A Transformer-based Function Symbol Name Inference Model from an Assembly Language for Binary Reversing

HyunJin Kim
Sungkyunkwan University
Suwon, South Korea
khyunjin1993@skku.edu

JinYeong Bak
Sungkyunkwan University
Suwon, South Korea
jy.bak@skku.edu

Kyunghyun Cho
New York University
New York, USA
kyunghyun.cho@nyu.edu

Hyungjoon Koo*
Sungkyunkwan University
Suwon, South Korea
kevin.koo@skku.edu

ABSTRACT

Reverse engineering of a stripped binary has a wide range of applications, yet it is challenging mainly due to the lack of contextually useful information within. Once debugging symbols (*e.g.*, variable names, types, function names) are discarded, recovering such information is not technically viable with traditional approaches like static or dynamic binary analysis. We focus on a function symbol name recovery, which allows a reverse engineer to gain a quick overview of an unseen binary. The key insight is that a well-developed program labels a meaningful function name that describes its underlying semantics well. In this paper, we present AsmDepictor, the Transformer-based framework that generates a function symbol name from a set of assembly codes (*i.e.*, machine instructions), which consists of three major components: binary code refinement, model training, and inference. To this end, we conduct systematic experiments on the effectiveness of code refinement that can enhance an overall performance. We introduce the per-layer positional embedding and Unique-softmax for AsmDepictor so that both can aid to capture a better relationship between tokens. Lastly, we devise a novel evaluation metric tailored for a short description length, the Jaccard* score. Our empirical evaluation shows that the performance of AsmDepictor by far surpasses that of the state-of-the-art models up to around 400%. The best AsmDepictor model achieves an F1 of 71.5 and Jaccard* of 75.4.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

reversing, assembly, function name, neural networks, Transformer

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0098-9/23/07…$15.00
https://doi.org/10.1145/3579856.3582823

ACM Reference Format:
HyunJin Kim, JinYeong Bak, Kyunghyun Cho, and Hyungjoon Koo. 2023. A Transformer-based Function Symbol Name Inference Model from an Assembly Language for Binary Reversing. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579856.3582823

1 INTRODUCTION

Reverse engineering (reversing) is a process of attempting to comprehend an unknown component or a product by dismantling or reasoning it. Software is one of common applications for such reversing because high-level semantic information has been discarded (*i.e.*, stripped) in an executable binary. Indeed, binary reversing has a wide spectrum of applications including software copyright infringement [10], binary similarity detection [16, 26, 29, 79], malware-related study [9, 11, 33, 40, 44], vulnerability discovery [12, 13, 19, 23, 50, 54, 61, 71, 74], and digital forensics [57, 66].

However, reversing a stripped binary is non-trivial because it holds a compact representation with machine code instructions, causing fruitful information unavailable in source code like variable names and types, function names and parameters, and structure information. Even equipped with various automation tools (*e.g.*, disassembler and decompiler in Figure 1), a binary reversing task requires experts' knowledge, skills, and insights with tedious manual efforts [53]. Decompilation is one of popular means to deduce the behavior of a binary by transforming an assembly to a high-level language, however, it still lacks syntactic and semantic information. In general, the contextual recovery from the absence of such information is technically yet viable with traditional approaches like static or dynamic binary analysis.

In response, recent advancements borrow the idea of natural language processing (NLP) for binary analysis, training an inference model via a deep neural network (DNN). DEEPVSA [28] proposes a novel approach that can improve the capability of a value set analysis (VSA) with a sequence-to-sequence DNN by inferring a memory region that VSA fails to identify. Structure2vec [54] suggests an automated feature extraction from a control flow graph (CFG), applying it to a binary similarity detection task. Another noticeable direction is to reconstruct debugging symbol information in a stripped binary, including a variable name [14, 30, 45], a variable type [14], and a function name [17, 25, 30]. It is noted that

a binary-oriented approach differs from a source-centric one like source code summarization [3, 4, 15, 77] in that the former must handle machine-interpretable code. Dire [45] leverages a decompiler's internal abstract syntax tree (AST) representation to recover a variable name. Similarly, Debin [30] targets the recovery of both symbol names and types. Close to our work, NERO [17] predicts procedure names by utilizing enriched representations of call sites, and SymLM [36] models the execution behavior of a calling context and instructions; however, the main downside is that it requires a call invocation. Similarly, NFRE [25] proposes a lightweight framework that reassigns a function name with a better performance than Debin [30] and Nero [17]. DIRTY [14] infers variable types and names based on Transformer [75], aiding further binary reversing. Yet, none of the above approaches thoroughly studies the properties of an assembly language (as data), and carefully considers those characteristics in designing a DNN model.

In this paper, we present AsmDepictor that allows for inferring an original function symbol by directly learning from an assembly. Inspired by InnerEye [82], we view this problem as a *translation from an assembly language to a natural language that describes a function.* The translation differs from a summarization task in that the former generates a series of outputs in another language whereas the latter in the same one. Another key insight behind this setting is that a function symbol name of a well-developed program often follows a meaningful label that well describes its underlying semantics. It would be greatly beneficial for a reverse engineer to quickly delineate the basic intent of an unseen binary with a list of (reliably) inferred function symbol names since a binary typically contains multiple binary functions (§3.2).

AsmDepictor consists of three main components that deal with data refinement, model training, and a practical inference engine for fulfilling our objective. To the best of our knowledge, we first conduct systematic experiments on the effectiveness of binary codes' normalization and tokenization (for a better representation) before feeding them into an actual neural network. In essence, our empirical finding shows that the best data refinement strategy is tokenizing an assembly code solely with the byte-pair encoding (BPE) algorithm [69], and without any code normalization. We observe that BPE can indeed address a known vocabulary problem (*e.g.*, out-of-vocabulary) as proposed by Karampatsis et al. [37]. Second, we adopt Transformer that fits well on our sequence-to-sequence neural machine translation (*i.e.*, predicting a function symbol name from a set of machine instructions). To enhance the performance of the AsmDepictor model, we introduce the following techniques with Transformer: ① a *per-layer positional embedding* scheme instead of utilizing a positional encoding from the naïve Transformer architecture, and ② the *Unique-softmax* activation function that assists to capture better relationship of tokens, and ③ *layer reduction* (*e.g.*, from six to three layers) that prevents attention values' diminishing at the upper layers. It is noteworthy mentioning that the above design choices are indeed based on our thorough experiments for a better data representation in practice. Besides, we devise a novel evaluation metric, dubbed the Jaccard* score, which is specialized in a short output generation task (a function symbol typically consists of a limited number of words) by considering both an order and a brevity penalty. Third, the AsmDepictor inference engine can produce a list of candidate function symbol names as well as a predicted one because, by nature, a sequence of assembly instructions possibly maps into multiple functions. We maintain such function candidates at the time of data refinement.

Our experiments demonstrate that AsmDepictor outperforms the state-of-the-art baseline models (*e.g.*, Debin [30], NERO [17]), by a wide margin (*i.e.*, up to four times better performance). Besides, AsmDepictor achieves an F1 of 71.5 (*i.e.*, accurately predicting seven out of 10) with a large volume of dataset (around $520K$ functions). The main contributions of our work are as follow.

- We propose AsmDepictor, the full-fledged Transformer-based framework that assists binary reversing, which can infer a function symbol name from an assembly language.
- We systematically explore a better data representation means (*i.e.*, normalization and tokenization) of both an assembly code (input) and a function symbol (output) for further efficient learning.
- We introduce two main ideas for building AsmDepictor: per-layer positional embedding and Unique-softmax to enhance overall model performance.
- We devise a model evaluation metric, Jaccard* tailored for a short output (*e.g.*, function name) generation task.
- We thoroughly evaluate the practicality and effectiveness of AsmDepictor, demonstrating that our model by far surpasses state-of-the-art baseline models.

We have open-sourced AsmDepictor [1] to foster further research in the domain of binary analysis with a deep neural network.

2 BACKGROUND

Encoder-decoder Architecture. A recent advancement to handle variable-length (end-to-end) input and output texts is an encoder-decoder architecture [7, 51], which is particularly suitable for a machine translation task (*e.g.*, producing an output by taking an arbitrary size of an input). Transformer [7] leverages the encoder-decoder architecture at its core implementation into sequence-to-sequence transformations. Unlike prior recurrent neural network (RNN) models, technical enhancements like multi-head self-attention and positional encoding in Transformer enable one to capture long-range dependencies. The main component of Transformer is a scaled dot-product attention: $\mathrm{Attn}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$ where (Q, K, V), and d_k denote (queries, keys, values) matrices and the scaling factor of a dimension, respectively. Notably, multiple heads contain different projections (*i.e.*, mapping between Q and KV pairs) to capture the relationship between words within a sentence. An encoder-decoder architecture learns both the semantics of an input (*e.g.*, machine instruction) within an encoder and that of an output (*e.g.*, function symbol) that is associated with the input within a decoder, rendering a translation task possible (*i.e.*, multi-head attention). Besides, the structure allows for parallel computation on GPUs, making it scalable to build a model on a real-world dataset, rising the popularity of Transformer-based architectures such as GPT (Generative Pre-trained Transformer) [64] and BERT (Bidirectional Encoder Representations from Transformers) [20]. In this regard, we harness Transformer for our sequence-to-sequence

[1] https://github.com/agwaBom/AsmDepictor

transformation task that predicts a function symbol name from machine instructions.

