A Lightweight Framework for Function Name Reassignment Based on Large-Scale Stripped Binaries

Han Gao
CAS Key Laboratory of
Electro-magnetic Space Information
University of Science and Technology of China
Hefei, China
gh2018@mail.ustc.edu.cn

Shaoyin Cheng*
CAS Key Laboratory of
Electro-magnetic Space Information
University of Science and Technology of China
Hefei, China
sycheng@ustc.edu.cn

Yinxing Xue
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
yxxue@ustc.edu.cn

Weiming Zhang*
CAS Key Laboratory of
Electro-magnetic Space Information
University of Science and Technology of China
Hefei, China
zhangwm@ustc.edu.cn

ABSTRACT

Software in the wild is usually released as stripped binaries that contain no debug information (e.g., function names). This paper studies the issue of reassigning descriptive names for functions to help facilitate reverse engineering. Since the essence of this issue is a data-driven prediction task, persuasive research should be based on sufficiently large-scale and diverse data. However, prior studies can only be based on small-scale datasets because their techniques suffer from heavyweight binary analysis, making them powerless in the face of big-size and large-scale binaries.

This paper presents the Neural Function Rename Engine (NFRE), a lightweight framework for function name reassignment that utilizes both sequential and structural information of assembly code. NFRE uses fine-grained and easily acquired features to model assembly code, making it more effective and efficient than existing techniques. In addition, we construct a large-scale dataset and present two data-preprocessing approaches to help improve its usability. Benefiting from the lightweight design, NFRE can be efficiently trained on the large-scale dataset, thereby having better generalization capability for unknown functions. The comparative experiments show that NFRE outperforms two existing techniques by a relative improvement of 32% and 16%, respectively, while the time cost for binary analysis is much less.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Social and professional topics → Software reverse engineering**.

*Shaoyin Cheng and Weiming Zhang are the corresponding authors.

KEYWORDS

Binary Analysis, Reverse Engineering, Neural Networks

ACM Reference Format:
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 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464804

1 INTRODUCTION

Most commercial off-the-shelf (COTS) software is closed-source. Additionally, the software is usually released as stripped binaries that contain no debug information for the purpose of easy distribution or copyright protection. Practitioners who want to analyze these programs should conduct reverse engineering and check the logic at the binary level. The disassemblers such as IDA Pro [36] can translate machine (binary) code into assembly language. However, assembly representation exists as plain instruction mnemonics with limited high-level information, making it hard to read and understand. Even an experienced reverse engineer may have to spend much time in determining the functionality of an assembly code snippet [22, 67]. To mitigate this problem, researchers from academia and industry have been studying *decompilation*, which is a process of lifting assembly into C-like pseudo-code for better readability [7, 17, 35, 42, 55, 63, 70]. The state-of-the-art decompilers on the market, such as Hex-Rays Decompiler [35], JEB [63], and RetDec [7], can reconstruct variables, types, functions and source-level structures from assembly code, and then output much higher level text which is more concise and much easier to read.

Although the decompilers can convert assembly-level idioms into high-level abstractions, they are weak at recovering high-level semantic information (e.g., variable names, and function names in stripped binaries). Especially, function names play an important role in program comprehension [30]. In practice, practitioners preliminarily guess the functionality of a function from its name [67]. Unfortunately, existing decompilers can only set address-related placeholders (e.g., `sub_40B8F0`) as default function names in the absence of debug symbols. Such non-descriptive names are not informative.

To help construct better decompilers and facilitate reverse engineering, we focus on function name prediction, aiming to reassign descriptive names for functions in stripped binaries. The current issue can be stated informally as the need to learn the correspondence between an assembly code snippet and a set of tokens that forms the function name. Although similar issues have been extensively studied at the source level [1, 3, 5, 27, 51], it is still hard because of the limited information in assembly code (discussed in Section 2).

In essence, the current issue is a data-driven prediction task so that persuasive research should be based on large-scale and diverse data. DEBIN [34] and Nero [22] are the only two existing techniques, but their datasets are far from large-scale. As shown in Table 1, the datasets used by DEBIN and Nero merely consist of 3,000 and 541 binaries, respectively. Nero is trained on about 60,000 functions. By contrast, the datasets used for source-level function name prediction consist of millions of functions [4, 5, 51, 52]. As for the assembly level, *a small and simple dataset contains limited instruction and tokens, making it unable to reflect the complexity of the current issue. A model trained from such a trivial dataset has a limited effect on real-world usages.* According to our reproduction and the comments on DEBIN in [49], it is the efficiency problem that prevents them from being trained and evaluated on large-scale binaries: Both DEBIN and Nero suffer from heavyweight binary analysis (e.g., binary lifting, data-flow analysis), which is time- and resource-consuming, making them powerless in the face of big-size and large-scale binaries.

In this paper, we present NFRE (Neural Function Rename Engine), a lightweight framework for function name reassignment that utilizes the instruction sequences and control-flow information of assembly code. NFRE uses fine-grained and easily acquired features, which make it much more effective and efficient.

Additionally, NFRE is trained and evaluated on a large-scale and well-built dataset. The binaries are collected from Ubuntu [15] without manual compilation, so the dataset can be large enough and of rich variety. However, two primary problems affect the usability of the dataset, which are *label noise* and *label sparsity*. The former refers to the existence of functions with meaningless names in the dataset. Non-descriptive names are useless to help understand the logic of functions. As the noise samples, they can bias the training process and hurt the model performance [41, 72]. The latter is caused by the overlarge vocabulary of tokens used in function names, which exacerbates data sparsity. The sparsity problem makes it difficult for learning-based techniques to learn the correspondence between assembly code and tokens. Prior studies [22, 34] neglect the two problems, while we present two data-preprocessing approaches to help mitigate them, thereby improving the usability of the dataset. We release the code and some instructions for reproduction at [33] to facilitate subsequent research. In summary, the contributions of this paper can be summarized as follows:

- We present NFRE, a lightweight framework for the reassignment of function names in stripped binaries. It does not require heavyweight binary analysis, so it can be efficiently trained and evaluated on large-scale binaries. It also has a wider application scope than Nero in design.
- We summarize the label noise and sparsity problems and present two data-preprocessing approaches to help mitigate them. In this way, we improve the usability of the large-scale dataset in a (semi-)automated manner.
- We conduct extensive experiments to evaluate NFRE and validate our intuitions. The results demonstrate the significance of data preprocessing and show NFRE outperforms existing techniques, DEBIN and Nero, by a relative improvement of 32% and 16%, respectively, while the time cost for feature extraction is much less.

2 BACKGROUND

Developers usually set descriptive names to describe the functionalities of functions [2]. However, the developer-chosen names no longer exist in stripped binaries. Reassigning descriptive names for functions is an emerging and significant issue. It has various application scenarios, such as building better decompilers, inferring library functions or constructing domain-specific (ad hoc) function name predictors. In general, predicting function names from assembly code is non-trivial, more challenging than that at the source level for the following reasons:

Limited Information. Source code contains a wealth of high-level information (e.g., semantic tokens, abstract syntax tree) that can facilitate the related tasks. Researchers even leverage copy mechanism [32] to "directly" copy tokens from function bodies to names [3, 27] because tokens in names often appear in the corresponding bodies[1]. In contrast, most of the descriptive information is discarded in compiling and stripping, resulting in extremely limited information in assembly code.

Token & Code Diversity. (1) The open property of function naming results in a large vocabulary of tokens. In comparison to the natural language texts, word abbreviations and domain-specific jargons are widely used in function names [27]. (2) Due to compiler differences, optimizations and potential obfuscation techniques, assembly representation is even more diverse than source code, making the assembly-level tasks more challenging.

