1, function name prediction is a multi-class multi-lablel classification problem. Unlike existing works [27, 38] that use sequential models (e.g. LSTM [43]) to generate output words, we use a multi-layer perception (MLP) model, as we find it is more effective for this task according to our evaluations (see §5.5). The MLP model takes as input the function semantic embeddings $E_{\mathcal{E}}$ and then predicts function names by:

$$W = \psi_2(\omega_2 \cdot \psi_1(\omega_1 \cdot E_{\mathcal{E}} + b_1) + b_2) \tag{14}$$

where ω and b are the weights, $\psi(\cdot)$ is the activation function, and $W = \{w_1, w_2, ..., w_n\}$ is the function name that has been preprocessed into individual words w_i. To update parameters of the MLP model, the pre-trained model, and the external function embedding layer, we minimize the loss:

$$L = \sum_{w_i \in \mathcal{V}} l_{CE}(w_i, \hat{w}_i) \tag{15}$$

where w and $\hat{w}$ are the ground truth and the predicted words, $\mathcal{V}$ is the vocabulary, and $l_{CE}(\cdot)$ is the cross entropy loss function.

Function Name Inference. Function name inference is to predict names of test binary functions with the well trained SymLM model. However, as discussed in §2.2, it is very challenging to have the predicted names exactly matching the ground truth, which prohibits us demonstrating the effectiveness of SymLM. To address this problem, we take advantage of our proposed preprocessing approaches and the CodeWordNet module to perform semantic matching and mitigate OOV words of the ground truth. For semantic matching, we preprocess the ground truth to tokenize function names into words and reduce the OOV words. Next, we use CodeWordNet to match semantically similar words between the ground truth and predictions. Specifically, if the predicted words do not match the ground truth, we run CodeWordNet to identify their morphological words. If the morphological words are found in the ground truth, we treat them as correct predictions in our evaluations.

5 EVALUATION

We have implemented SymLM using Ghidra [69], and also the open-source microtrace-based pretrained model from Trex[4] [77], with additional 1,864 lines of our own code. Particularly, we built our function name preprocessing component based on NLTK [17] and SentencePiece [56], and constructed CodeWordNet based on Gensim [79]. We developed a Ghidra plugin and scripts to disassemble binaries, construct ICFGs, and parse debug information. We built the other components of SymLM with Pytorch [72] and fairseq [71].

In this section, we present the evaluation results. Particularly, we evaluated SymLM to answer the following research questions:

- **RQ1**: How effective is SymLM in function name prediction?
- **RQ2**: How does SymLM compare to the state of the art?
- **RQ3**: How can SymLM generalize to unknown binaries and resist obfuscations?
- **RQ4**: How can SymLM's components improve its performance?

Table 3: The Projects in Our Dataset

Project	# Binaries	Project	# Binaries
libtomcrypt	6528	libpng	48
binutils	3750	inetutils	44
coreutils	2485	bison	43
sg3-utils	1872	ImageMagick	32
putty	188	bc	32
findutils	128	libmicrohttpd	29
libgmp10	120	nano	24
imagemagick	108	sed	24
less	96	cflow	24
diffutils	88	wget	22
sqlite3	84	sqlite	16
openssl	81	busybox	16
curl	72	zlib1g	4
bash	69	-	-

5.1 Evaluation Setup

Datasets. We built our datasets with 27 open-source projects, including these being widely used in prior works [31, 75, 93], e.g., coreutils [1] and binutils [3]. We compile these projects using gcc-7.5 into different computer architectures (x86, x64, ARM, and MIPS) and different optimization levels (O0, O1, O2, and O3). To examine how SymLM can resist obfuscations, we obfuscate binaries with Hikari [4]. Specifically, we enable four obfuscation options, including bogus control flow (`bcf`), control flow flattening (`cff`), instruction substitution (`sub`) and basic block splitting (`split`), which are also used in other learning-based binary tasks [31, 77]. Finally, we obtain the datasets with 16,027 different binaries and 1,431,169 functions. Table 3 presents the details of binary numbers across the projects. We then split this dataset into training, validation, and test sets with a split ratio of 8:1:1 in binary level, which is adopted by other function name recovery experiments [27, 41].

Evaluation Metrics. We follow the same evaluation metrics as prior works [27, 38]. Specifically, given the ground truth function name $W = \{w_1, w_2, ..., w_n\}$ and the predicted function name $\hat{W} = \{\hat{w_1}, \hat{w_2}, ..., \hat{w_m}\}$, we define a membership function:

$$\mathbb{1}(W, \hat{w}) = \begin{cases} 1, & \hat{w} \in W \\ 0, & \hat{w} \notin W \end{cases} \tag{16}$$

[4]Note that Trex has open sourced its pretrained model at https://github.com/CUMLSec/trex with 60,606,229 number of parameters. Their model is pretrained on 1,472,066 binary programs in 10 days using 4 Nvidia RTX 2080-Ti GPUs. We directly leverage this pretrained model, but we have to perform additional training for our semantic fusing encoder module and the MLP decoder, which took 8 days in our evaluation environment.