Approaches for a Position Representation. The position of a word is one of the significant factors to deduce the contextual meaning of a sentence in the area of NLP. The main approaches to represent a position in a neural network are through either a positional encoding or a positional embedding. One approach that represents positional information is to assign a hard-coded value to a position in a sentence. For example, the original Transformer [75] utilizes an absolute sinusoidal value to each token in a sequence. Another approach that indicates a position is a positional embedding by learning it (*i.e.*, learned positional encoding) while training. For instance, BERT [20] adopts a positional encoding scheme via learnable parameters. In this paper, we harness a positional embedding for better expressiveness.

Byte-Pair Encoding. The design of the BPE [69] algorithm originates from data compression. Simply put, BPE segments a word into frequently appearing subwords as follow: ① seeking a consecutive character pair, ② defining the pair as another character (*i.e.*, byte pair), ③ replacing the pair with the new character, and ④ iteratively performing ① - ③ with a parameter. To exemplify, "playing" can be split into "play@@" and "ing" when the subword "ing" frequently re-appears in a training set. Lately, BPE has been applied to the domain of NLP to tackle an OOV (out-of-vocabulary) problem due to the enormous size of vocabularies. Karamptsis et al. [37] demonstrate that BPE could properly handle an extremely large and sparse vocabulary (*e.g.*, millions of unique tokens) as well as OOV when building open-vocabulary models for source code on a large scale. To the best of our knowledge, we first adopt BPE to an assembly language.

Evaluation Metrics for Generative Models. The two widely adopted metrics for a language generation model have been introduced to evaluate how a machine-translating sentence is close enough to a human speech or writing: BLEU (Bilingual Evaluation Understudy) [58] and Rouge-l (Recall-Oriented Understudy for Gisting Evaluation) [49] scores. A BLEU score calculates the occurrence of matching words (*i.e.*, n-gram) in the candidate translation over the reference sentence irrespective of the order of words. Meanwhile, a Rouge-l score measures the longest matching sequence of words using the longest common subsequence (*i.e.*, recall), capturing its appearance in the reference sentence. In this respect, generating as many matching words as possible achieves a high BLEU while predicting a long matching sequence leads to a high Rouge-l. However, both BLEU and Rouge-l scores are inappropriate for assessing a relatively short sequence of words (*e.g.*, function identifier), proposing the Jaccard* score for our purpose (See §4.3).

3 FUNCTION SYMBOL INFERENCE FROM ASSEMBLY CODE

3.1 Problem Definition and Goal

Problem Definition. We informally define a machine code (*i.e.*, assembly language) description problem as *inferring an original function symbol that best describes its behavior* from machine instructions (*i.e.*, assembly) within a function boundary in a stripped binary. Note that we employ a common term of a "function symbol" as part of debugging symbols that are available in the symbol table of a non-stripped binary, indicating a function identifier (name) from a source. In this paper, we use those terms interchangeably.

Goal. The objective of a function symbol name inference is to quickly portray the intent of an unseen binary by delineating a chunk of instructions at the binary function level, which aids further binary reversing. To this end, we borrow the concept of a language translation for learning a function symbol name corresponding to an assembly code. Note that a translation is a communication between a source (machine code) and a target (natural language). In this paper, we harness the Transformer architecture [75] that has lately achieved great success on neural machine translation tasks.

Assumption. The two key insights are that ① a well-developed program that maintains a function label that describes a code snippet well, and ② the transformation from an assembly to a function identifier well fits into another neural machine translation application because the distribution of the language follows the Zipf's law [62] like Figure 7 in Appendix. Oftentimes naming a function follows its convention [78], which allows one to read and understand source code using a handy rule such as multiple-word identifiers. For instance, one may utilize delimiter-separated words like an underbar (*i.e.*, snake case; *e.g.*, `get_user_groups`), or case-separated words that indicate word boundaries with medial capitalization (*i.e.*, camel case; *e.g.*, `getUserGroups`). We presume that a binary has been compiled with a high optimization level (*i.e.*, -O2 or -O3) because taking a low optimization level (*e.g.*, -O0 or -O1) has been rarely seen in practice. Our experiment utilizes the `x86-64` assembly language for Intel 64-bit processors. Recognizing a function relies on a reversing tool such as IDA Pro [2], angr [72], Radare [63], and Ghidra [56]. A function boundary identification problem in a binary [5, 6, 8, 43, 70, 76] is beyond the scope of this paper.

3.2 Demonstrative Example

The essence of binary reversing lies in an in-depth comprehension via the analysis of an unseen binary, inferring the contextual semantics of a program at a high level rather than evaluating every single instruction. Albeit varying automation tools for reversing such as a disassembler, unpacker, emulator, binary similarity detector, or decompiler, it still requires expertise knowledge with tedious manual efforts. Figure 1 illustrates how our approach could assist in reversing a binary. In a nutshell, we leverage a neural machine translation technique to generate a model to predict a function symbol from learned machine instructions. The example demonstrates that a reversing practitioner could gain a quick glimpse of a binary with a list of (inferred) function symbols (❸ in Figure 1). A binary analysis tool represents a sequence of machine instructions with a function boundary (*e.g.*, IDA [2]). Intuitively, the actual function symbol, `ipc_sem_free_info`, can offer better contextual information for `FUN_00108d70` (mechanically generated by *e.g.*, Ghidra [56]) although the decompiler [68] provides a high level code in the C programming language (❷ in Figure 1).

3.3 Challenges

Refining data (*i.e.*, pre-processing an assembly) is one of the most significant tasks for generating a model that expects the best performance [73]. Unlike generating a translation model in NLP, a set of

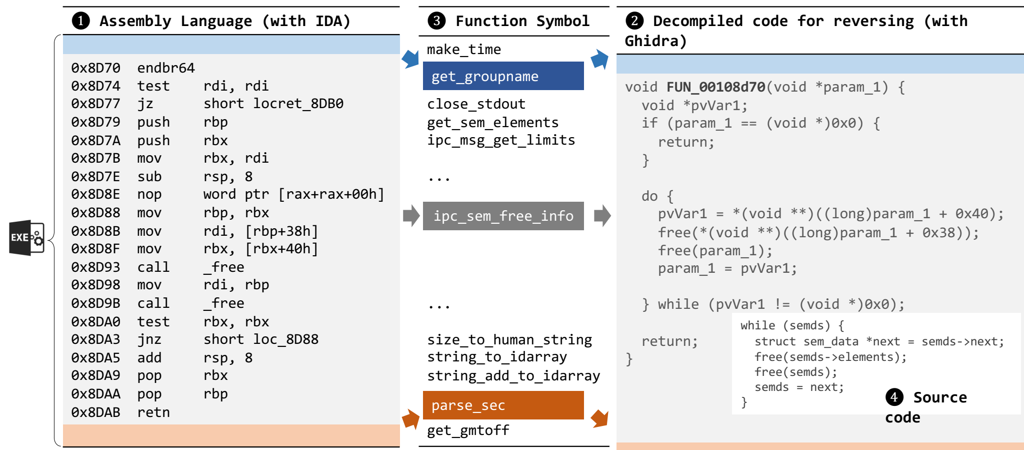


Figure 1: Illustrative example of a typical binary analysis. Given an unseen binary, a reverse engineer often utilizes a disassembler (*e.g.*, ❶ IDA [2]) or a decompiler (*e.g.*, ❷ Ghidra [56]) to deduce the underlying semantics of the source codes (*e.g.*, ❹). If one could obtain a list of function symbols within the binary close to the ground truth, it would be considerably fruitful for further analysis. We aim to generate such a model to directly learn ❸ a function symbol (available from debugging information) from an assembly language (❶) like a natural language translation, aiding to quickly infer the contextual meaning of a binary. In this example, the actual function symbol for `FUN_00108d70` from the decompiled code (❷) is `ipc_sem_free_info`.

input vocabularies from machine instructions encompasses a vast number of words as well as immensely sparse ones. In an assembly code, the operand of `call` or `jmp` family instructions represents an absolute address or a relative offset of a target for invoking a function or being jumped to. Similarly, an immediate value contains a constant that is loaded into a memory/register for performing an arithmetic/logical operation. Considering a huge number of such possible operands and immediate values (*e.g.*, four billion with four bytes), computing every word as a token would be not only prohibitively expensive but also suffering from an OOV problem (*i.e.*, training all words beforehand is not possible) and rarely appeared words (*i.e.*, meaningful embedding updates are failed due to the insufficient number of appearances) [37]. This problem must be taken into account for an output language because a software developer can create an arbitrary function identifier.

4 ASMDEPICTOR DESIGN

4.1 Design Overview

Figure 2 illustrates an overall workflow of AsmDepictor, which comprises three modules.

Data Refinement. A set of machine instructions (*i.e.*, assembly) in a function provides an accurate description of a certain task performed by a processor. Although learning an assembly as it stands may offer rich information (presumably the richest) for building a model, it is quite impractical because a means of vectorization per instruction not only requires too much computational resources, but also suffers from meaningful embedding generation due to both OOV and sparse instructions as stated in §3.3. Recall that we aim to deduce a function symbol (like a debugging symbol available from a non-stripped binary). Thus, it is essential to transform machine codes to an appropriate form for training a model with them by striking a balance between holding a substantial amount of information of machine codes and tackling the issues of both sparsity

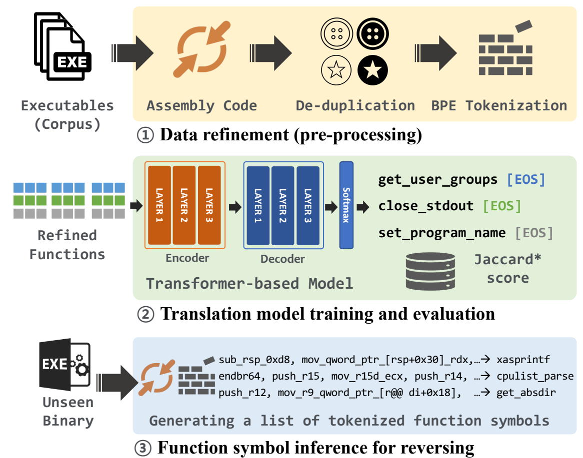


Figure 2: Overall workflow of AsmDepictor. In preparation for training, we ① refine machine instructions (input) and function symbols (output) from a corpus (§4.2). We ② build a Transformer-based generative model and assess it with our Jaccard* score (§4.3). The model ③ predicts a function symbol, aiding a binary reversing task (§4.4).

and OOV in practice. Besides, we should take duplicate function bodies into account because they hinder appropriate learning (*e.g.*, a generative model may be confusing when encountering different function symbols with the same function body). We adopt a data refinement process including function de-duplication and BPE tokenization per instruction with thorough experiments (§4.2).