The former affects the modeling and representation of assembly code, thereby limiting the capability of the machine learning model to fit data distribution. The latter enlarges the search space and exacerbates the data sparsity.

2.1 Motivating Example

A real-world function `aesni_cbc_encrypt` in project `libgnutls` is used as a motivating example. This is an encryption function based on the AES-CBC (Cipher Block Chaining) algorithm, which can be deduced from its name. It is complicated, containing 400+ instructions and 30+ basic blocks. Moreover, it uses the Intel Advanced Encryption Standard Instructions (AES-NI) [37], which is unfamiliar for common practitioners (but may be the distinctive features for the learning-based model). Fig. 1 illustrates the corresponding code snippet. It is hard for human engineers to judge the

[1]According to the statistics of [27], roughly 33% of tokens in names can be copied directly from tokens in the source code of function bodies.

functionality of such a complicated function. We expect to reassign descriptive names for functions in a data-driven manner.

```
C9BA8: aesenc        xmm2,xmm1
C9BAD: dec           eax
C9BAF: movups        xmm1,xmmword ptr [rcx]
C9BB2: lea           rcx,[rcx+10h]
C9BB6: jnz           short loc_C9BA8
C9BB8: aseenclast    xmm2,xmm1
C9BBD: mov           eax,r10d
C9BC0: mov           rcx,r11
C9BC3: movups        xmmword ptr [rsi],xmm2
C9BC6: lea           rsi,[rsi+10h]
```

Figure 1: An assembly code snippet of a complicated encryption function `aesni_cbc_encrypt`.

2.2 Existing Techniques

2.2.1 DEBIN . DEBIN [34] is a non-neural model for predicting debug information of stripped binaries. It can recover variable names and types, function names and types from binary code. Concretely, DEBIN lifts binary code to Intermediate Representation (IR) [13] and then constructs variable dependency graph. Finally, it makes predictions by the Conditional Random Fields (CRFs) [50], which is a probabilistic graphical model. The primary limitations of DEBIN are as follows: (1) The prediction model used in DEBIN is CRFs, not the neural model. As pointed out by [22], DEBIN suffers from inherent sparsity from the model perspective. The advantages of neural models over CRFs in function name prediction task are also discussed in [5]. (2) DEBIN is trained and evaluated in an exactly matching manner. In other words, only the case that the prediction is exactly the same as the label, DEBIN considers it successful; otherwise, fail. Since function names usually consist of several tokens, the exact match will lead to inherent imprecision [22]. The model can only output full function names encountered during training. It has no ability to predict *neologisms* [2], that is, function names that have not appeared in training set.

2.2.2 Nero . Nero [22] is designed for predicting function names in stripped binaries. It is based on the encoder-decoder paradigm, using a Graph Neural Network (GNN) [45] as the encoder and a Long Short-Term Memory (LSTM) network as the decoder. Nero is more suitable for function name prediction. It uses neural model for prediction and token-level metrics for training and evaluation. However, Nero still suffers from the following limitations: (1) Nero utilizes function calls with the restored arguments to model functions. It is an inherent limitation because not every function makes (internal or external) function calls. In addition, Nero is quite dependent on the semantic information provided by the library function names[2]. For the functions (e.g., functions in statically linked binaries) that only have internal function calls, Nero has no semantic information to use except for the restored arguments. However, the restored arguments are low-level and abstract so that they are weak at individually modeling functions. The performance of Nero drops sharply at this time. (2) As we have mentioned in earlier sections, the dataset used by Nero is quite small, merely containing 541 binaries that are compiled by a single compiler *gcc*. In addition, the training and test sets overlap in their dataset, resulting in their results being debatable. In our experiments, we find that Nero overfits the training data and has poor generalization capability for unknown functions (discussed in Section 5.5.2).

[2]According to our statistics, the functions that have library function calls account for roughly 30% of all functions.

2.3 Challenges for Large-Scale Evaluation

It is non-trivial to evaluate existing techniques on large-scale binaries. Both DEBIN and Nero suffer from inefficiency because the features they use require heavyweight binary analysis. In our experiments, when we tried to train them on large-scale and real-world binaries, the process of feature extraction consumes too much time, making the evaluation infeasible in practice. The efficiency problem of DEBIN has also been commented by [49]. Nero is often trapped in complex analysis, resulting in the remaining time tends to be unpredictable. In summary, the complexity of heavyweight analysis makes DEBIN and Nero powerless in the face of big-size and large-scale binaries, preventing them from being trained and evaluated on the large-scale binaries. Therefore, one goal of our research is to build a lightweight framework that uses easier-to-get features for efficient training and inference.

2.4 Challenges for Dataset Construction

It is not easy to construct a large-scale dataset while maintaining good usability. Firstly, a large number of unstripped binaries is required. However, the compilation process is difficult to automate due to the configuration differences of various projects. Under this premise, manual compilation is an option, which is adopted by Nero [22] to generate small-scale binaries. In our opinion, the best practice is to use the compiled software in the real world, just like DEBIN [34]. This practice allows us to obtain enough and diverse binaries and ensures the dataset can reflect the real-world scenarios. However, two problems damage the usability of the dataset, which are *label noise* and *label sparsity*.

Label Noise. Due to potential code obfuscation techniques or excessive abbreviations, there are many functions whose names are non-descriptive or meaningless (e.g., `_vsubfpx`). As the noise samples, they can bias the training process and damage the practical applicability of the model (i.e., making the model less effective in real-world cases) eventually [12, 72].

Label Sparsity. Because of the open property of function naming, the vocabulary of tokens tends to be very large. Since we are currently facing large-scale and diverse data, using the raw tokens directly can lead to severe sparsity problem - even semantically similar tokens are independent of each other. In addition, the overlarge vocabulary will increase the complexity of the model and make it difficult to train.

For the small-scale datasets, the two problems are not severe or can be alleviated by manual inspection. However, purely manual inspection is impractical for large-scale data. The other goal of our

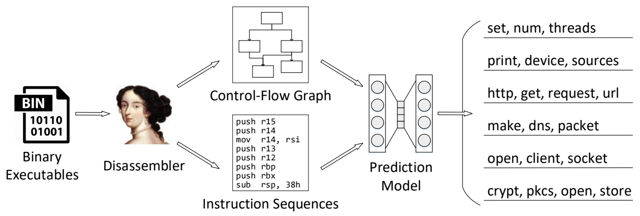

Figure 2: The overall workflow of NFRE.

research is to mitigate the two problems in a (semi-)automated manner.

3 THE NFRE FRAMEWORK

3.1 Overview

NFRE is designed as a plugin in the disassembler such as IDA Pro [36] to automatically suggest descriptive names for functions in stripped binaries. Fig. 2 illustrates the overall workflow. As a prerequisite, the stripped binary is disassembled by IDA Pro, and the functions in it are correctly recognized. Since the symbol table (i.e., `.symtab` section) is discarded, IDA Pro can only set address-related placeholders as the default function names. At this moment, NFRE is ready to work. It is based on the encoder-decoder paradigm, taking the instruction sequences and control-flow information as input and outputting the likely function names. Next, we elaborate on the principle and implementation of the NFRE framework.

3.2 Structural Instruction Embedding

NFRE utilizes the structural information brought by Control Flow Graph (CFG) to facilitate the modeling of assembly code. CFG is a structured code representation with a low overhead of construction. It is widely used in the assembly-level studies [22–24, 69, 75].

The structural information is used in a pre-training manner. First, we perform structural instruction embedding based on CFGs, representing instructions as embeddings (i.e., high-dimensional numerical vectors) that potentially aggregate the control-flow information. Then we use the pre-trained embeddings as the input of the neural model so that the model can benefit from the structural information. Concretely, there are three steps: (1) instruction normalization, (2) instruction-level CFG construction, and (3) graph-based embedding.