Table 4: Overall Performance across Different Architectures (ARCH) and Optimizations (OPT)

ARCH	OPT	Precision	Recall	F1 Score
x86	O0	0.637	0.646	0.642
	O1	0.682	0.702	0.692
	O2	0.744	0.829	0.784
	O3	0.783	0.833	0.807
x64	O0	0.497	0.567	0.530
	O1	0.769	0.827	0.797
	O2	0.808	0.831	0.830
	O3	0.829	0.830	0.849
arm	O0	0.446	0.494	0.469
	O1	0.611	0.681	0.644
	O2	0.672	0.717	0.694
	O3	0.646	0.689	0.667
mips	O0	0.453	0.511	0.480
	O1	0.507	0.529	0.518
	O2	0.724	0.790	0.755
	O3	0.563	0.588	0.575

which indicates whether the predicted word $\hat{w}$ exists in the ground truth W. We then calculate the true positive (tp), false positive (fp), and false negative (fn) by:

$$tp = \sum_{\hat{w_i} \in \hat{W}} \mathbb{1}(W, \hat{w_i}), \quad fp = ||\hat{W}|| - tp, \quad fn = ||W|| - tp \quad (17)$$

where $||W||$ and $||\hat{W}||$ are the numbers of words in W and $\hat{W}$. Given N evaluation functions, we calculate precision (P), recall (R) and F1 score ($F1$) by:

$$P = \frac{\sum_{i=1}^{N} \frac{tp_i}{tp_i + fp_i}}{N}, \quad R = \frac{\sum_{i=1}^{N} \frac{tp_i}{tp_i + fn_i}}{N}, \quad F1 = 2 \times \frac{P \times R}{P + R} \quad (18)$$

Evaluation Environment. We performed the experiments on two machines: (1) a 64-bit Ubuntu 18.04 desktop machine, which has a 12-core Intel Xeon E5-1650 3.60GHz CPU, 64 GB memory, 4TB disk storage and 4 NVIDIA GeForce GTX 1080 Ti graphics cards, and (2) a 64-bit Windows 10 desktop machine, which has a 6-core AMD Ryzen 5 5600X CPU, 16 GB memory, and 1TB disk storage.

5.2 RQ1: Overall Effectiveness

We first studied how SymLM performs on our datasets. Table 4 presents the overall performance of SymLM. Among different architectures and optimizations, SymLM achieves the weighted average performance with 0.634 precision, 0.677 recall, and 0.655 F1 score. As SymLM predicts function names at the word level, this performance implies it can make semantically (or partially) correct predictions at the name level (see examples in §6.1).

We observe that SymLM performs better on O1-O3 binaries than O0 binaries. By manually inspecting binaries in different optimizations, we find that there are more functions in O0 binaries than O1-O3 binaries, which are compiled from the same projects. For example, compiling the `bison`[5] project with O0 and O1 optimizations renders 1,158 and 795 functions, respectively, for the x64 architecture. We hypothesize that function inlining causes the different numbers of functions, which is enabled for O1-O3 optimizations as specified in GCC manual [7]. Another possible reason for the lower performance of O0 optimization may come from information loss during function semantics encoding, where SymLM compresses callees into embeddings and fuses them into function embeddings. With more callees in O0 functions, this information loss increases.

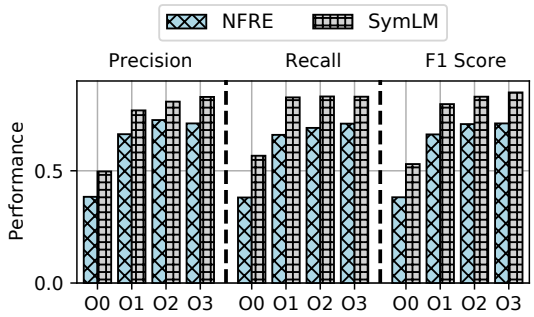


Figure 5: Performance of NFRE and SymLM on Predicting Function Names of x64 Binaries[6]

RQ1 Answer: SymLM is effective in predicting binary function names, and it achieves 0.634 precision, 0.677 recall, and 0.655 F1 score across different architectures and optimizations.

5.3 RQ2: Baseline Comparison

We compare SymLM with the state-of-the-art binary function name prediction tools, NERO [27] and NFRE [38]. According to their reported results, both tools outperform Debin [41], e.g., NERO is 39.4% more accurate than Debin. We have also run Debin's released models[7] on our datasets for 178 hours, and we obtained similar observations, e.g., it achieves 0.040 precision, 0.042 recall and 0.041 F1 score. Punstrip [73] is publicly available[8], but we cannot set it up based on its documentations. And the same problem was also reported by the other users[9], which has not been addressed yet. Therefore, we skip reporting the evaluation results of Debin and Punstrip.