Model Training. Our initial attempt with the naïve Transformer (using the pre-processed dataset with previous normalization approaches) failed to train a model. We discover that a code normalization inadvertently encounters a case to barely see any relationship between instructions on the upper layers of Transformer (Interested

readers refer to Figure 8 in Appendix). For example, replacing 64-bit registers (*e.g.*, `rax`, `rcx`, `r9`) with a symbol that represent a 64-bit register (*e.g.*, `reg8`) [80] can produce consecutively identical words in a function prologue or epilogue (*e.g.*, `pop reg8`, `pop reg8`). To this end, we establish a handful of strategies for efficiently learning a function symbol from an assembly (§4.3) by ① reducing the number of layers (from six to three), ② introducing a per-layer positional embedding for better expressiveness, and ③ devising an activation function dubbed Unique-softmax.

Function Inference. Once a model is ready, we can deduce a function symbol by taking a sequence of (disassembled and BPE-tokenized) machine codes as an input. Then, the model generates an output ending with the <EOS> token (*i.e.*, special token for the end of a sentence). As a produced output often contains a short number of words, we devise an evaluation metric, the Jaccard* score (§4.3).

4.2 Assembly Codes Refinement

Function De-duplication. Our finding shows two common cases as follow: ① the identical function body with different function symbols, and ② the identical function identifier with different function bodies. The former case is often shown when one can label a function with a similar (or the same) procedure, or a function that reduces a relatively short routine after compiler optimizations (*e.g.*, approximately 22% as in Table 1), which aligns with the observation in NFRE [25]. For example, we observed a few samples that a single function body maps to more than a thousand function symbols in our dataset, which consists of a small number of tokens. In the latter case, we randomly include one of the function identifiers and its function body for training while excluding others because this assists to learn varying patterns of a single function symbol (in terms of code semantics) by exposing different function bodies. Note that we maintain a mapping between the selected function symbol (as a key) and other symbols (as a value) so that they could be part of an inferred output (Table 6).

Code Normalization and Tokenization. In essence, taking an assembly code to a neural network requires to handle three main vocabulary issues: ① total number of vocabularies, ② out-of-vocabulary problem, and ③ sparse vocabulary problem. Hence, transforming an instruction into an appropriate form is essential because it serves a basis for vectorizing instructions as an internal data representation. Prior work [48, 55, 80, 82] pre-process an assembly language with in-house rules before feeding it into a neural network. Hereinafter, we call such an overall conversion code refinement that includes code normalization and tokenization. With extensive experiments for such normalization and tokenization techniques (§A.2 in Appendix), we conclude that the best choice to handle the vocabulary issues while keeping a performance is *the strategy of BPE tokenization without code normalization.* It is noted that we merely separate word boundaries without BPE for function symbols with an underbar (snake case naming) or a medial capital (camel case naming).

4.3 AsmDepictor Model

Per-Layer Positional Embedding. A human can understand the meaning of a sentence even if the words in the sentence are scrambled. However, in assembly, the order of machine instructions (*e.g.*, topological order in a control flow graph) plays a crucial role in precisely performing a desirable task. Hence, providing high-quality positional information is needed to make a model better understand the pattern of a sequence of instructions. To this end, we adopt an absolute positional embedding of BERT [35] instead of a sinusoidal positional encoding in vanilla Transformer. Unlike BERT that applies a positional embedding only before the first layer of an encoder layer, we introduce a *per-layer positional embedding*, providing a positional representation at each layer of the encoder layer to prevent the disappearance of positional information on the upper layer. Figure 4 illustrates how positional information is added to each encoder layer. It is noted that positional information is not included as Transformer can learn it by itself [34, 67] in a decoding phase (*i.e.*, masking effect).

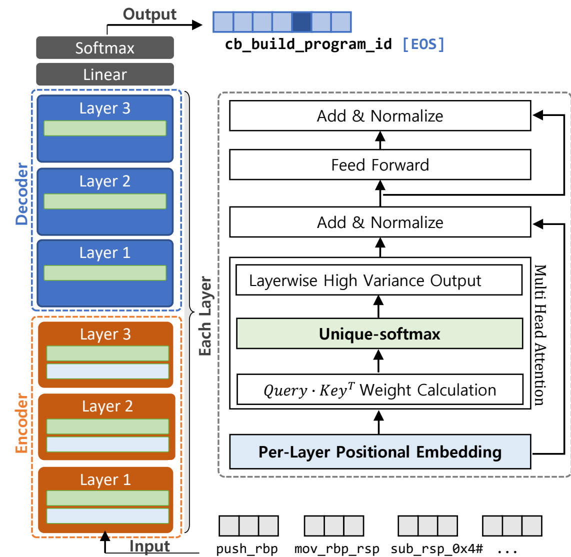


Figure 3: AsmDepictor architecture that consists of a stacked Transformer-based encoder and decoder. We adopt a per-layer positional embedding (at encoders) for learning the positional representation of an assembly, and a Unique-softmax function (at both encoders and decoders) for better quality of vectors per each layer, leading a high performance on a function symbol inference task.

Unique-softmax Function. One of common characteristics in NLP is a frequent appearance of stop words [22] like articles (*e.g.*, 'a', 'the', 'of'), which conveys little contextual meanings (thus often ignored). Although a function may include a few instructions that stay away from the original context like the `nop` operation (at the address of `0x8D8E` in Figure 1), each instruction represents a valid operation. The softmax function in Transformer calculates attention values between tokens that take a product of a query and key vectors, dividing it by the sum of its elements, $\mathrm{softmax}(\boldsymbol{x}_i) = \frac{\exp(\boldsymbol{x}_i)}{\sum_j^n \exp(\boldsymbol{x}_j)}$ where $\boldsymbol{x}$ is the input vector (*i.e.*, product of a query and key vectors in Transformer), and n is the size of the vector (*i.e.*, the number of tokens). Because of layer normalization and scaling by Softmax in Transformer (Figure 3), it becomes common to remain a few values to be large and the rests to be extremely small. This results

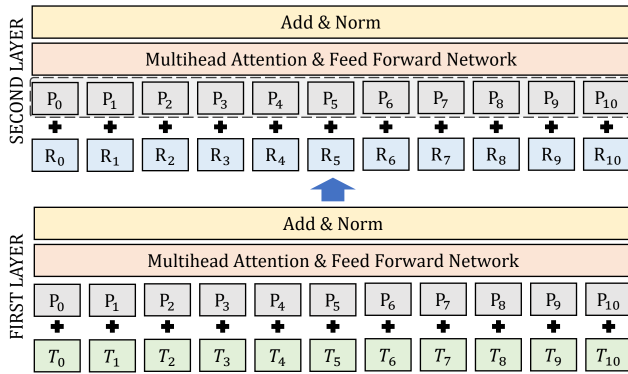


Figure 4: Illustration of a per-layer positional embedding in AsmDepictor. Unlike vanilla Transformer which takes positional embedding (P) at the first layer only, we provide an additional positional embedding (dotted area) to the upper layers. P, T, and R denote a position, token, and (internal) representation encoding, respectively.

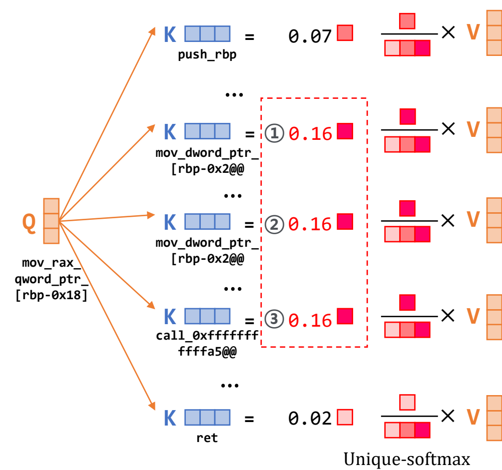


Figure 5: Illustration of Unique-softmax activation function. A query vector with each key vector in a sentence computes a final attention value. Two consecutively identical instructions (*e.g.*, ① and ②) or even different instructions (*e.g.*, ② and ③) produce the same attention outputs after round up as shown in a dotted square box.

in restricting the capability of understanding the relationships between tokens. Figure 8 in Appendix shows an attention heatmap of each transformer layer, which supports that utilizing the softmax function renders the attention values between tokens considerably fainted in the upper layers like Figure 8a, Figure 8b, and Figure 8c. To address the above dimming issue, we suggest the *Unique-softmax function*. It normalizes the input tokens by grouping similar tokens in the input vector, then dividing each by the sum of unique values (*e.g.*, Figure 8d): Unique-softmax$(\boldsymbol{x}_i) = \frac{\exp(\boldsymbol{x}_i)}{\sum_j^d \exp(\boldsymbol{u}_j)}$ where $\boldsymbol{u}$ is the vector that holds a unique value among $\boldsymbol{x}$, and d is the size of $\boldsymbol{u}$ vector. The denominator of Unique-softmax function is less than or equal to that of the softmax because it does not add the same value multiple times: $\sum_j^d \exp(\boldsymbol{u}_j) \leq \sum_j^n \exp(\boldsymbol{x}_j)$. Therefore, the output of Unique-softmax is higher than that of the softmax by following d value. To find a similar value in a vector, we round up values to the parameter r. Figure 5 shows an example of the Unique-softmax function (See algorithm 1 in Appendix).