3.2.1 *Instruction Normalization.*

Due to the diversity of mnemonics and operands, directly using raw instructions will exacerbate the sparsity of input data, which makes it difficult for the model to learn the data distribution. The model will also suffer from severe out-of-vocabulary (OOV) problem when inferring functions that contain unknown instructions during training.

To mitigate this problem, the instructions should be normalized before participating in experiments[3]. Inspired by the related studies [11, 24, 57, 75], we empirically set the following rules to normalize instructions: (1) Retaining all the mnemonics and registers. (2) Replacing all the constant values with `<POSITIVE>`, `<NEGATIVE>` and `<ZERO>` (i.e., only considering the sign of the value). (3) Replacing all the internal function addresses with `<ICALL>`. (4) Replacing all the library function names with `<ECALL:function_name>`. (5) Replacing all the destinations of local jump with `<LOCALJUMP>`. Fig. 3 illustrates an example of instruction normalization.

[3]Assembly instructions in this paper adopt the Intel syntax, i.e., `op dst, src(s)`.

```
mov     eax, ebx             | mov     eax, ebx
push    80h                  | push    <POSITIVE>
push    0                    | push    <ZERO>
call    sub_4DF0             | call    <ICALL>
call    _fwrite              | call    <ECALL:_fwrite>
jz      short loc_DD58       | jz      <LOCALJUMP>
lea     esp, [ebp-0Ch]       | lea     esp, [ebp+<NEGATIVE>]
lea     edi, [eax+10h]       | lea     edi, [eax+<POSITIVE>]
```

Figure 3: An example of instruction normalization. The left part is the original instructions, and the right part is the normalized instructions.

Instruction normalization is similar to the stemming and lemmatization of words in Natural Language Processing (NLP). The latter is to restore different forms of words to the original forms before actually experimenting. For example, `cats` to `cat` (stemming) and `driving` to `drive` (lemmatization). The purpose of both is to decrease the diversity of data (instructions or words) while keeping the basic semantics unchanged.

3.2.2 *Instruction-level CFG.*

The assembly code snippets can be represented as CFGs according to the control transfer mnemonics such as `jmp` and `je`. A canonical CFG is comprised of basic blocks and jump control flows. The nodes portray basic blocks, and the edges portray jump control flows. In our experiments, we refine the canonical CFG and construct the Instruction-Level CFG (IL-CFG). Specifically, we split the basic block into several instructions and add the sequential control flows. The nodes of IL-CFG portray instructions, and the edges portray control flows (both jump and sequential). Fig. 4 shows an example.

```
4091AE: test  eax,eax
4091B0: je    0x4091BA
4091B2: test  [rbx],0x100
4091B8: jne   0x4091C8
4091BA: mov   rdi,rbx
4091BD: pop   rbx
4091BE: jmp   0x401D60
4091C3: call  0x401DA0
4091C8: mov   rdi,rbx
4091CB: mov   edx,0x1
```

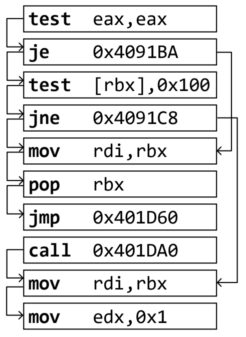

Figure 4: An example of Instruction-level CFG. The left part is the assembly code snippet, and the right part is the corresponding IL-CFG.

Notably, constructing IL-CFG does not require heavyweight binary analysis such as data-flow analysis. Loosely speaking, the disassembler only needs to recognize the control transfer mnemonics so that the construction process is fast.

3.2.3 Graph-based Embedding. We adapt DeepWalk [62] for instruction embedding based on the IL-CFG representation. DeepWalk is an unsupervised graph embedding technique. It can generate structure-sensitive instruction embeddings, allowing NFRE to use the control-flow information. DeepWalk is internally based on the Skip-gram [58] model for embedding. The Skip-gram model is initially used in NLP for word embedding. In the following, we briefly introduce the Skip-gram model and the DeepWalk algorithm.

Skip-gram. Skip-gram [58] is a widely used word embedding model. It learns embeddings from the context in which a word occurs. As a result, if two words have similar meanings or usage, their embeddings will be close to each other in the high-dimensional space; otherwise, far away (e.g., "`Paris`" is close to "`Tokyo`", while far away from "`Apple`"). The essence of the Skip-gram model is a shadow neural network with only one hidden layer, and it uses the current word to predict the surrounding words. In practice, there is a sliding window moving on the text, treating the middle word as input and targeting the other words in the window. The training sample exists in pair (x, y), where x is the input, and y is the target. The training objective is to adjust word embeddings so that they can be used to predict the surrounding words accurately.

DeepWalk. Graph embedding is a conversion of graph data (e.g., nodes, edges, substructures, or the whole graph) into embeddings. It learns a mapping from the graph into the embedding space in which the relevant information of the graph is maximally preserved [14]. DeepWalk [62] is an unsupervised graph embedding algorithm. It works based on the Skip-gram model. There are two main steps: (1) Sampling node sequences through a random walk strategy. (2) Using the sampled sequences as corpus and adapting the Skip-gram model for node embedding. Since the node sequences are generated by a random walk on the graph, the node embeddings potentially aggregate the structural information of the graph.

The original DeepWalk algorithm runs on a single large graph. However, our data is a batch of IL-CFGs. Here we adopt an alternative strategy to suit our needs. To generate the training corpus for the Skip-gram model, we perform the random walk and sample instruction sequences on each IL-CFG. In other words, we assume that there is a large virtual graph that contains all the instructions and control flows of the dataset. Each IL-CFG is a sub-graph of it. We run the DeepWalk algorithm on each sub-graph instead of running on the virtual graph. In the pre-experiments, we found it worked.

3.3 Neural Prediction Model

The prediction model of NFRE is based on the encoder-decoder paradigm. Given a disassembly function body, it takes the normalized instruction sequences as input, which naturally utilizes the sequence information of assembly code. After inference, it outputs the tokens that form the function name (i.e., from instruction sequence to token sequence). In the following, we briefly introduce the encoder-decoder paradigm.

Encoder-Decoder Paradigm. Encoder-decoder paradigm is widely used for sequence translation tasks, such as Neural Machine Translation (NMT) [8, 65, 66]. The input sequence is encoded into a context vector by the encoder, while the decoder decodes the vector into target data. The types of encoder and decoder are usually Recurrent Neural Network (RNN) to handle variable-length sequences [8, 20, 65]. Some studies also use other structures, such as Convolutional Neural Network (CNN) [18], Transformer [66] and GNN [9, 10, 22].

The prediction model is simple yet effective. The encoder of our model is a Bidirectional LSTM (BiLSTM) network. Compared with the (unidirectional) LSTM, it can capture the association of both the current instruction and the previous/next instruction. The decoder is an attentional LSTM network that is incorporated with the attention mechanism [56] to capture the long-distance dependencies of instructions.

3.3.1 Fine-Grained Utilization of Library Functions. There are two linking mechanisms for binary executables, static or dynamic linking—the former bakes all the library functions required for the program into executables at the linking stage. The latter only bakes the references (i.e., library function names) into files, and the actual linking is performed by OS when the binaries load to the memory. For the dynamically linked binaries, library function names are retained after stripping because the Linux OS needs to locate the function bodies from libraries by names. The intact library function names are precious because they bring additional semantic information. As the callee, the library function is directly related to the functionality of the caller. Our statistics show that over 30% of functions have at least one token, which appears both in its name and the library function names it invokes.

NFRE utilizes the library function names in a fine-grained manner. Inspired by the practice of [22], we tokenize library function names into multiple tokens. Then we use them along with the instruction sequences as input in the hope that they can provide auxiliary information for the prediction model. The intuition is that each token is potentially related to the functionality of the caller. Compared with treating each library function as a whole, this practice can utilize the semantic information in a fine-grained manner and alleviate the OOV problem.