Comparison with NFRE. NFRE's training code is not open sourced in its Github repository[10]. So we contacted NFRE's authors to request the code and received its scripts, IDA Pro [5] plugins, and data for training and evaluating models for x64 binaries. To evaluate NFRE, we first build a raw dataset of x64 binaries by running IDA Pro with the plugins. Next, we process this dataset based its original preprocessing steps [38], and train NFRE models on it. To give a fair comparison, we report NFRE's best performance and the evaluation results of SymLM that are trained and evaluated on the same training and test sets in Figure 5.

SymLM outperforms NFRE by 16.9%, 25.1%, and 22.1% in precision, recall and F1 score. Specifically, NFRE achieves the average performance of 0.621 precision, 0.610 recall, and 0.616 F1 score, and SymLM gets the average performance of 0.726 precision, 0.764

[5] https://www.gnu.org/software/bison/

[6] The data and codes received from NFRE's authors only work for x64 binaries.
[7] https://github.com/eth-sri/debin
[8] It was available in https://github.com/punstrip/punstrip at the time of paper writing.
[9] https://github.com/punstrip/punstrip/issues/1
[10] https://github.com/USTC-TTCN/NFRE

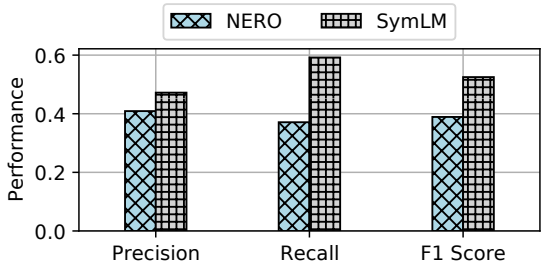
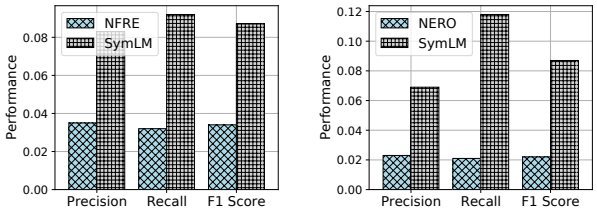


Figure 6: Performance of NERO and SymLM on NERO's Datasets



(a) SymLM v.s. NFRE **(b) SymLM v.s. NERO**

Figure 7: Generalizability of NFRE, NERO, and SymLM on Unknown Binary Functions

recall, and 0.751 F1 score. Among the three metrics, SymLM significantly outperforms NFRE on recall, which is important for many reverse engineering tasks, e.g., malware analysis [92] and disassembling [85]. While a high recall indicates a large true positive and a small false negative (Equation 18), it means SymLM can correctly predict more ground truth than NFRE. While training NFRE, we also observe that NFRE converts instructions into standard representations. For example, `push rbp` and `mov rbp, rsp` are converted into `INST_1325` and `INST_0b25`. This observation and the performance difference shown in Figure 5 confirm our hypothesis (§2.3): our function semantics encoding methods can achieve better performance than learning from hand-crafted features.

Comparison with NERO. The dataset and code of NERO have been open sourced in Github[11], so we evaluate its best model (the graph model) on its own dataset. NERO's dataset contains 483, 45 and 13 binaries for training, validation, and testing. We follow NERO's original settings to train the model, and we also train and evaluate SymLM on the same datasets. Figure 6 presents the evaluation results, in which SymLM outperforms NERO by 15.4%, 59.6%, and 35.0% in precision, recall, and F1 score. Similar to the results of NFRE, SymLM gains much better recall than NERO, which is important for downstream tasks. Since NERO discards all instructions that are not related to function call sites and arguments, we believe that the performance difference shown in Figure 6 demonstrates the importance of learning the full function semantics.

RQ2 Answer: SymLM is more effective than the state-of-the-art works. For example, it outperforms NERO by 15.4%, 59.6%, and 35.0% on precision, recall and F1 score.

5.4 RQ3: Generalizability and Obfuscation Resistance

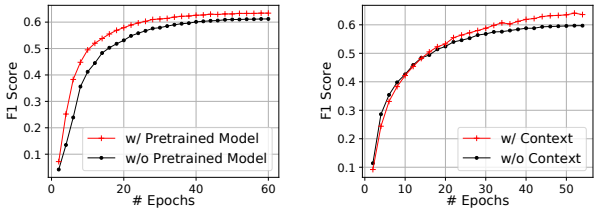
Generalizability. In the real-world deployment scenarios, test binaries are usually new to machine learning models, whose generalizability can be revealed by evaluations on unknown binaries. A poor generalizability means either the model is over-fitted, or its features are limited to its training sets. To examine the generalizability of SymLM, we first build an evaluation dataset with 540 binaries, compiled from 12 open sourced projects (e.g., usbutils [8]) with the same compilation setting as §5.1. We guarantee that the evaluation binaries have never been used for training before. Next, we directly run the previously trained SymLM and NFRE models on the evaluation dataset (with retraining) and report the results in Figure 7a. Compared with NFRE, SymLM achieves 137.1%, 187.5%, and 155.9% better performance on precision, recall and F1 score, which indicates a better generalizability.