Jaccard* Score. One of the widely adopted metrics to evaluate a language generation model is the BLEU score [58]. As shown in Table 1, directly applying BLEU to a short output (*e.g.*, an average token length is around three) is inappropriate because it gives an excessive *brevity penalty* within a short sequence of tokens. Considering such a concise output, one can simply catch the original meaning from an inferred function symbol without taking an order into account; *e.g.*, `make time` from `time make`. However, such an order-agnostic metric could falsely drive a model to deduce a long sequence of words that encloses the ground truth as a subset; *e.g.*, `make size info time build get` where its reference text is `make time`. To this end, we devise the order-free Jaccard* score that enables one to effectively evaluate a short sequence of words (*i.e.*, function symbol). In particular, we suggest the factor of a *negative brevity penalty* (*NBP*) to prevent generation toward a superset of the ground truth, which the penalty begins to be applied when the number of predicted words is longer than that of reference words. Equation 4.3 briefly shows the proposed Jaccard* score tailored for evaluating a short output generation task where H and R represent a set of inferred words and reference words, respectively.

$$\text{Jaccard}^* = NBP \times \frac{|R \cap H|}{|R|},\ NBP = \begin{cases} 1 & \text{if } |\text{H}| \leq |\text{R}| \\ \exp\left(1 - \frac{|H|}{|R|}\right) & \text{otherwise} \end{cases}$$

4.4 Function Symbol Inference

Once a model training is complete, AsmDepictor is ready for inferring a function symbol given a sequence of machine instructions (*e.g.*, function) as an input. It is noted that the input requires to be refined via both disassembly and BPE tokenization process. As in Figure 3, fetching an instruction embedding and its positional embedding from the model, an encoder calculates attention values within the sequence of an assembly (*i.e.*, inputs' relevance), passing it to a decoder. Finally, the decoder computes attention values using both the token embedding of a decoder and that of an encoder-decoder (applying no positional information at this moment), predicting a word with the highest probability with a softmax layer and uses a beam search [22] to output a predicted token. We repeat this process until the model predicts `[EOS]`.

5 IMPLEMENTATION

Binary Build and Analysis. We leverage the built-in IDAPython [31] from IDA Pro 7.6 [2] to analyze each binary, building a database that contains individual binary information for further refinement (*e.g.*, code normalization, de-duplication, tokenization). We utilize debugging information for extracting function boundaries (*i.e.*, function start and end) and symbol names as ground truth. The original NERO dataset [18] (DS_N) has been released with debugging sections being present. Meanwhile, we build additional binaries with debugging information available in preparation for the AsmDepictor dataset (DS_A), by leveraging `apt-build` [1] that

automatically retrieve sources and rebuild a list of common packages from popular Ubuntu Linux distributions. For the comparison of code refinement techniques (§6.1), we implemented our own script that follows a handful of rules on immediate values, registers and pointers, proposed by DeepSemantic [42], InnerEye [82], and PalmTree [48]. Note that we apply BPE tokenization on top of a raw assembly for building a vocabulary.

Model Implementation. With a skeleton of the original Transformer implementation [7] and PyTorch [59], we develop the framework AsmDepictor that supports both per-layer positional embedding and Unique-softmax techniques (§4.3). Unlike NLP that varying pre-trained models are prevalent, the model for binary codes is not commonly available. Then, we initialize our model with Xavier [27], taking around 179 hours (around seven and half days) to train the AsmDepictor model using $\mathbf{DS_A}$ (§6) with 40, 004, 102 parameters. As the size of a Transformer model grows exponentially according to the length of (sequential) input tokens, we set up a token length limit to 300; that is, we discard all tokens longer than the limit. We use a scheduled Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$ as proposed by the original Transformer. However, we reconfigure a warmup step and a multiplication factor to (24, 000, 0.8) for $\mathbf{DS_N}$ [18] and (18, 000, 1.0) for $\mathbf{DS_A}$, respectively, because they are affected by the batch size and the data length of a model. We set up the batch size of 10 and 90 for $\mathbf{DS_N}$ and $\mathbf{DS_A}$. We use a dropout rate of 0.1 after Attention and Feed-Forward. As a final note, we display primary hyperparameters that impact overall performance in our repository[1].

6 EVALUATION

This section evaluates AsmDepictor by answering four research questions about effectiveness and efficiency. We run experiments on a 64-bit Ubuntu 20.04 system equipped with Intel(R) Xeon(R) Gold 5218R CPU 2.10GHz, 256GB RAM, and two RTX A6000 GPUs. Note that we utilize $\mathbf{DS_N}$ for verifying the effectiveness of a AsmDepictor model, and a common Rouge-l metric for handy comparison.

Program Corpus. We harness two different datasets for evaluation. First, we utilize $\mathbf{DS_N}$ [18] that is a publicly available corpus, enabling AsmDepictor to make a direct performance comparison with other state-of-the-art baselines [17, 30]. We additionally build 2, 522 executable binaries (ELF for `x86_64` architecture) with debugging information available, including system utilities, networking tools, and varying libraries across four different Ubuntu Linux distribution versions (*i.e.*, 14.04, 16.04, 18.04 and 20.04). The common packages across different distributions incorporate different versions of a program, which helps efficient model training. Note that we exclude a function symbol that complies with a name mangling rule (*e.g.*, C++) because a compiler-generated function identifier may bring about degrading a model performance. We have $\mathbf{DS_A}$ include $\mathbf{DS_N}$ to see performance improvement on a large scale. As shown in Table 1, $\mathbf{DS_N}$ includes 84, 433 functions from 541 binaries whereas $\mathbf{DS_A}$ holds 520, 532 functions from 3, 063 binaries. We excluded 16, 767 (19.85%) and 116, 326 (22.35%) functions that have an identical body but a different symbol name as well as the ones that have the same body and name (*i.e.*, de-duplication as part of data refinement), acquiring 67, 666 and 404, 206 functions in each corpus (§4.2). Interestingly, the volume of vocabularies for $\mathbf{DS_N}$

Table 1: Statistics for $\mathrm{DS_N}$ and $\mathrm{DS_A}$. The number of vocabularies with BPE for $\mathrm{DS_A}$ is quite similar (slightly smaller) to that for $\mathrm{DS_N}$.

Statistic	$\mathrm{DS_N}$	$\mathrm{DS_A}$
Number of binaries	541	3,063
Number of function symbols	84,433	520,532
Identical function bodies	16,767	116,326
Identical function symbols	53,902	430,310
Final corpus (de-duplicated)	**67,666**	**404,206**
Number of a training set	54,134	323,364
Number of a test set	13,534	80,842
Token (Function) length on average	135.04	204.24
Token (Symbol) length on average	2.55	3.01
Number of vocabularies with BPE	**9,958**	**9,923**

(9, 958) is slightly larger than that for $\mathbf{DS_A}$ (9, 923) after applying BPE to assembly codes although the latter holds almost six times larger function symbols than the former.

Research Questions. We raise the following research questions for AsmDepictor evaluation from four aspects: ① data refinement strategy, ② AsmDepictor model approaches, ③ performance comparison with examples, and ④ efficiency of the models.

- **RQ1.** How much improvement does our suggested data refinement strategy contribute to performances (§6.1)?
- **RQ2.** How much enhancement does our proposed techniques contribute to performances (§6.2)?
- **RQ3.** Does AsmDepictor surpass other state-of-the-art baseline models for generating a function symbol (§6.3)?
- **RQ4.** How efficient is AsmDepictor in practice (§6.4)?

6.1 Effectiveness of Code Refinement (RQ1)

Binary code (as data) pre-processing is commonplace [42, 48, 55, 80, 82] before learning a model due to vocabulary issues, however, its effectiveness has hardly been evaluated yet. To the best of our knowledge, our work is the first study to make a systematic assessment for the validity of binary code refinement quantitatively.

Code Normalization. Normalizing an assembly code is an informal process of trimming seemingly-less-useful-information to mainly reduce the number of vocabularies before feeding it into a neural network. We conduct an experiment with the NERO [17] dataset to demonstrate the effectiveness of a few code normalization techniques, which prior work [42, 48, 55, 80, 82] has introduced. DeepSemantic [42] suggests the well-balanced code normalization that converts the representation of registers, pointers, and immediate values into a generalized form to reduce the number of vocabularies. As the baseline with the DeepSemantic rules, we adopt other approaches such as retaining a register from InnerEye [82] (*e.g.*, `reg8` → `rax`), and a two-byte immediate value proposed by PalmTree [48] (*e.g.*, `immval` → `0x36`), which can offer supplementary information for further learning. Table 8 summarizes the comparison between a combination of several code normalization techniques, empirically confirming that feeding additional information aids to enhance overall performance. Note that the case with raw assembly codes show the best performance (F1 of 57.67 or Jaccard* score of 60.67). However, the size of model parameters must be considered as the volume of dataset increases (Table 2).