4 DATASET CONSTRUCTION

In this section, we elaborate the methodology of dataset construction, including two data-preprocessing approaches for mitigating label noise and sparsity problems.

4.1 Overview

We opt for the software provided by Ubuntu [15] as the data source. As a popular Linux OS, Ubuntu provides its users with the compiled software packages (`*.deb` files) for easy installation, covering tens of thousands of software projects (e.g., `redis`, `nginx` and `gcc`). The debug symbol packages (`*.ddeb` files) are also offered for debugging [16]. The software is compiled by different compilers under various compiler options, ensuring a rich variety in our dataset. We use IDA Pro [36] to disassemble binaries, assigning the developer-chosen names to the functions in stripped binaries by address. In accordance with [22, 34], we choose the programs written in C language to construct our dataset.

Table 1 details the statistics for the raw datasets (`NFRE-x86/x64-Raw`). For each architecture, almost 30,000 binaries from roughly

7,400 projects are involved in the dataset construction. Table 1 also shows the statistics for the datasets used by DEBIN and Nero. Since the authors of DEBIN have not publicly released their dataset so far, we can only get the limited information from their paper [34]. In comparison with the datasets used in existing studies, our dataset contains much more binaries. On the one hand, this makes our experiments more persuasive. On the other hand, it also increases the difficulty of data preprocessing and puts forward higher requirements on the time efficiency of our framework in turn. Here we additionally give the volume information to gain a general understanding of our dataset: The total size of unstripped binaries is about 14GB (13.18GB of x86 and 15.04GB of x64). About 91% (92.53% of x86 and 90.65% of x64) of binaries are less than 1MB, and about 1.6% (1.29% of x86 and 2.04% of x64) of binaries are larger than 5MB. The average number of instructions per function is roughly 90 (96.78 of x86 and 86.52 of x64).

Table 1: Statistics on the number of projects, binaries, functions of the datasets.

Dataset	# Projects	# Binaries	# Functions
NFRE-x86-Raw	7,375	29,696	3,742,027
NFRE-x86-Denoised	-	-	3,098,486
NFRE-x86-Deduplicated	-	-	1,457,426
NFRE-x64-Raw	7,408	27,686	3,335,643
NFRE-x64-Denoised	-	-	2,886,458
NFRE-x64-Deduplicated	-	-	1,361,092
DEBIN [34][1]	-	3,000	-
Nero [22][2]	91	541	67,880[3]

[1] DEBIN has three versions for x86, x64, and arm binaries. For each version, the dataset consists of 3,000 binaries. Since the authors have not publicly released their dataset, we can only get limited information from their paper.
[2] The dataset of Nero only contains x64 binaries.
[3] Without function-level deduplication.

4.2 Label Noise Mitigation

To help alleviate label noise, we adopt a detection-based strategy. The general idea is to find as many noise samples as possible and then filter them out from the dataset. To this end, a binary classifier is built to judge whether a function name is descriptive (meaningful). It is inspired by the practice of Domain Generation Algorithms (DGAs) detection [60, 73]. DGAs are used to automatically generate pseudo-random names (e.g., `asedfvfwk.com`) to evade blacklist-based detection. Distinguishing them from the normal names is similar to the current problem. Specifically, we use the n-gram of function names as features. We opt for unigram ($n = 1$) and bigram ($n = 2$), building a 702-dimensional ($26+26 \times 26$) feature vector for each name. The frequency of the combinations of n characters is counted to compose different dimensions. Then we leverage Gradient Boost Decision Tree (GBDT), an ensemble machine learning algorithm, to build the classifier. In the following, we briefly introduce the GBDT algorithm.

Gradient Boost Decision Tree. Decision Tree is a supervised learning model. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules, and each leaf node represents the outcome. The input propagates from the root to a leaf node of the tree, where the final classification decision is made. Gradient Boost Decision Tree [28, 29] is an ensemble algorithm. In principle, it builds one decision tree at a time and corrects the error made by the previous tree. Predictions are based on the entire ensemble of trees together that make the final prediction.

Training such a binary classifier needs a large number of annotated samples (both meaningful and meaningless). The meaningful function names are collected from the most popular projects in GitHub [31]. The assumption is that the popular projects are generally well-documented, and the functions are usually well-named. We automatically scrape thousands of top-starred projects, covering Java, Python and C languages. Then we use parsers [25, 26] in combination with regular expressions to extract function names. As a result, we get roughly four million function names as meaningful samples. As for meaningless samples, we use a trick in this case. Since the meaningless function names are essentially pseudo-random texts, we write a Python script to generate pseudo-random texts just like DGAs, and then we use the outputs as the meaningless samples. According to our preliminary evaluation, the classifier achieves roughly 80% F1-score (based on a dataset of 2,000 representative samples screened manually, in which the ratio of positives and negatives is 1:1). Given a binary executable, if more than 60% of the functions are judged to be noise samples, we discard all the functions in it.

Table 2: Qualitative examples of the detected non-descriptive function names.

Project	Meaningless Function Names
weight-watcher	zpnrev, codfwd, copfwd, airfwd, coofwd, sphfwd, tnxfwd, qscfwd, molfwd, cscfwd, cypfwd, glsfwd, wcsfwd, aitfwd, tnxrev, tscfwd, pcofwd, ...
yabause-qt yabause-gtk	OP_0x0010, OP_0x0018, OP_0x001F, OP_0x0020, OP_0x0027, OP_0x0028, OP_0x0030, OP_0x0038, OP_0x0039, OP_0x003C, OP_0x0040, ...
qemu-user-static	_vslh, _vslw, _vsr, _vsrb, _vsrh, _vsrw, _vsubcuw, _vsubeuqm, _vsubfp, _vsubuwm, _vupkhpx, _vslb, _vupkhsb, _vupklpx, _vupklsb, _xmknod, ...
libz80ex1	op_CB_0xaf, op_CB_0xdf, op_DDCB_0x00, op_CB_0xef, op_CB_0xfd, op_DDCB_0xfe, op_FDCB_0x00, op_FDCB_0x01, op_FD_0x02, ...
chicken-bin	f_1028, f_10284, f_10286, f_10287, f_1029, f_10290 f_10292, f_10296, f_10296_0, f_10297, f_10304, f_10306, f_10308, f_10318, f_10322, f_10324, ...

Table 2 shows the qualitative examples of the detected noise samples. By analyzing the detected function names, we discover some naming patterns (e.g., `OP_XXXX`, `f_XXXX` and `op_XXXX_XXXX`). They may be generated by specific code obfuscation techniques. According to the fixed naming patterns, we use regular expression-based matching to further find out the pseudo-negative samples that are missed by the detection.

The statistics for the denoised datasets are shown in Table 1 (`NFRE-x86/x64-Denoised`). About 15% (17.20% of x86 and 13.47% of x64) of the functions are classified into noise samples and removed from the datasets.

4.3 Label Sparsity Mitigation

The general idea to mitigate label sparsity is to establish associations among tokens that have similar meanings. We should find out these tokens first. As mentioned in earlier sections, the dictionary- or rule-based stemming and lemmatization in NLP can restore different forms of words to the original forms. However, they are weak at processing function names due to the use of abbreviations, which is a primary reason for the diversity of function names [27]. Most abbreviations are not standard English words (e.g., `cmp` for `compare`, `conn` for `connect`) and they are prone to be OOV. In addition, synonyms are also interchangeably used in function naming, such as `argument` and `parameter`.