Furthermore, we evaluate and compare the generalizability of SymLM and NERO from a data mining perspective, following the approach proposed by Gao et al. [38]. That is, in NERO's test set, some functions exist in both train and evaluation sets, which are considered as leaked samples. Testing the models on the non-leaked test functions can also show their generalizabilities [38]. Therefore, we first collect non-leaked functions from NERO's test set. Next, we evaluate NERO and SymLM with the collected functions and report their results in Figure 7b. It clearly shows the performance gap between NERO and SymLM, in which SymLM outperforms NERO by 200.0%, 461.9%, 295.5% on precision, recall and F1 score.

We believe that the main reason for the generalizability difference between SymLM and the baselines is that SymLM learns function semantics instead of the dataset-specific features. However, although we have demonstrated the better generalizability of SymLM, its overall performance is still not as good. We investigated and identified two potential reasons. First, the training and generalizability evaluation sets have different function name vocabularies, e.g., 11.8% of evaluation vocabulary words are not present in SymLM's training sets. Second, the KL divergence [77] between the generalizability evaluation and training sets is 2.4x larger than that between training and test sets used in §5.2, where the distribution shifts degrade SymLM's performance. We provide more discussions in §7.

Obfuscations Resistance. Real-world binaries are often obfuscated (particularly for malware), so we also examine how SymLM can resist obfuscations. Specifically, we show its performance by comparing with NFRE, which has the best reported performance among the baselines [38]. Table 5 presents the F1 scores of SymLM and NFRE evaluated on the orginial (None) and obfuscated (`bcf`, `cff`, `sub` and `split`) binaries (§5.1). For both SymLM and NFRE, we observe the performance degradation when obfuscation is applied to binaries. For example, the F1 scores of SymLM and NFRE drop by 6.1% and 17.5%, respectively, when `bcf` is applied. However, SymLM is more robust to obfuscations than NFRE. For example, the worst F1 score of SymLM is 0.726 in `sub` binaries, which is much better than the minimum F1 score (0.445 in `cff` binaries) of NFRE. Moreover, the average performance degradation of SymLM is 5.7%, which is much smaller than NFRE (18.6%). We believe that

[11] https://github.com/tech-srl/Nero



(a) Pretrained Model (b) Calling Context

Figure 8: Effectiveness of Pretrained Model and Calling Context

Table 5: F1 Scores of SymLM and NFRE on Obfuscation (OBF) Test Binaries. None indicates obfuscations are not applied.

OBF	SymLM	NFRE
None	0.806	0.595
bcf	0.757 (-6.1%)	0.491 (-17.5%)
cff	0.768 (-4.7%)	0.445 (-25.2%)
sub	0.726 (-9.9%)	0.505 (-15.2%)
split	0.788 (-2.2%)	0.496 (-16.6%)

the performance difference shown in Table 5 is reasonable because SymLM can learn function semantics, which will not change when obfuscations are applied.

RQ3 Answer: SymLM has the better generalizability and obfuscation resistance than the state-of-the-art works. For example, SymLM outperforms NERO by 200.0%, 461.9%, 295.5% in precision, recall and F1 score on unknown test binaries and it has 157% better F1 score than NFRE on obfuscated binaries.

5.5 RQ4: Ablation Study

To evaluate the effectiveness of SymLM's components, we perform a set of ablation studies in this section.

Pretrained Model. To understand how the pretrained model can help SymLM learn function semantics, we compare its performance trained with and without the pretrained model. Specifically, we evaluate and compare the performance when SymLM is (1) with the pretrained model, and (2) with the same microtrace-based model architecture (Figure 2) but randomly initialized model weights. Figure 8a presents the F1 scores of SymLM on the same test set after each training epoch. We observe that the pretrained model can significantly improve SymLM's performance after several training epochs, e.g., 52.7%, 33.6%, 25.1%, and 20.2% improvement in the first 5, 10, 15, and 20 epochs on average F1 score. Moreover, after being trained with more epochs, SymLM with and without the pretrained weights both converge to the stable performance, but SymLM with the pretrained model still outperforms SymLM without it.