Tokenization. We investigate various tokenization means in terms

Table 2: Comparison of the size of parameters and vocabularies depending on different tokenization methods. The volume of model parameters grows in accordance with that of vocabularies. In case of DS_A, the number of parameters exceeds 1 billion where the number of tokens is 3.25 million. BPE tokenization can appropriately adjust both the size of parameters (around 40M) and vocabularies (below 10K).

Tokenization Methods	DS_A		DS_N	
	Param Size	Vocab Size	Param Size	Vocab Size
Instruction	1,012,857,350	3,253,394	222,776,838	699,970
Instruction w/ D_U	628,541,958	2,359,047	162,619,398	563,036
Instruction w/ D_{US}	521,095,174	2,150,998	138,883,590	536,832
Instruction w/ BPE	**40,004,102**	**9,923**	**32,251,910**	**9,958**

of both assembly codes (input) and function symbols (output). We conduct another experiment with four groups for an instruction and three groups for a function symbol to choose the best tokenization strategy (See Table 10 for examples). For an assembly code, we ① set up a raw machine instruction as a baseline. Next, we ② separate an instruction with the delimiter of an underscore (D_U), and ③ take special characters (*e.g.*, `[,],+,-, *`) apart as well as an underscore (D_{US}). Finally, we ④ apply BPE [69] as suggested by Karampatsis et al. [37]. For a function symbol, we ① configure a full function identifier as a baseline. We ② apply a word separation with the delimiter of a snake case (*i.e.*, underscore) and a camel case (D_{UC}), and ③ BPE. Table 9 summarizes experimental results of $F1$ and Jaccard* score with DS_N. Interestingly, our empirical finding shows that a function symbol with BPE exacerbates an overall performance, and the combination of an intact instruction and a function symbol with D_{UC} ranks the first of all groups (*i.e.*, $F1$ of 57.67 and Jaccard* score of 60.67). Based on our experimental results, we choose a tokenization strategy of BPE with assembly codes and D_{UC} with function symbols for AsmDepictor that strikes a balance between model scalability and model performance, resulting in $F1$ of 57.14 and Jaccard* score of 60.26. As seen in Table 2, the size of model parameters generally grows in proportion to the number of vocabularies. The strategy of BPE tokenization for an instruction maintains a balance of both the volume of parameters (approximately 40M) and vocabularies (below 10K) even for DS_A.

6.2 Effectiveness of AsmDepictor (RQ2)

Layer Reduction. The original Transformer employs the architecture with a stack of six encoders and decoders. However, we encounter that building a model with the naïve Transformer had been consistently failed with previous code normalization techniques. Our investigation reveals that scaling (*e.g.*, softmax, layer normalization) causes vanishing attention values, extracting meaningful relationships between tokens (*i.e.*, instructions) unavailable. Figure 8a in Appendix illustrates that the uppermost layer indeed barely sees the relationship between instructions. In this respect, we reduce the number of layers (*e.g.*, three layers in AsmDepictor) at both encoding and decoding components, obtaining an additional F1 of 5.6 and Jaccard* score of 4.4 (Table 5).

Table 3: Performance comparison according to different strategies for holding positional information. An embedding scheme is way better than an encoding one.

Strategy	Precision	Recall	F1	Rouge-l	Jaccard*
Positional encoding (Transformer)	38.53	41.30	37.21	39.58	41.92
Positional embedding (BERT)	55.77	56.00	55.83	57.75	59.77
Per-Layer positional embedding	57.13	57.17	**57.14**	**58.68**	**60.26**

Table 4: Performance comparison across different activation functions. Unique-softmax achieves the best performance in all metrics.

Activation Function	Precision	Recall	F1	Rouge-l	Jaccard*
Sigmoid	53.32	54.22	53.60	56.30	57.91
Softmax	53.90	54.19	53.97	56.45	57.75
Unique-softmax	57.13	57.17	**57.14**	**58.68**	**60.26**

Table 5: Ablation results of AsmDepictor over our proposed approaches. We use Transformer with a BPE-encoded assembly as a baseline. Taking all techniques together enhances overall performance up to around 10 in all metrics.

Applied Techniques	Precision	Recall	F1	Rouge-l	Jaccard*
Transformer (T)	47.31	47.80	47.46	50.74	50.12
T + Layer Reduction (LR)	53.22	53.79	53.10	53.31	54.48
T + LR + Positional Embedding (PE)	53.90	54.19	53.97	56.45	57.75
T + LR + PE + Unique-softmax	57.13	57.17	**57.14**	**58.68**	**60.26**

Per-Layer Positional Embedding. We introduce a per-layer positional embedding (§4.3) to offer high-quality position information. Recall that our scheme slightly differs from a positional embedding in BERT in that BERT combines a positional embedding with a token embedding merely for the first layer whereas AsmDepictor does for all layers. We conduct an experiment to confirm that our approach assists an overall performance. Table 3 shows the effectiveness of a per-layer positioning embedding alone in comparison with a positional encoding from Transformer and a positional embedding from BERT. Notably, the AsmDepictor strategy for position information outperforms other approaches in both F1 (*i.e.*, increase of 18.62 and 1.31) and Jaccard* score (*i.e.*, increase of 18.34 and 0.49).

Unique-softmax Function. As stated in §4.3, we devise the Unique-softmax function to reduce the side effects of layer normalization and scaling by the softmax in Transformer. For comparison, we utilize different activation functions including sigmoid and the softmax. Although using a sigmoid function can be a straightforward alternative by allowing each word (each scalar in a key vector) to map into the range of $(0, 1)$, it cannot represent the weighted sum of a word as softmax. Table 4 shows the effectiveness of Unique-softmax in comparison with sigmoid and softmax. The Unique-softmax exceeds other activation functions with F1 by an increase of 3.54 and 3.17, and with Jaccard* by an increase of 2.35 and 2.51.

Ablation Study. We evaluate AsmDepictor with ablation on three techniques to enhance its performance to demonstrate the effectiveness of each technique. We adopt our techniques atop the naïve Transformer, such as a layer reduction, per-layer position embedding, and Unique-softmax. Note that we feed an instruction with BPE to Transformer with three layers. The ablation study (Table 5)

Table 6: Performance comparison with state-of-the-art baseline models. AsmDepictor outperforms Debin [30] and NERO [17] with a wide margin (*i.e.*, 4x better performance). The AsmDepictor model with DS_A achieves up to 71.5 of F1 and 75.4 of Jaccard* score.

Model	Precision	Recall	F1	Rouge-l	Jaccard*
Debin	5.73	5.66	5.66	5.87	5.94
NERO	12.35	12.36	12.35	14.07	14.99
Transformer	47.31	47.80	47.46	50.74	50.12
AsmDepictor (DS_N)	57.13	57.17	57.14	58.68	60.26
AsmDepictor (DS_A)	71.52	71.53	71.52	73.75	75.42

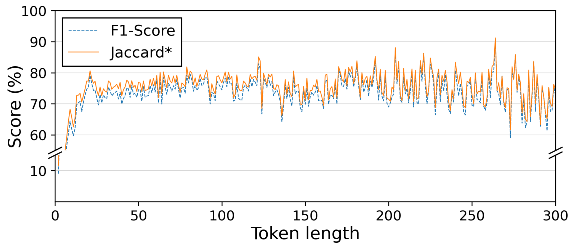


Figure 6: F1 and Jaccard* scores by a token length per function. Empirical results show a decent performance (up to around 80) with the number of tokens ranging from 20 to 200, which accounts for 64.1% of the whole functions.

confirms that performance has been gradually improved by adding each technique. Adding all strategies together results in considerable performance advances by 9.68 in F1 and 10.14 in Jaccard* score, compared to the baseline Transformer. Note that we observe that it is indeed rare to see redundant tokens in a function body with BPE processing. Rather, it is common to see redundant values of Attention, which advances overall performance.

6.3 AsmDepictor Performance (RQ3)

Baseline Models. Debin [30] is a non-neural network model that instead utilizes a conditional random field [46] to predict both function and variable names. Although the main design of Debin focuses on a variable name generation task (rather than a function name), we brought its function name prediction scores for comparison. Lately, NERO [17] targets a function name generation task via a graph neural network (GNN) [41] that is the closest in spirit to our work. The approach of NERO leverages enriched representations with a set of call-site sequences to predicting a subroutine name, constraining its application solely to a function in the presence of a call site. Besides, we include experiments using a naïve Transformer [75] (*i.e.*, six layers). It is noteworthy mentioning that we follow Debin and NERO's pre-processing for their evaluations, where we use data refining with BPE for other models. We employ DS_N for training all models but AsmDepictor with DS_A. Additionally, we utilize the Rouge-l metric for further comparison.

Results. Table 6 shows the performance comparison of Debin, NERO, Transformer, and AsmDepictor with the two different corpora. Our model trained with DS_N *surpasses* 400% *or more performance* than previous the-state-of-the-art models ($F1$ of 57.1 or Jaccard* score of 60.3). The AsmDepictor model with DS_A achieves up to $F1$ of 71.5 or Jaccard* score of 75.4, indicating that three out of four (learned) function symbols have been precisely inferred. Moreover, we analyze performance variation by the token length of a function. Figure 6 depicts that a small number of tokens (*e.g.*, < 20) exhibits a foreseeable low performance mainly because, as an extreme instance, a function with a single token (*e.g.*, `ret`) does not convey much information to deduce a function symbol. Meanwhile, the performance of a longer token length (*e.g.*, > 200) has a relatively higher variance. Note that AsmDepictor records a fairly reasonable Jaccard* score for the token length between 20 and 200, which two thirds of the entire functions belong to. Moreover, we conduct an additional experiment with a large number of tokens (*e.g.*, > 300) per function body, resulting in a comparable score (*e.g.*, F1= 72.85, Jaccard*= 76.17) after truncating them.