In our experiments, we notice that *the semantically similar tokens usually have similar contexts*. For example, given two function names `getMessageType` and `getMsgType`. The `message` and the `msg` are standard word and its abbreviation, while other tokens (`get` and `type`) in the two function names are the same or similar. Based on the above observation, a general idea is to leverage Skip-gram [58] for token embedding, and then find out context-similar tokens based on the cosine distance. The Skip-gram model is detailed in earlier sections.

We propose a hybrid approach that combines data-driven idea and empirical rules to summarize semantically similar tokens. Firstly, we adapt the Skip-gram model to perform token embedding. We treat each function name as a separate window. Each token in the function name will be treated as input while the other tokens as targets. Fig. 5 illustrates an example. After embedding, tokens with similar contexts will be closed to each other in the embedding space. Next, for each token t, we extract the top n tokens closest to it as the candidates l_t. Formally, the overall candidate list L can be represented as

$$L = [l_1, ..., l_P] \quad (1)$$

$$l_t = [\alpha_1, ..., \alpha_n], \quad 1 \leq t \leq P \quad (2)$$

where P is the number of unique tokens. α_i is one of the candidate tokens. In our experiments, the size of token embeddings is 128 and n is set to 80.

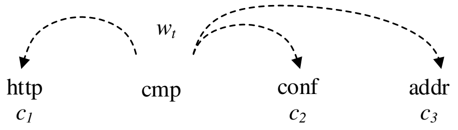

Figure 5: The function name is http_cmp_conf_addr**, which can be split into four tokens:** http**,** cmp**,** conf **and** addr**. When performing token embedding, we use the current token** w_t (cmp) **to predict the contexts** c_1 (http)**,** c_2 (conf) **and** c_3 (addr)**.**

The data-driven idea allows us to obtain each token with the top n tokens with similar contexts. In order to further narrow the scope, we use the empirical rules to filter the candidates. Given a token a in l_b and a token b in l_a, if they meet any of the following rules, we consider that they may be semantically similar:

(1) a starts with b or b starts with a.
(2) The first letter of a is the same as the first letter of b, and the Levenshtein similarity [6] between a and b is larger than 0.6.
(3) The last letter of a is also the same as the last letter of b, and the Levenshtein similarity is larger than 0.6.
(4) a and b are synonyms.

The first rule is for the case of `conn`, `connect`, `connecting` and `connected`. The second rule is for the case of `send` and `sent`. The third rule is for the case of `str` and `cstr`. The last rule is for the case of `argument` and `parameter`. We utilize WordNet [59] to judge whether two words are synonyms. In summary, the first three rules are about the morphology, while the last one is based on semantics.

In our experiments, the hybrid approach automatically summarized about 7,000 groups of tokens. Then we performed the manual inspection for more accurate results. Since the hybrid approach has dramatically narrowed the scope, empirical inspection is entirely feasible at this time, which removes unreasonable tokens or entire groups. Overall, the inspection cost about 4 man-hours, and roughly 2,000 groups of tokens remain. According to our statistics, the summarized tokens account for roughly 10% of the vocabulary, but they appear in more than 80% of function names. The qualitative examples are shown in Table 3. The hybrid approach can find out the different forms of a word (e.g., `equal`, `equals` and `equivalent`), the abbreviations (e.g., `dbg` and `debug`) and the synonyms (e.g., `argument` and `parameter`).

Table 3: Qualitative examples of the semantically similar tokens.

Word	Semantically Similar Tokens
parameter	parms, argument, arguments, args, arg, param, ...
subscribe	subscription, subscriber, subscriptions, ...
debug	dbg, debugger, debugging
print	fprint, sprintf, fprintf, oprint, printf, eprintf
equal	equals, equivalent, eq, unequal
attribution	property, attributes, prop, attr, attribute, properties
authorization	auth, authentication, authenticate, oauth, ...
error	err, error, errno, errors, errorhandler
compare	cmp, comparison, comparator, comp, comparable, ...
function	func, procedure, funcs, procedures

For the training set, we restore the tokens in a cluster (which is merged by groups and consists of all semantically similar tokens) to a single token. When testing or inference, if the model outputs a token in the same cluster as ground truth, we deem it a success.

4.3.1 *Synonyms Effectiveness.* Here we elaborate on the role of synonyms for mitigating label sparsity. The synonyms mainly serve as *bridges*. For instance, the embedding-based mining and morphology-based rules can discover (`argument,arg,args`) and (`parameter,param,params`), but they cannot associate the two clusters. Once `argument` and `parameter` are identified as synonyms, the two clusters can merge into one. There are some other cases, e.g., (`function,func,funcs`) and (`procedure,procedures`).

4.4 Deduplication

Prior studies [22, 34] perform file-level deduplication in their experiments. In other words, there are no two identical binaries in their dataset. However, for reasons such as code reuse and the use of libraries, such a coarse-grained practice cannot avoid function-level duplication. When checking the dataset released by Nero [22], we found that some functions (e.g., `_start`) in the test set also appear in the training set. From the data mining perspective, it is a mistake called *data leakage* [43], resulting in exaggerating results. Nevertheless, some researchers argue that it is reasonable to allow such duplication since reverse-engineering binaries that link against known libraries is a realistic use case [49].

In this paper, we adopt a more comprehensive strategy. Firstly, we perform function-level deduplication before splitting the dataset into training, validation, and test sets. As shown in Table 1 (`NFRE-x86/x64-Deduplicated`), the number of functions has been reduced by roughly 53% (52.96% of x86 and 52.85% of x64). It is thorough deduplication to avoid double counting. Then we refer to the practice of [49]. For each model, we use two test sets for evaluation. The first set is composed of known functions, which means the functions have appeared in the training set. The second set is comprised of unknown functions (i.e., the functions that do not appear in the training set). It is the actual test set from the data mining perspective. We randomly select the same number of samples from the training set as the actual test set to form the first set. In this way, we can observe not only the performance of NFRE for known functions, but also its generalization capability for unknown functions.

5 EVALUATION

We conduct extensive experiments to answer the following research questions:

RQ1. How effective is NFRE in assigning names to functions of stripped binaries? (Section 5.2)
RQ2. How does each component of NFRE contribute to its efficacy? (Section 5.3)
RQ3. Do the label noise and sparsity problems indeed affect the usability of the dataset? (Section 5.4)
RQ4. How does NFRE perform in comparison with existing techniques? (Section 5.5)

5.1 Experimental Setup

5.1.1 Environment. We used an Ubuntu 16.04 machine with Intel Core i7 8700k, GeForce GTX 1080Ti and 64GB RAM to perform experiments. We used Python language with OpenNMT [46], Gensim [64], scikit-learn [61], XGBoost [19] and python-Levenshtein [6] to implement the framework.

5.1.2 Dataset Partition. We split the dataset into training, validation, and test sets at the function level. The number of functions in the training set accounted for 80% of the whole dataset. The validation set and the test set accounted for 10%, respectively.

5.1.3 Function Name Tokenization. For the function names that follow the canonical naming conventions (e.g., camel case like `setConnPort` and snake case like `set_conn_port`), we can tokenize them via the symbol "_" or upper case letters. However, function names without obvious splitting marks, such as `setcmdfmt`, are also prevalent in the dataset. For these function names, we followed the practice in [49], using SentencePiece [48] to tokenize them. It splits function name `setcmdfmt` into `set`, `cmd` and `fmt` based on the statistics of token frequency [47].

5.1.4 Hyper Parameters. The dimension of instruction and token embeddings is 128. We used a 2-layer BiLSTM with 500 hidden states as the encoder and a 2-layer LSTM with 500 hidden states as the decoder. The dropout rate is experimentally set to 0.3, randomly dropping 30% of the cell's output. We adopted negative log-likelihood loss and the teacher forcing strategy for training. Then we trained the model by Adam optimizer [44] with an initial learning rate of 0.001. The batch size is 64. We trained the model for 1000 thousand steps at most and performed validation every 20 thousand steps. If the performance does not improve for 3 validation steps (early stopping) or the gradient vanishes, the training will be terminated. We used the beam search strategy in inference, and the beam size is set to 10.