Calling Context Modeling. We also study how our calling context modeling method can help improve SymLM's performance in a similar way. That is, we evaluate SymLM trained with and without considering calling context and report results in Figure 8b. The figure clearly shows the effectiveness of calling context on improve SymLM's optimal performance. For example, after SymLM achieves stable performance (after 45 epochs), the average F1 scores

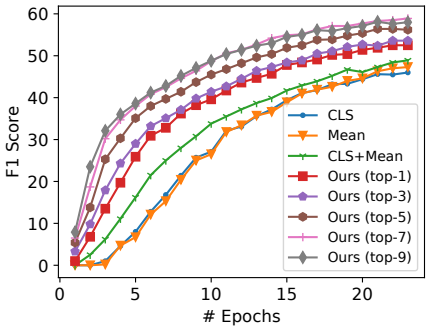
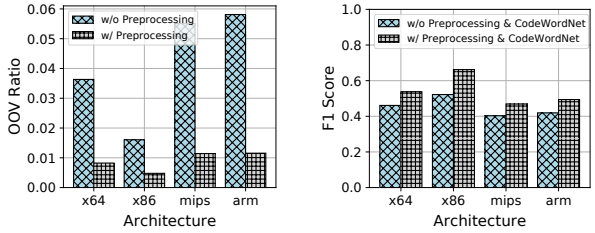


Figure 9: SymLM with Different Pooling Schemes



(a) OOV Word Mitigation (b) Performance Improvement

Figure 10: Effectiveness of Preprocessing and CodeWordNet on Mitigating OOV Words and Performance Improvement

of SymLM with and without context information are 0.596 and 0.643, which means learning calling context improves SymLM's performance by 7.9%. The effectiveness of calling context supports our observation: the calling context is an important component of function semantics, and fusing context information into function semantics indeed helps predict function names.

Internal Function Embedding Pooling. The internal function embedding pooling scheme is introduced to address the anisotropic issue in function token embeddings (§4.1). To verify its effectiveness, we compare SymLM with different pooling schemes, including (1) only using the `[CLS]` token embedding, (2) only using the mean of token embeddings (Mean), (3) concatenating both `[CLS]` token and the mean of all token embeddings (`[CLS]`+Mean), and (4) our proposed pooling scheme. For our pooling scheme, we also evaluate the parameter k of Equation 4 to demonstrate how it affects SymLM's performance. Figure 9 presents the evaluation results of SymLM with different pooling schemes in each epoch. It clearly shows the better performance of our proposed scheme compared to the baseline schemes (1-3). For example, the worst F1 score achieved by our scheme ($k = 1$) is better than those of (1) `[CLS]`, (2) Mean and (3) `[CLS]`+Mean schemes by 35.4%, 36.2%, and 18.6% on average, respectively. Moreover, among the baseline schemes (1-3), the `[CLS]` + Mean scheme achieves a better F1 score than the others, which means that both the [CLS] token and the mean of all token embeddings can only represent partial function semantics. We also find that increasing the parameter k can boost SymLM's performance till the upper bound, i.e., the maximum F1 scores are observed when $k = 7$.

Table 6: Performance of MLP and LSTM as Prediction Head

Model	Precision	Recall	F1-score
MLP	**0.497**	**0.567**	**0.530**
LSTM	0.464	0.362	0.406

Function Name Preprocessing and CodeWordNet. To validate the performance contributions of our function name preprocessing approaches, we first study the out-of-vocabulary (OOV) word ratios in test sets when function names are processed with and without our proposed preprocessing approaches. Specifically, for SymLM without preprocessing, we only tokenize function names into individual lower-case words based on our observed naming conventions (§4.2). We calculate the OOV ratio by the proportion of function name words of test sets that are not presented in the vocabulary of training sets. Figure 10a presents the OOV ratios in both settings among different architectures. Overall, the OOV word ratios are dramatically reduced from 4.14% to 0.90% across all architectures when our preprocessing approaches are enabled. In particular, the OOV ratio drops from 5.51% to 1.14% on MIPS binaries, which means 79.3% of the OOV words are mitigated. To further understand how our function name preprocessing approaches and the CodeWordNet module can help improve the performance of SymLM, we evaluate SymLM with and without them. Figure 10b presents the evaluation results on binaries across different architectures, which clearly demonstrates the effectiveness of our preprocessing approaches and the CodeWordNet module, such as they improve SymLM's F1 score by 16.7% on the MIPS binaries.

Decoder Model. The decoder model is to map function semantics encodings into names. For this, we also evaluate how our MLP decoder compares to the sequential model, which is commonly used in our baselines, as prediction head. Specifically, we compare the MLP model with the LSTM model [43], which was used as decoder for NFRE. We follow two steps to get the optimal training and inference results for fair comparisons while the LSTM model is used as SymLM's decoder. First, we enable "professor forcing" [58] that uses the ground truth from a prior time step as input in model training. Second, we use beam search [89] to get the optimal LSTM model prediction results, which generates function name words with considering the optimal probability of the activate candidates in the beam. Table 6 presents our evaluation results of SymLM with the MLP and LSTM decoders respectively, in which MLP outperforms LSTM by 7.1%, 55.8% and 30.5% on precision, recall and F1 score.

RQ4 Answer: SymLM's components can improve its performance on function name prediction. For example, our proposed preprocessing approaches and the CodeWordNet module can improve the F1 score of SymLM by 16.7% in MIPS binaries.