Table 7: Comparison of training time and memory size per epoch depending on different instruction tokenization methods and positional embedding strategies.

Method (Strategy)	Training Time	Memory Size
Instruction	14m 44s	18,529 MB
Instruction w/ D_U	11m 2s	12,653 MB
Instruction w/ D_{US}	10m 9s	10,991 MB
Instruction w/ BPE	**6m 41s**	**5,961 MB**
Positional encoding (Transformer)	6m 40s	5,967 MB
Positional embedding (BERT)	6m 38s	5,957 MB
Per-Layer positional embedding	**6m 41s**	**5,961 MB**

6.4 Efficiency of AsmDepictor (RQ4)

In this section, we exhibit the efficiency of AsmDepictor in terms of practicality. Table 7 summarizes training time and memory size per epoch (with DS_N) across different instruction tokenization methods and positional embedding strategies. Our tokenization means with BPE shows a significant reduction on a computational cost because it helps to decrease the total number of vocabularies and their embedding size. Meanwhile, different positional embedding strategies show marginal gaps. Note that the whole training for our models took 16 hours 42 minutes (150 epochs) with DS_N, and 7 days 11 hours 46 minutes (200 epochs) with DS_A.

6.5 Discussion and Limitation

Output Word Limitation. One of evident limitations is that AsmDepictor solely allows for generating a sequence that consists of known vocabularies. In other words, the performance of the AsmDepictor decoder relies on a learned set of output tokens. Thus, an attempt to infer a function symbol that contains an unknown word during training may severely impact an overall performance, being unable to assist a reversing task. To mitigate the current restriction, we envision two directions as part of our future work. First, the limitation can be relaxed by training with a rich description like documentation comments that annotate source code for other accessible formats (*e.g.*, HTML, PDF). Second, recent advancement

with the GPT architecture [64] allows the AsmDepictor encoder to be able to blend with a GPT-2 pre-trained model, efficiently tackling an OOV problem for an output.

Unique-softmax Computation. Unlike the softmax activation, the Unique-softmax computation requires additional resources for sorting values ($O(n \log n)$) within a matrix to remove duplicates. As a result, training AsmDepictor indeed takes 33% more time than Transformer. However, one may prefer a trade-off to save training time with a naïve Transformer decoder because our ablation study (Table 5) shows limited performance degradation due to the rare occurrence of a redundant token. Note that the training is a one-time process.

Unbalanced Token Frequency. As illustrated in Figure 7, the token frequency of instructions with BPE entails a long tail, which means that considerable amount of tokens have been rarely appeared. Thus, it is possible that AsmDepictor could produce a relatively poor result with a function that accommodates many tokens with a low frequency.

Function Inlining and Outlining. As one of compiler optimization techniques, inlining a function is a common process by embedding the function into another to reduce runtime overheads. On the contrary, a compiler may carry out a function outlining task that defines an identical sequence of instructions as a separate function. In this work, we do not consider such optimizations that could distort the original function symbols.

Function Name Mangling. Modern compilers take advantage of name mangling (*i.e.*, name decoration) to resolve a conflict when having the same identifier across different namespaces or having different function signatures (*e.g.*, function overloading). The current implementation of AsmDepictor excludes a function name using such a name mangling scheme because it often implies a class hierarchy, data type or structure that does not directly depict the behavior of a function. It is noted that there is not a standard name mangling rule, resulting in different mangled identifiers across different compilers, compiler versions, or architectures.

Other Applications. We believe that our work can be expanded to other useful applications For instance, the AsmDepictor structure can learn a malware behavior description (or comments from previous analysis) corresponding to a sequence of machine codes, providing a quick overview when a suspicious code is given. Learning function symbols from ossified code fragments (*e.g.*, driver, kernel, network stack, firmware) across different versions can be another good application of AsmDepictor for finding a known vulnerability within a function. It is worth noting that it is possible to build an instruction-set-specific model per architecture because the way of learning an assembly language in AsmDepictor is architecture-agnostic.

7 RELATED WORK

Binary Code Representation for DNN. A wide spectrum of previous approaches [21, 42, 48, 55, 60, 70, 80, 82] pertaining to binary code representation have been introduced for deep learning. Early work [70] takes a simple approach by feeding each byte to a deep neural network for recognizing a function boundary in a binary. InnerEye [82] introduces a simple pre-processing rule (*e.g.*, converting an immediate to 0) to avoid OOV for assembly codes. Asm2vec [21] proposes an assembly code representation learning based on PV-DM (Distributed Memory version of Paragraph Vector) model [47]. DeepBinDiff [80] generates an embedding with an opcode and operands weighted with a TF-IDF (Term Frequency-Inverse Document Frequency) model [65]. DeepSemantic [42] introduces a well-balanced code normalization with a fine-grained code transformation considering a register size, pointer conversion, and varying cases in an operand (*e.g.*, call target, reference). PalmTree [48] adopts BERT [20] with three tasks (MLM; Masked Language Model, CWP; Context Window Prediction, Def-Use Prediction; DUP) for capturing complex internal formats of each instruction. Lately, Karampatsis et al. [37] suggest an advanced approach with BPE [69] in the software engineering literature for handling the following vocabulary issues: a large corpus of vocabulary, an unseen vocabulary (*i.e.*, OOV), and a rarely appeared vocabulary. We adopt BPE for AsmDepictor because a set of machine instructions inevitably entails the above problems.

ML-assisted Binary Reversing. A compilation process irrevocably eliminates useful information for understanding code semantics in a stripped binary, reverse engineers struggle to deduce its original context. A plethora of prior work leverage deep learning techniques to facilitating binary analysis [28, 54], decompilation [24, 28, 38, 39], and essential information recovery [14, 17, 30, 45, 52] including a type [14, 52], a variable name [14, 30, 45] and a function name [17, 30]. Dire [45] proposes a variable prediction model based on LSTM (Long Short-Term Memory) [32] and GNN [41] with an AST. Debin [30] first introduces a prediction system for debug information (*e.g.*, function and variable symbol) that utilizes an Extremely randomized Tree classifier and a linear probabilistic graphical model. In a similar vein, NERO [17] leverages augmented representations of call sites with GNN to generate a function name. We compare AsmDepictor with Debin [30] and NERO [17] as baseline models. DIRTY [14] presents a Transformer-based model that recommends a variable type and name for producing high-quality decompilation output. One of the latest work, SymLM [36], is probably the closest in spirit to our work, which proposes a neural architecture that learns context-sensitive function semantics by jointly modeling the execution behavior of a calling context and function instructions.

As a final note, AsmDepictor requires both an encoder and a decoder for a description generation task (*e.g.*, function symbol name) while a discriminative model generally merely needs an encoder for a label classification task (*e.g.*, function similarity).

8 CONCLUSION

Even equipped with varying automation tools and techniques, binary reversing still requires expertise knowledge with tedious manual efforts. We present AsmDepictor, the Transformer-based inference framework for efficiently predicting a function symbol name from an assembly language, which aims to give a quick glimpse over a stripped binary for further binary analysis in practice. The AsmDepictor framework consists of three main components: data refinement (function de-duplication, BPE tokenization), model training (per-layer positional embedding, Unique-softmax), and a function name inference. Our empirical evaluation shows that

AsmDepictor outperforms other state-of-the-art models, achieving both F1 and Jaccard* scores up to four times.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00421; AI Graduate School Support Program (Sungkyunkwan University), No. 2022-0-01199; Graduate School of Convergence Security (Sungkyunkwan university), No. 2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes), and the Basic Science Research Program through NRF grant funded by the Ministry of Education of the Government of South Korea (No. NRF-2022R1F1A1074373). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] 2022. General Commands Manual - apt-build. https://manpages.debian.org/testing/apt-build/apt-build.1.en.html.

[2] 2022. IDA Pro 7.6. https://hex-rays.com/ida-pro. Accessed: 2022-04-05.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Association for Computational Linguistics (ACL)*. Online.

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. Online.

[5] Jim Alves-Foss and Jia Sone. 2019. Function Boundary Detection in Stripped Binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroSP)*. Paris, France.

[7] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the 2015 International Conference on Learning Representations (ICLR)*. San Diego, CA, USA.

[8] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[9] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-flow Graph Matching. In *Proceedings of the 3rd Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Berlin, Germany.

[10] Business Software Alliance. 2018. Software Management: Security Imperative, Business Opportunity. Global Software Survey (2018), 24.

[11] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 11, 4 (2013), 307–317.

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Tan Hee Beng Kuan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Seattle, WA.

[13] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Tan Hee Beng Kuan. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[14] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *Proceedings of the 31th USENIX Security Symposium (Security)*. Boston, MA.

[15] YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning Sequential and Structural Information for Source Code Summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Online.

[16] Yoon-Ho Choi, Peng Liu, Zitong Shang, Haizhou Wang, Zhilong Wang, Lan Zhang, and Junwei Zhou. 2020. Using Deep Learning to Solve Computer Security Challenges: a Survey. *Cybersecurity* (2020).

[17] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. In *Proceedings of the 31th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. New York, NY.

[18] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs Dataset. https://zenodo.org/record/4099685.

[19] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. Minneapolis, MN.

[21] Steven H. H. Dinga, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[22] Jacob Eisenstein. 2019. *Introduction to natural language processing*. MIT press.

[23] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[24] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An End-to-End Neural Program Decompiler. In *Proceedings of the 33rd Neural Information Processing Systems (NeurIPS)*. Vancouver, Canada.

[25] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. A Lightweight Framework for Function Name Reassignment Based on Large-Scale Stripped Binaries. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA.