5.1.5 Metrics. In accordance with Nero [4, 5, 22], we used Precision, Recall, and F1-score to evaluate the performance of NFRE with its variations and the existing techniques. The metrics are token-level, case-, order-, and duplication-insensitive, and ignoring non-alphabetical characters. Given a predicted function name $\hat{X} : \{\hat{x_1}, ..., \hat{x_j}\}$ and the ground truth $X : \{x_1, ..., x_k\}$,

$$\text{TP} = \sum_{i=1}^{j} \mathbb{I}\{\hat{x_i} \in X\}, \text{FP} = \sum_{i=1}^{j} \mathbb{I}\{\hat{x_i} \notin X\}, \text{FN} = \sum_{i=1}^{k} \mathbb{I}\{x_i \notin \hat{X}\} \quad (3)$$

the metrics are defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (4)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (5)$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6)$$

where x_i ($\hat{x_i}$) is a token in function name, $\mathbb{I}$ is the indicator function, $\mathbb{I}\{true\} = 1$ and $\mathbb{I}\{false\} = 0$. The results presented in this paper are the average results. It should be noted that there is another definition of Precision and Recall [3], and the difference is in the denominator. For Precision, they use the length of $\hat{X}$ (i.e., j) as denominator, which is equal to TP + FP. For Recall, they use the length of X (i.e., k) as denominator, which is not always equal to TP + FN. This definition also makes sense. In the pre-experiments, we found the difference of two definitions has slight effect on results.

When comparing with existing techniques, we additionally introduced a metric to measure the efficiency, which is the average time cost of feature extraction for each binary executable. It can reflect the feasibility of the technique in the real world. We tried to align the model configurations (e.g., workers) to make the results reasonable. Note that we omitted the training time of prediction models because it is highly related to the training steps and equipment. In fact, we found that the training of model is not the main concern for time efficiency because the computation of neural networks can be greatly accelerated by GPU. It is the binary analysis stage that most affects the overall training and inference time.

5.2 Overall Effectiveness

The results are shown in the last row of Table 4. When assigning names for known functions, NFRE achieves roughly 40% precision, 34% recall, and 36% F1-score. When assigning names for unknown functions, NFRE maintains an acceptable capability of generalization, achieving more than 32% precision, 27% recall, and 28% F1-score. It should be noted that the performance of the machine learning model on the training set (known functions) and test set (unknown functions) is not always positively correlated. There is a trade-off, and an extreme case is overfitting. At that time, the model performs quite well for the known samples but poorly for the new samples. The left side of Table 5 displays the qualitative examples of the predictions for function samples, such as file operation, and network communication.

Answer to RQ1: Overall, NFRE achieves 36% precision, 31% recall, and 32% F1-score for the function name reassignment in stripped binaries.

5.3 Component Contribution

We conducted an ablation study to validate the contribution of structural information and library function names to the overall performance of NFRE. For the NFRE that ablates structural information, we used the randomly initialized instruction embeddings as the input of the prediction model. For the NFRE that ablates library function information, we no longer used the tokens from library function names as additional input. Instead, we treated all library function names as indistinguishable and assigned them a uniform name `<ECALL>`. It also represents the scenario where the library function name is obfuscated. At this time, NFRE has to perform inference in the absence of the semantic information provided by library function names. As a reference, we also show the results of the basic NFRE that neither uses structural information nor library function information in the antepenult row of Table 4.

The upper part of Table 4 shows the results for the variants of NFRE that ablate different components. Briefly, both the structural information and library function names contribute to the ultimate effectiveness of NFRE. The right side of Table 5 shows the predictions made by the incomplete NFRE. The ablation of any component will affect the overall effectiveness of NFRE.

Answer to RQ2: Each component of NFRE contributes to the overall effectiveness.

5.4 Data Impact on Model

Based on the intuition that the label noise and sparsity problems will affect the usability of the dataset, we present two data-preprocessing approaches described in earlier sections for mitigation. Here we validated the significance of this practice. For each architecture, we randomly selected some samples from the corresponding raw dataset (i.e., `NFRE-x86/x64-Raw`) and constructed a sub-dataset. The number of functions in sub-dataset is the same as that of the preprocessed dataset (i.e., `NFRE-x86/x64-Deduplicated`). Then we trained the basic NFRE on the sub-dataset while keeping the hyper-parameters unchanged.

The results are shown in the penultimate row of Table 4. In consistent with our previous expectations, the model trained on the raw dataset has lower performance than that on the preprocessed dataset. It demonstrates the significance of mitigating label noise and sparsity problems.

Answer to RQ3: The label noise and sparsity problems indeed affect the usability of the dataset and hurt the performance of the model. It is significant to perform data preprocessing for mitigation.

5.5 Comparison with Existing Techniques

We compared NFRE with DEBIN and Nero, respectively. Like [49], we encountered the problem that it is impractical to train DEBIN and Nero on our full dataset due to time and resource restrictions. Therefore, we adopted an alternative but reasonable strategy: We trained NFRE on their dataset (or the same scale dataset as theirs). In the interest of fairness, we did not introduce any data-preprocessing approaches (including deduplication) described in earlier sections to avoid the potential bias in favor of NFRE.

5.5.1 Comparison with DEBIN. Since the authors of DEBIN have not publicly released the dataset till now, we built the same scale dataset with our data. As mentioned in their paper [34], for each architecture, the authors used a dataset composed of 3,000 binaries to evaluate DEBIN (2,700 for training and 300 for testing). We followed their practice and randomly selected 3,000 binaries (per architecture) from our dataset. Then we trained and evaluated DEBIN and NFRE on the alternative dataset. Notably, DEBIN also chooses the software provided by Ubuntu as data sources so that this strategy is generally reasonable.

The results are summarized in Table 6. NFRE achieves higher precision, recall, and F1-score than DEBIN while consuming much less time. Particularly, NFRE outperforms DEBIN by a relative F1-score improvement of roughly 32% (33.90% of x86 and 30.56% of x64). As for time efficiency, DEBIN needs to perform heavyweight binary analysis, including binary lifting, variable recovery, and data-flow analysis, to construct the variable dependency graph. The entire process is much more time-consuming than CFG construction, causing DEBIN to consume more than twenty times that of NFRE. Considering the limitations of DEBIN (i.e., the non-neural design and the exact match problem), the overall result is in line with our expectations.

5.5.2 Comparison with Nero . The authors of Nero have released their dataset at [71]. They divide the binaries into three folders, `TRAIN`, `VALIDATE` and `TEST`, which contain 483, 45 and 13 binaries, respectively. We trained NFRE based on their original partition of the dataset.

The results are summarized in Table 7. We present two sets of results. The former is under the original setup used in their paper, which means that the training and test sets overlap and both the training and test sets contain duplicate functions. However, the results obtained in such a data-leakage manner cannot reflect the generalization capability of the model for unknown functions. To this end, we performed another statistic. In this case, we only

Table 4: Experimental results of NFRE and its variations that ablate different components.