6 CASE STUDY

6.1 Qualitative Evaluation

To gain more insights of SymLM, we perform a qualitative study of the predicted names. Table 7 presents some examples of incorrect predictions. The first two columns are the ground-truth function names before and after preprocessing, and the third column shows the predicted function name words, in which the correctly predicted ones are underlined. We observe three categories of prediction errors: (*i*) our predictions miss some ground truth words, such as `log` for {`log`, `printf`}; (*ii*) our predictions include other words that are not in the ground truth, e.g., the predicted word `arg` is not in the ground truth {`quote`, `mem`}; and (*iii*) our predictions have the same semantic meanings as the ground truth but different words, e.g., `compare` and {`equality`, `comparator`}. By studying the errors, we find that some error predictions have captured the essential function semantics of the ground truth, while the calculated performance is not good, e.g., the F1 score of our prediction `compare` is 0 but it has the same meaning as its ground truth. That is, although the overall F1 score of SymLM is only 0.655 (§5.2), its real performance may be better than the reported number.

Table 7: Prediction Errors

Ground Truth		
Original	**Preprocessed**	**Prediction**
logprintf	log printf	printf
c_isascii	c is ascii	is ascii
bi_init	bi init	init
overwrite_string	overwrite string	string
quote_mem	quota mem	quote arg mem
find_non_slash	find non slash	find non slash u
hash_string	hash string	hash string csgre
equality_comparator	equality comparator	compare
do_exit	do exit	exit

```
uint32_t read(uint32_t ulPin){
  ...
  if (pin == NC) uVar3 = 0;
  else {
    uVar2 = read_value(pin);
    uVar3 = (uint32_t)uVar2;
    if (uVar4 != 0xc) {
      if ((uint) uVar4 < 0xc)
        return (uint)(uVar2 >> (0xcU - uVar4 & 0xff));
      return uVar3 << (uVar4 - 0xcU & 0xff);
    }
  }
  return uVar3;
}
```

Figure 11: The prediction result of the Gateway [2] firmware function which have been shown in Figure 1.

6.2 Firmware Analysis

To further examine the use case of SymLM, we study how it can be used to annotate the function in firmware. To collect test firmware binaries, we first investigate the open-source firmware datasets of existing firmware analysis works [36, 98]. Next, we identify and collect 8 32-bit ARM firmware images used by P2IM [36] from its Github[12]. Since the binaries are not stripped, we are able to collect the ground truth function names. From the binaries, we successfully obtain 1,061 functions by our binary parser (§4.1). Next, we preprocess the ground truth with our proposed approaches (§4.2). To predict the function names of the firmware binaries, we reuse SymLM that is trained on our own ARM binary

[12] https://github.com/RiS3-Lab/p2im-real_firmware/tree/master/binary

dataset and the corresponding vocabulary. After preprocessing, we observe that 18.2% of the function name words do not appear in our ARM dataset vocabulary. By manual investigations, we find that most of OOV words are IoT or firmware specific terms, such as `HAL`, `NVIC`, and `MPU`. Since our own ARM datasets do not contain any firmware images, it is reasonable to observe the high OOV ratio.

To mimic real-world deployment scenarios where binaries are usually stripped, we predict function names without using any debug symbols. With SymLM, we successfully obtain partially correct predicted names for 172 functions out of 1,061 functions. Figure 11 presents the prediction result of the `Gateway` [2] firmware function that has been shown in Figure 1, where SymLM predicts names of this function and its callee as {`read`} and {`read`, `value`}. We use the common function name delimiter to combine {`read`, `value`} as the predicted function name `read_value` with preserving the original word order. According to the binary's debug symbols, the ground truth function names of these two functions are `analogRead` and `abc_read_value` as shown in Figure 1. By studying the decompiled code of these functions, we find that the word `analog` is a term related to signal processing and `abc` is related to the devices that use the firmware. And both words are the OOV words for SymLM. Since SymLM was not trained on the firmware images, we think it is reasonable to miss these two OOV words in the prediction results. And this example shows that SymLM is able to predict the function names that preserve key function semantics.

7 DISCUSSION

This section discusses the limitations of our work and future directions. We consider the following cases that can bias our evaluation results, and addressing them can all be interesting future works.

Dataset Size. We only train and evaluate SymLM with binaries compiled from 27 open-source projects. A larger-scale study that includes more binaries from the other sources, e.g., Linux kernel, may further improve SymLM's performance, though it has shown the better generalizability than prior works, as presented in §5.4. Moreover, generalizability evaluations can be performed in different ways, e.g., cross optimizations, cross architectures, and cross unseen projects, while we focus on cross unseen projects in this paper. We believe that evaluating the generalizability in other settings can be interesting future studies.

Obfuscation. While we have evaluated SymLM against several compiler-based obfuscations, we do not consider other types of obfuscations such as encryption and packing [45]. We treat them as an orthogonal problem and any advancement in handling them [18] is complementary to our approach.