[26] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *Comput. Surveys* (2017).

[27] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th Journal of Machine Learning Research (JMLR)*. Sardinia, Italy.

[28] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[29] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–38.

[30] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.

[31] Hex-rays. 2022. IDAPython Documentation. https://www.hex-rays.com/products/ida/support/idapython_docs/.

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[33] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 22nd USENIX Security Symposium (Security)*. Washington, DC.

[34] Kazuki Irie, Albert Zeyer, Ralf Schlüter, and Hermann Ney. 2019. Language Modeling with Deep Transformers. In *Proceedings of the 20th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. Graz, Austria.

[35] Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. 2019. What Does BERT Learn about the Structure of Language?. In *Proceedings of the 57th Association for Computational Linguistics (ACL)*. Florence, Italy.

[36] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2021. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.

[37] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the 42th International Conference on Software Engineering (ICSE)*. Seoul, South Korea.

[38] Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy.

[39] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards neural decompilation. *arXiv preprint arXiv:1905.08325* (2019).

[40] TaeGuen Kim, Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2019. Binary Executable File Similarity Calculation using Function Matching. *The Journal of Supercomputing* 75, 2 (2019), 607–622.

[41] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 2017 International Conference on Learning Representations (ICLR)*. Palais des Congrès Neptune, Toulon, France.

[42] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. 2021. Semantic-aware Binary Code Representation with BERT. *arXiv preprint arXiv:2106.05478* (2021).

[43] Hyungjoon Koo, Soyeon Park, and Taesoo Kim. 2021. A Look Back on a Function Identification Problem. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[44] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Seattle, WA.

[45] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2018. DIRE: A Neural Approach to Decompiled Identifier Renaming. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[46] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the 2001 International Conference on Machine Learning (ICML)*. San Francisco, CA, USA.

[47] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (PMLR)*. Bejing, China.

[48] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Virtual.

[49] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Proceedings of the 42nd Association for Computational Linguistics (ACL)*. Barcelona, Spain.

[50] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.

[51] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Lisbon, Portugal.

[52] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Göteborg, Sweden.

[53] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. RE-Mind: a First Look Inside the Mind of a Reverse Engineer. In *Proceedings of the 31th USENIX Security Symposium (Security)*. Boston, MA.

[54] Luca Massarelli, Giuseppe A. Di Luna, , Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*. San Diego, CA.

[55] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Göteborg, Sweden.

[56] National Security Agency (NSA). 2019. Software Reverse Engineering (SRE) Suite of Tools. https://ghidra-sre.org/.

[57] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. 2020. o-glassesX: Compiler Provenance Recovery with Attention Mechanism from a Short Code Fragment. In *Proceedings of the 3rd Workshop on Binary Analysis Research (BAR)*. San Diego, CA.

[58] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2004. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Association for Computational Linguistics (ACL)*. Philadelphia, Pennsylvania, USA.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd Neural Information Processing Systems (NeurIPS)*. Vancouver, Canada.

[60] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv preprint arXiv:2012.08680* (2020).

[61] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[62] Steven T Piantadosi. 2014. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic bulletin & review* 21, 5 (2014), 1112–1130.

[63] Radare2. 2019. Libre and Portable Reverse Engineering Framework. https://rada.re/n/.

[64] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[65] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Citeseer, 29–48.

[66] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, Canada.

[67] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. 2021. Linear transformers are secretly fast weight programmers. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*. Baltimore, MD, USA.

[68] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Proceedings of the 1st Workshop on Binary Analysis Research (BAR)*. San Diego, CA.

[69] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2019. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Association for Computational Linguistics (ACL)*. Berlin, Germany.

[70] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

[71] Paria Shirani, Leo Collard, Basile L. Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. BinArm: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Device. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Paris, France.

[72] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[73] Eliza Strickland. 2022. Andrew Ng, AI Minimalist: The Machine-Learning Pioneer Says Small is the New Big. *IEEE Spectrum* 59, 4 (2022), 22–50. https://doi.org/10.1109/MSPEC.2022.9754503

[74] Brian Testa, Heng Yin, Yao Cheng, Qian Feng, Rundong Zhou, and Chengcheng Xu. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the 31st Neural Information Processing Systems (NeurIPS)*. Long Beach, CA.

[76] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME*. Shanghai, China.

[77] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Proceedings of the 33rd Neural Information Processing Systems (NeurIPS)*. Vancouver, Canada.

[78] Wikipedia. 2022. Naming convention (programming). https://en.wikipedia.org/wiki/Naming_convention_(programming).

[79] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. 2019. Machine Learning-Based Analysis of Program Binaries: A Comprehensive Study. *IEEE Access* (2019).

[80] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[81] G.K. Zipf. 1950. Human behaviour and the principles of least effort. *The Economic Journal* (1950).

[82] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, and Zhexin Zeng, Qiang andZhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison Beyond Function Pairs. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

A APPENDIX

Algorithm 1: Unique-softmax Algorithm Pseudocode

Input: $\boldsymbol{x} \in \mathbb{R}^{B\times H\times Q\times K}$ where B: Batch size, H: number of heads, Q: Query size, and K: Key size
Round parameter r

Output: $\frac{\exp(\boldsymbol{x})}{\sum_i exp(\boldsymbol{u}_i)} \in \mathbb{R}^{B\times H\times Q\times K}$ where $\boldsymbol{u}$ is unique values vector among $\boldsymbol{x}$ rounded up by r

1. $maxes \leftarrow torch.max(x, dim = -1, keepdim = True)[0]$
2. $xExp \leftarrow torch.exp(x - maxes)$
3. $xExp \leftarrow torch.round(xExp \times torch.pow(10, r))/(torch.pow(10, r))$
4. $uVec, _ \leftarrow xExp.sort(dim = -1)$
5. $uVec[:, :, :, 1:] \leftarrow uVec[:, :, :, 1:] \times ((uVec[:, :, :, 1:] - uVec[:, :, :, :-1])! = 0).long()$
6. $xExpSum \leftarrow torch.sum(uVec, dim = -1, keepdim = True)$
7. **return** $\frac{xExp}{xExpSum}$

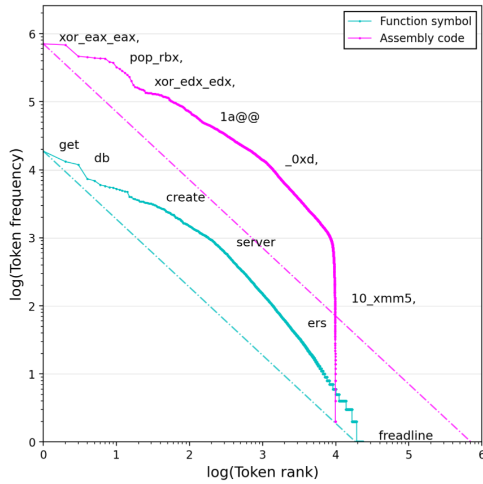


Figure 7: The log scale plots of the frequency and rank for input and output vocabularies approximately follows Quasi-Zipfian distributions [81] (slightly concave curves). The token frequency of an assembly rapidly drops from a certain point due to its rare appearance in our dataset.

A.1 Case Study

Table 11 shows comparison results of selected examples that four different models yield the prediction of each procedure symbol. NERO [17] and Debin [30] are state-of-the-art baselines, where we utilize the two AsmDepictor models with $\mathbf{DS_N}$ and $\mathbf{DS_A}$ corpus. The AsmDepictor model ($\mathbf{DS_A}$) is not only superior to others, but it is also practical by presenting a series of candidates from a function de-duplication process. For example, `xcalloc` is relatively a small routine with 19 tokens, showing four other possible function symbols despite the accurate inference of AsmDepictor: `m_calloc`, `event_alloc`, `bof_object`, and `state_new`. We discover that oftentimes a routine with an identical assembly (as a function body) holds similar identifiers. Another good example would be `is_alpha_mbchar` that AsmDepictor ($\mathbf{DS_A}$) made imprecise prediction as `as_is_alnum_mbchar`, however, interestingly one of its candidate contains the true symbol.

Table 8: Empirical results of inferring a function symbol with different code normalization means, which, we decide to stay assembly codes intact.

Code Normalization	Precision	Recall	F1	Rouge-l	Jaccard*
Well-balanced (WB)	14.72	14.73	14.72	17.45	20.06
WB + Registers (R)	15.84	15.81	15.82	19.03	21.29
WB + Immediates (I)	17.02	17.01	17.01	19.90	22.09
WB + R+ I	19.67	19.64	19.64	21.91	24.31
Assembly code	57.25	57.15	**57.67**	**58.81**	**60.67**

Table 9: Empirical results with a combination of input and output tokenization means. D_X represents a separation by the delimiter(s) of X where U, S, and C denote an underscore, special characters (*e.g.*, `[,],+,-, *,:`), and a camel case. We choose BPE and D_{UC} for an assembly code and function symbol tokenization that shows the best performance.