Model	x86			x64		
	Prec.	Rec.	F1.	Prec.	Rec.	F1.
NFRE w/o Structural Information	35.55 / 30.83	29.55 / 25.63	31.23 / 27.06	40.74 / 30.03	**35.71** / 24.95	**37.05** / 26.31
NFRE w/o Library Function Names	31.02 / 27.42	27.23 / 24.03	29.41 / 26.10	38.75 / 28.25	32.43 / 24.23	34.17 / 24.82
NFRE w/o StrucInfo. & LibF.	30.79 / 26.66	25.62 / 22.40	27.06 / 23.52	37.40 / 27.12	31.83 / 22.92	33.40 / 24.01
NFRE w/o StrucInfo. & LibF. (Raw Data)	26.41 / 22.17	21.61 / 21.31	22.95 / 22.33	33.05 / 22.73	27.79 / 21.86	29.29 / 22.88
NFRE	**39.41 / 32.35**	**34.19 / 28.19**	**35.66 / 29.25**	**41.00 / 31.10**	34.71 / **26.35**	36.50 / **27.52**

The former results come from the evaluation on known functions, and the latter results come from the evaluation on unknown functions.

Table 5: Qualitative examples of the predictions made by NFRE and its variations that ablate different components.

Prediction Model Label	NFRE	NFRE w/o StrucInfo.	NFRE w/o LibF.	NFRE w/o both.
stop timer	reset **timer**	set i **timer**	set **timer**	sig handler
rt read packet vers a tile	**read packet**	pcap **read packet**	**read packet**	xf rd recv **packet**
jpeg encode raw	**jpeg** decode frame	**jpeg** get data	x frame decode frame	mus percent convert
write event tree to print string	**print tree**	**write** node	**print tree**	**print** p p results
string free erase	**free**	x **free**	cleanup	lambda widget destroy event
active connection get by path	find **connection by** name	**get** device **by** name	**get** name	**get** uri
report create	list **create**	hash table **create**	acpi ut **create** mutex	acpi ut **create** object
sec pkcs content type	**sec pkcs** choose **content type**	get **content** length	**sec pkcs** hash	ssl cipher pref get
ssl load priv key	**ssl load key**	**load** cert	**load key** file	open file
set input name	blk id partition **set name**	**set** string	gw db category **set name**	config **set** string

Table 6: Comparison with DEBIN.

Model	x86			x64			Time
	Prec.	Rec.	F1.	Prec.	Rec.	F1.	
DEBIN [34]	29.48	29.30	29.35	27.27	27.17	27.19	>90s
NFRE	**38.94**	**40.10**	**39.30**	**35.12**	**36.64**	**35.50**	**≈4s**

counted functions that do not appear in the training set, producing the latter results.

As shown in Table 7, NFRE outperforms Nero by a relative F1-score improvement of 16.23% (under the original setup) and 127.75% (eliminating data-leakage). Most notably, the metrics of Nero and NFRE drop sharply when the training and test sets do not overlap. It indicates that the models suffer from serious overfitting at the moment, so the generalization capability is weak. This phenomenon is predictable. It confirms our standpoint: For the current issue, the learning-based model cannot effectively learn the data distribution from such a small dataset, and it is prone to be overfitted. Comparing the results of NFRE in Table 4 and that in Table 7, we can observe the difference in the generalization capability of the learning-based model trained on different scale datasets. Benefiting from the large-scale dataset, NFRE in Table 4 has better generalization capability for unknown functions. It demonstrates the significance of the large-scale data for the current issue. Table 7 also shows the time efficiency of the two models. Since recovering call arguments requires deep data-flow analysis, the overhead is also expensive, resulting in the inefficiency of feature extraction. Benefiting from the easier-to-get features, NFRE is much more efficient than Nero.

Answer to RQ4: In comparison with existing techniques that require heavyweight analysis, NFRE is much more effective and efficient.

Table 7: Comparison with Nero.

Model	Prec.	Rec.	F1.	Time
Nero [22]	40.17 / 2.26	39.86 / 2.14	39.87 / 2.09	>90s
NFRE	**47.00 / 5.16**	**46.25 / 4.79**	**46.34 / 4.76**	**≈4s**

The former results come from the original setup in the paper of Nero. And the latter results come from the statistics after eliminating data leakage.

6 DISCUSSION & LIMITATION

We have demonstrated the superiority of NFRE over existing techniques. In this section, we discuss the failure modes and the practical applicability of our approaches.

Failure Modes. We analyze the failure modes of NFRE in the hope of providing insight for subsequent studies. It should be noted that the failure modes of a learning-based framework are complex, involving data and models. Since the interpretability of deep neural networks is not clear, we can only tentatively explore from the data perspective. Specifically, we focus on the following factors: (1) The ratio of input (i.e., # of instructions) to output length (i.e., # of tokens). (2) The token imbalance in function names. The former is raised based on the property of the sequence-to-sequence model, while the latter is presented from the perspective of data imbalance. We have experimentally explored the effect of the above factors on the model performance, and here we briefly describe and analyze our findings. For the former factor, we found that the ratio of input to output length is negatively correlated with the overall prediction results. In other words, given a function, the more disparate the number of instructions to the number of tokens, the worse the results tend to be. For the latter factor, we found the function names consisting of high-frequency tokens tend to have better scores, while the low-frequency tokens are usually hard to be predicted correctly. According to our experiments, we found that the characteristics of tokens exhibit long-tailed distribution, where small number of tokens (e.g., `get` and `set`) appear very often while most of the others appear more rarely. The model tends to output the high-frequency tokens to get higher scores but neglects the learning of low-frequency tokens. The reason for this phenomenon is data imbalance, which has generally affected learning-based techniques. Approaching long-tailed distribution data is a promising direction in the field of machine learning-based application.

Practical Applicability. In our experiments, we built a domain-unspecific dataset, which means the dataset tries to simulate all possible cases of function naming. Existing studies [22, 34] also adopt this strategy to construct their datasets. Although NFRE has better performance than existing techniques, the metrics are still relatively low, especially for the inference of unknown functions. Therefore, the current NFRE can hardly be said to have good practical applicability. Nevertheless, we emphasize that the low metrics are mainly affected by the dataset. Such a domain-unspecific and diverse dataset allows to evaluate the model for general purpose, but it also causes the complexity of the current issue to tend to be infinite. Empirically, if the issue can be limited in scope (e.g., domain-specific prediction), the complexity will be much lower. In this way, the distribution of samples in the dataset will be more clear, and the learning-based model can also achieve higher performance and show better practical applicability. Domain-specific application is also a promising direction for the current issue. In summary, the current issue is worth studying and requires more research efforts.

7 RELATED WORK

In this section, we briefly survey the related work about data-driven function name prediction and debug information recovery.

Predict Function Names from Source Code. Predicting function names from source code usually refers to the source code summarization task, which generates brief natural language descriptions for source code snippets. Allamanis *et al.* [3] presented a convolutional attention network to summarize source code. LeClair *et al.* [51] proposed a NMT-based model that used code sequences and Abstract Syntax Tree (AST) as input. Their model has two independent encoders. One is used to accept the code sequences, and the other is used to accept AST. Alon *et al.* [5] proposed a path-based attention model to embed code snippets. They represented code snippet as AST and discomposed AST as a bag of paths. Fernandes *et al.* [27] proposed a GNN-based model. They used a GNN-based encoder to encode AST and an RNN-based encoder to encode code sequences. Ahmad *et al.* [1] proposed a Transformer-based approach to summarize code. Since the function bodies usually contain some tokens used in the corresponding function names, some researchers used the copy mechanism to directly copy tokens from function bodies [3, 27].