Operating System. We focus only on Linux binaries in this paper. However, programs for other OSs can have the different calling conventions, in which SymLM's calling context semantics encoding module may fail to handle the effect of function calls.

Noise in Ground Truth. While we have attempted to address the ambiguous function name issues, the problem is not fully resolved, due to the limitations [80] of the word embedding algorithms used in SymLM. Moreover, we only consider the semantic similarity at the word level, but do not consider similar phrases with multiple words. We leave addressing these problems in our future research.

8 RELATED WORKS

While we have discussed many related works in §2, in this section, we review other additional related works.

Function Name Prediction. Function name prediction is a challenging task for both source and binary code. For source code, there are many research efforts to predict function names for name inconsistency checking and recommendations [9, 51, 60, 64, 70]. For example, Nguyen et al. [70] propose an abstractive function summarization model to check name consistency. Li et al. [60] introduce a context-based representation learning method. Meanwhile, the research community also treats it as a task of code summarization [11, 47, 61, 62, 96], where a brief description is generated from source code. Allamanis et al. [11] propose a convolutional attention model to describe the source code. Zhang et al. [96] present a retrieval-based neural source code summarization method. However, binary code contains far less semantic information than source code, making it hard to apply existing methods to inferring function names of stripped binaries. As discussed in §2.2, SymLM differs from these prior works by learning the function execution behavior through context-sensitive and execution-aware code embeddings.

Machine Learning for Reverse Engineering. Recently, machine learning has been widely used in many binary reverse engineering tasks, including function similarity detection [31, 32, 66, 77], type inference [10, 75], variable name recovery [13, 22], value-set analysis [40], and disassembling [14, 37, 68, 76, 94]. For example, Asm2vec [31] and DeepBinDiff [32] use unsupervised learning methods to learn binary code representations. Marcelli et al. [66] evaluate the start-of-the-art learning-based binary function similarity works on two benchmarks. DIRTY [22] infers variable names and types with a transformer-based model. VarBERT [13] recovers the variable names with a BERT model. DeepDi [94] uses a relational graph model to accelerate the disassembly task. XDA [76] leverages transfer learning for accurate and robust disassembly. Although these works have achieved dramatic performance in their own tasks, none of them can predict binary function names.

9 CONCLUSION

We have presented SymLM, a novel neural architecture for predicting binary function names by learning context-sensitive behavior-aware code embeddings. We designed a novel semantics fusing encoder module to model function execution behavior semantics, CodeWordNet to encode function name meanings, and proprocessing approaches to solve the OOV problems. Our evaluation results showed that SymLM is up to 35.0% more accurate than state-of-the-art tools, while demonstrating its generalizability, obfuscation resistance, component effectiveness and potential usability.

ACKNOWLEDGMENTS

We thank Han Gao and Guoqiang Chen for helping us evaluate NFRE. We also thank the anonymous reviewers for their insightful comments. This research was supported in part by ARO award W911NF2110081, DARPA award N6600120C4020, and NSF awards 1834215 and 2112471, as well as a research gift from Amazon. Any opinions, findings, conclusions, or suggestions expressed are the authors' and not necessarily those of the sponsors.

REFERENCES

[1] "Coreutils - gnu core utilities," https://www.gnu.org/software/coreutils/, accessed: 2022-04-14.

[2] "Gateway," https://github.com/RiS3-Lab/p2im-real_firmware/blob/master/binary/Gateway, accessed: 2022-04-26.

[3] "Gnu binutilss," https://www.gnu.org/software/binutils/, accessed: 2022-04-14.

[4] "Hikari," https://github.com/HikariObfuscator/Hikari#hikari, accessed: 2022-03-14.

[5] "Ida pro," https://hex-rays.com/ida-pro/, accessed: 2022-04-14.

[6] "Linux system call table," https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md, accessed: 2022-04-14.

[7] "Options that control optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, accessed: 2022-08-29.

[8] "usbutils," https://github.com/gregkh/usbutils, accessed: 2022-04-11.

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

[10] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[11] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

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

[14] E. Bauman, Z. Lin, and K. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2018.

[15] K. Beck, *Implementation patterns*. Pearson Education, 2007.

[16] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: the case of single-letter variables," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 45–54.

[17] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[18] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 643–659.

[19] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[20] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary Code Extraction and Interface Identification for Security Applications," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2010.

[21] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 678–689.

[22] 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.

[23] S. Chen, Z. Lin, and Y. Zhang, "SelectiveTaint: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1665–1682.

[24] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1960–1979, 2019.

[25] K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[26] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[27] 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.

[28] Derek Anderson and Scott Randal, "Word ninja," https://github.com/keredson/wordninja, accessed: 2022-02-26.

[29] 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.

[30] Devopedia, "Naming conventions," https://devopedia.org/naming-conventions, accessed: 2022-02-15.

[31] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[32] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Security Symposium*, 2020.

[33] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.