Input (Assembly)	Output (Function)	Precision	Recall	F1	Jaccard*
Instruction	Function	46.78	41.16	42.11	41.16
Instruction w/ D_U	Function	41.84	41.19	41.10	41.20
Instruction w/ D_{US}	Function	39.24	39.17	39.10	39.18
Instruction w/ BPE	Function	40.62	40.65	40.62	40.65
Instruction	Function w/ BPE	21.01	22.00	21.23	20.72
Instruction w/ D_U	Function w/ BPE	23.93	24.87	24.18	23.55
Instruction w/ D_{US}	Function w/ BPE	22.08	22.62	22.21	21.78
Instruction w/ BPE	Function w/ BPE	28.91	30.37	29.31	29.12
Instruction	Function w/ D_{UC}	57.25	57.15	57.67	60.67
Instruction w/ D_U	Function w/ D_{UC}	56.86	56.83	56.81	59.85
Instruction w/ D_{US}	Function w/ D_{UC}	56.36	56.42	56.37	59.31
Instruction w/ BPE*	**Function w/ D_{UC}***	57.13	57.17	**57.14**	**60.26**

A.2 Additional Experiments for Code Normalization

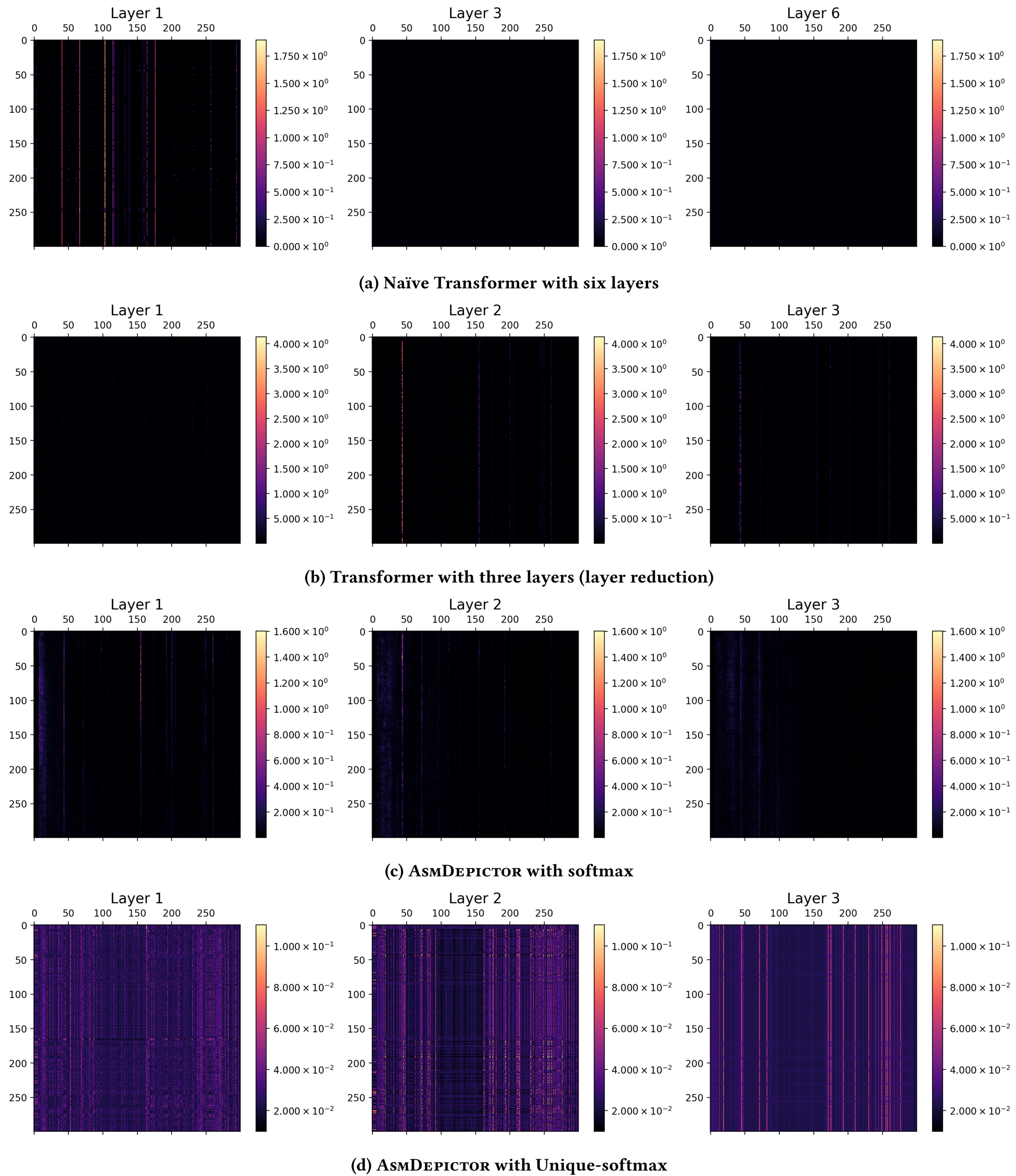
We quantitatively measure the effectiveness of code normalization by adopting several code normalization strategies proposed by InnerEye [82], DeepSemantic [42], and PalmTree [48]. InnerEye [82] first introduces a handful of simple rules to avoid OOV for code representation by replacing an immediate value, string, function name, and label (*e.g.*, target adderess) to `0, <str>, <FOO>`, and `<tag>`, respectively. Meanwhile, DeepBinDiff [80] splits an instruction into an opcode and operands where DeepSemantic [42] suggests a finer-grained rules (*e.g.*, well-balanced normalization) for an operand (*e.g.*, immediate, register, pointer). Similarly, PalmTree [48] proposes to remain two-byte immediate constants to offer rich information. Table 10 concisely shows the examples of varying code normalization and tokenization techniques. Not surprisingly, our empirical experiment (Table 8) indicates the more information offers better representation, however, it unavoidably incurs enormous computation resource (Table 2) as the volume of dataset increases. On the other hand, we carry out another experiment of the effectiveness of a tokenization (Table 9).

Table 10: Example of varying code normalization and tokenization techniques from previous approaches [42, 48, 80].

	Representation	Example1	Example2	Note
Assembly	Original instruction	`sub rsp, 0x50`	`mov qword ptr [rbp-0x58]`	Disassembled by IDA [2]
	Instruction as a single word	`sub_rsp_0x50`	`mov_qword_ptr_[rbp-0x58]`	Combined with an underbar
Normalization	Well-balanced (WB)	`sub_sp8_immval`	`mov_qword_ptr_[reg8-disp]`	Applying DeepSemantic rules [42]
	WB + Registers (R)	`sub_rsp_immval`	`mov_qword_ptr_[rbp-disp]`	Adding register names [80]
	WB + Immediates (I)	`sub_sp8_0x50`	`mov_qword_ptr_[reg8-disp]`	Adding 2-byte immediate values [48]
	WB + R + I	`sub_rsp_0x50`	`mov_qword_ptr_[rbp-disp]`	Adding both registers and immediates [48, 80]
Tokenization	Delimited with _ (DU)	`sub, rsp, 0x50`	`mov, qword, ptr, [rbp-0x58]`	Splitted by an underbar
	DU + Letter separation	`sub, rsp, 0x50`	`mov, qword, ptr, [, rbp, -, 0x58, ]`	Separating special letters including [, +, -,], :
	Byte-pair encoding (BPE)	`sub_rsp_0x5@@, 0`	`mov_qword_ptr_[rbp+0x@@, 58]`	Applying BPE

Table 11: Comparison of 20 selected function symbols in an alphabetical order across state-of-the-art baseline models [17, 30]. ● represents a precisely generated function symbol whereas ✗ means an empty output. The bold letters mean a partially accurate word that describes the original function regardless of its order. AsmDepictor is capable of producing possible function symbol candidates (*i.e.*, a parenthesis below) collected from a function de-duplication process.

Debugging Symbol	# Tokens	AsmDepictor (DS_A)	AsmDepictor (DS_N)	NERO	Debin
cb_build_perform_varying	292	**cb_build**_register	**cb_build**_program_forever	ecc_mul	arm_print_vma_and_name
cb_build_program_id	108	●	**cb_build_program**	options_menu	cint_remove
check_relaxed_syntax	227	**check**_type_name	**check**_backup_file	get_a_display	make_3way_diff
close_stdout	40	●	●	●	●
config_new	14	● new_kbnode, options_create	stdin_free	edit_new	conffile_bool
dbg_copy_some_packets	96	**dbg_copy**_all_**packets**	get_reloc_name	uuconf_process_time	api_get_file
default_homedir	66	●	set_**default_homedir**	dc_dc_quoted_free	mail_move_event
drop_privs	21	●	set_address	✗	lock_terminal
get_seckey_byname	72	●	v2i_general_name	quotearg_custom	display_mips_gnu_attribute
init_keywords	63	●	●	menu_**init**	hints_setup
is_alpha_mbchar	57	**is**_alnum_**mbchar** (is_blank_mbchar, is_alpha_mbchar, is_punct_mbchar)	**is**_selected	**is**_hashed	gettok
mb_copy	38	●	●	●	●
mbuiter_multi_next	181	●	●	●	●
parse_datetime	47	subst_string	●	**parse**	●
proc_encryption_packets	36	**proc**_signature_**packets**	●	dfacomp	print_reductions
quotearg_n_style_mem	31	●	●	●	●
set_program_name	91	●	●	●	●
write_file	215	●	●	process	to_rgip
xcalloc	19	● (m_calloc, event_alloc, bof_object, state_new)	●	●	alloc_common
yyensure_buffer_stack	114	●	grecs_json_ensure **buffer_stack**	gram_ensure **buffer_stack**	grecs_meta1_ensure **buffer_stack**



(a) Naïve Transformer with six layers

(b) Transformer with three layers (layer reduction)

(c) AsmDepictor with softmax

(d) AsmDepictor with Unique-softmax

Figure 8: Comparison of four Attention heatmaps with different configurations. The visualizations clearly show that the relationships (attention values) between tokens have been faded away when entering the upper layers (*e.g.*, Layer 1 VS Layer 3) in case of 8a, 8b and 8c. On the other hand, the built-in Unique-softmax for AsmDepictor effectively maintains the values even at the uppermost layer (8d).