[34] A. Farghaly and K. Shaalan, "Arabic natural language processing: Challenges and solutions," *ACM Transactions on Asian Language Information Processing (TALIP)*, vol. 8, no. 4, pp. 1–22, 2009.

[35] D. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer, "How developers choose names," *IEEE Transactions on Software Engineering*, 2020.

[36] 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.

[37] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1075–1092.

[38] 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.

[39] C. Gulcehre, S. Ahn, R. Nallapati, B. Zhou, and Y. Bengio, "Pointing the unknown words," *arXiv preprint arXiv:1603.08148*, 2016.

[40] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "{DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1787–1804.

[41] 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.

[42] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[43] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[44] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 217–227.

[45] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, "Diversification and obfuscation techniques for software security: A systematic literature review," *Information and Software Technology*, vol. 104, pp. 72–93, 2018.

[46] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 294–317.

[47] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[48] J. Huang, D. Tang, W. Zhong, S. Lu, L. Shou, M. Gong, D. Jiang, and N. Duan, "Whiteningbert: An easy unsupervised sentence embedding approach," *arXiv preprint arXiv:2104.01767*, 2021.

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

[50] Jeff Burt, "How ai can help reverse-engineer malware: Predicting function names of code," https://www.theregister.com/2022/03/26/machine_learning_malware/, accessed: 2022-04-26.

[51] L. Jiang, H. Liu, and H. Jiang, "Machine learning based automated method name recommendation: How far are we," in *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE'19). IEEE CS*, 2019.

[52] Y. Jiang, H. Liu, and L. Zhang, "Semantic relation based expansion of abbreviations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 131–141.

[53] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 346–356.

[54] D. Khurana, A. Koli, K. Khatter, and S. Singh, "Natural language processing: State of the art, current trends and challenges," *arXiv preprint arXiv:1708.05148*, 2017.

[55] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," *arXiv preprint arXiv:1804.10959*, 2018.

[56] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

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

[58] A. M. Lamb, A. G. ALIAS PARTH GOYAL, Y. Zhang, S. Zhang, A. C. Courville, and Y. Bengio, "Professor forcing: A new algorithm for training recurrent networks," *Advances in neural information processing systems*, vol. 29, 2016.

[59] B. Li, H. Zhou, J. He, M. Wang, Y. Yang, and L. Li, "On the sentence embeddings from pre-trained language models," *arXiv preprint arXiv:2011.05864*, 2020.

[60] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 574–586.

[61] Y. Liang and K. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[62] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 184–195.

[63] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.

[64] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1–12.

[65] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 475–487.

[66] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *USENIX 2022, 31st USENIX Security Symposium, 10-12 August 2022, Boston, MA, USA*, Usenix, Ed., Boston, 2022.

[67] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[68] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE'19, Montreal, Quebec, Canada, 2019, pp. 1187–1198.

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

[70] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1372–1384.

[71] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," *arXiv preprint arXiv:1904.01038*, 2019.

[72] 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.

[73] 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.

[74] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[75] 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.

[76] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.

[77] 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.

[78] Pytorch developers, "Embedding," https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html, accessed: 2022-03-24.

[79] R. Rehurek and P. Sojka, "Gensim–python framework for vector space modelling," *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, vol. 3, no. 2, 2011.

[80] S. Ruder, I. Vulić, and A. Søgaard, "A survey of cross-lingual word embedding models," *Journal of Artificial Intelligence Research*, vol. 65, pp. 569–631, 2019.

[81] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 2, pp. 1–43, 2017.

[82] S. Seeha, I. Bilan, L. M. Sanchez, J. Huber, M. Matuschek, and H. Schütze, "Thailmcut: Unsupervised pretraining for thai word segmentation," in *Proceedings of The 12th Language Resources and Evaluation Conference*, 2020, pp. 6947–6957.

[83] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[84] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *Network and Distributed System Security Symposium*. Citeseer, 2008.

[85] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *Network and Distributed System Security Symposium*, vol. 1, 2015, pp. 1–1.

[86] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 13–20.

[87] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003, pp. 252–259.

[88] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[89] A. K. Vijayakumar, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. Crandall, and D. Batra, "Diverse beam search: Decoding diverse solutions from neural sequence models," *arXiv preprint arXiv:1610.02424*, 2016.

[90] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *arXiv preprint arXiv:1711.07163*, 2017.

[91] K. Wang and Z. Su, "Blended, precise semantic program embeddings," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 121–134.

[92] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Malware analysis of imaged binary samples by convolutional neural network with attention mechanism," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 127–134.

[93] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, 2021.

[94] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[95] J. Zeng, Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation-resilient binary code reuse through trace-oriented programming," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, Berlin, Germany, November 2013.

[96] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[97] C. Zhao and S. Sahni, "String correction using the damerau-levenshtein distance," *BMC bioinformatics*, vol. 20, no. 11, pp. 1–28, 2019.

[98] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *USENIX Security Symposium*, 2021, pp. 2007–2024.