Predict Function Names from Binary Code. DEBIN [34] and Nero [22] are the only two published studies on this issue, which have been detailed in earlier sections. Since Nero [22] also opts for the neural prediction model based on the encoder-decoder paradigm, it is necessary to make a comparison between Nero and NFRE. From the model perspective, the primary difference is the encoder network. Nero opts for a 4-layer Graph Convolutional Network (GCN) [45] as the encoder to utilize the structural information brought by CFG. Since the multiple propagations between a node and its nearby neighbors, the multi-layer GCN model suffers from the over-smoothing problem [54, 74]. In brief, the features of nodes within each connected component gradually converge to the same value during training. As a result, it makes the nodes indistinguishable, thereby hurting the model performance. In contrast, BiLSTM encoder is maturer. It has better scalability and robustness in the face of large-scale and diverse data. Additionally, Nero merely uses a GCN-based encoder, which means the abandonment of the sequential information of assembly code [68]. It utilizes the tokens used in library function names and the restored arguments as features, resulting in narrower application scope (discussed in Section 2.2.2). By contrast, NFRE uses the instruction sequences as features, more general and fine-grained. It can capture the instruction-level differences, which is more subtle.

Predict Debug Information in Binaries. It is recognized that debug information can be of great help to reverse engineering. In recent years, many researchers focus on the recovery of debug information in stripped binaries. They put efforts to predict variable names [34, 39, 40, 49], variable types [34, 53], function names [22, 34], function types [21, 34] and library function names [38]. We recommend the interested readers refer to the above papers.

8 CONCLUSION AND FUTURE WORK

We study the issue of reassigning descriptive names for functions in stripped binaries. To overcome the inefficiency caused by heavyweight binary analysis, we present the Neural Function Rename Engine (NFRE), a lightweight framework for function name prediction. Additionally, we construct a large-scale dataset for training and evaluation, and we propose two data-preprocessing approaches to help mitigate label noise and sparsity problems. Experimental results demonstrate the significance of our data-preprocessing approaches and show that NFRE is much more effective and efficient than existing techniques. NFRE outperforms DEBIN and Nero by a relative F1-score improvement of 32% and 16%, respectively, while the time consumed for binary analysis is much less.

In this paper, we conduct research entirely at the assembly level. The features used in NFRE do not involve the higher-level abstractions brought by decompilation. On the one hand, this is for lightweight purpose. The decompilation will affect the efficiency of our framework. On the other hand, this is to not introduce additional undecidability. The correctness of the decompiled output is our main concern. Nevertheless, a recent study [55] points out that modern C decompilers have been progressively improved to generate quality outputs. In the future, we plan to introduce the output of the decompiler into NFRE to see if it facilitates the performance. We also call for more research efforts to the current issue from data and model perspectives.

ACKNOWLEDGMENTS

We thank the anonymous ISSTA reviewers for their valuable comments. This work was supported in part by the Natural Science Foundation of China under Grant U20B2047, 62072421, 62002334 and 61972373, Exploration Fund Project of University of Science and Technology of China under Grant YD3480002001, Fundamental Research Funds for the Central Universities under Grant WK2100000011, National Key Research and Development Program of China under Grant 2020YFA0309702, Public Service Platform Project for Industrial Technology Foundation under Grant 2019-00893-2-2, and by Anhui Initiative in Quantum Information Technologies under Grant AHY150400.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 38–49. https://doi.org/10.1145/2786805.2786849

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[6] Antti Haapala. 2021. python-Levenshtein. https://github.com/ztane/python-Levenshtein.

[7] Avast Software. 2021. RetDec. https://retdec.com.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[9] Joost Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. 2017. Graph Convolutional Encoders for Syntax-aware Neural Machine Translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1957–1967. https://doi.org/10.18653/v1/D17-1209

[10] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. 2018. Graph-to-Sequence Learning using Gated Graph Neural Networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 273–283. https://doi.org/10.18653/v1/P18-1026

[11] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., 3585–3597. https://proceedings.neurips.cc/paper/2018/file/17c3433fecc21b57000debdf7ad5c930-Paper.pdf

[12] F. A. Breve, L. Zhao, and M. G. Quiles. 2010. Semi-supervised learning from imperfect data through particle cooperation and competition. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN.2010.5596659

[13] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). 463–469.

[14] H. Cai, V. W. Zheng, and K. C. Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (Sep. 2018), 1616–1637. https://doi.org/10.1109/TKDE.2018.2807452

[15] Canonical Ltd. 2021. Enterprise Open Source and Linux | Ubuntu. https://ubuntu.com.

[16] Canonical Ltd. 2021. Ubuntu Debug Symbol Packages. https://wiki.ubuntu.com/Debug_Symbol_Packages.

[17] G. Chen, Z. Wang, R. Zhang, K. Zhou, S. Huang, K. Ni, Z. Qi, K. Chen, and H. Guan. 2010. A Refined Decompiler to Generate C Code with High Readability. In *2010 17th Working Conference on Reverse Engineering*. 150–154. https://doi.org/10.1109/WCRE.2010.24

[18] Qiming Chen and Ren Wu. 2017. CNN Is All You Need. *CoRR* abs/1712.09662 (2017). http://arxiv.org/abs/1712.09662

[19] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. 785–794. https://doi.org/10.1145/2939672.2939785

[20] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1724–1734. https://doi.org/10.3115/v1/D14-1179

[21] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[22] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 225 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428293

[23] S. H. H. Ding, B. C. M. Fung, and P. Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489.

[24] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 2020 Network and Distributed Systems Security Symposium (NDSS)*.

[25] Eclipse Foundation, Inc. 2021. Eclipse Java development tools. https://www.eclipse.org/jdt.

[26] Eli Bendersky. 2021. pycparser. https://github.com/eliben/pycparser.

[27] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1ersoRqtm

[28] Jerome Friedman. 2000. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29 (11 2000). https://doi.org/10.1214/aos/1013203451

[29] Jerome H. Friedman. 2002. Stochastic Gradient Boosting. *Comput. Stat. Data Anal.* 38, 4 (Feb. 2002), 367–378. https://doi.org/10.1016/S0167-9473(01)00065-2

[30] Edward M Gellenbeck and Curtis R Cook. 1991. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*. Ablex Publishing, Norwood, NJ, 65–81.

[31] GitHub, Inc. 2021. GitHub. https://github.com.

[32] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1631–1640. https://doi.org/10.18653/v1/P16-1154

[33] Han Gao. 2021. Code for Neural Function Rename Engine. https://github.com/USTC-TTCN/NFRE.

[34] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1667–1680. https://doi.org/10.1145/3243734.3243866

[35] Hex-Rays SA. 2021. Hex-Rays Decompiler. https://www.hex-rays.com/products/decompiler.

[36] Hex-Rays SA. 2021. IDA Pro. https://www.hex-rays.com/products/ida.

[37] Intel Corporation. 2021. Intel Advanced Encryption Standard Instructions (AES-NI). https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html.

[38] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling Library Functions in Stripped Binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '11)*. 1–8. https://doi.org/10.1145/2024569.2024571

[39] Alan Jaffe. 2017. Suggesting Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 1050–1052. https://doi.org/10.1145/3106237.3121274

[40] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*. 20–30. https://doi.org/10.1145/3196321.3196330

[41] Yingjiu Li Jiayun Xu and Robert H. Deng. 2021. Differential Training: A Generic Framework to Reduce Label Noises for Android Malware Detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS.*

[42] D. S. Katz, J. Ruchti, and E. Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 346–356. https://doi.org/10.1109/SANER.2018.8330222

[43] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Trans. Knowl. Discov. Data* 6, 4, Article 15 (Dec. 2012), 21 pages. https://doi.org/10.1145/2382577.2382579

[44] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[45] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* https://openreview.net/forum?id=SJU4ayYgl

[46] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proc. ACL.* https://doi.org/10.18653/v1/P17-4012

[47] Taku Kudo. 2018. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, Melbourne, Australia, 66–75. https://doi.org/10.18653/v1/P18-1007

[48] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* 66–71. https://doi.org/10.18653/v1/D18-2012

[49] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 628–639. https://doi.org/10.1109/ASE.2019.00064

[50] 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 Eighteenth International Conference on Machine Learning (ICML 2001),* Carla E. Brodley and Andrea Pohoreckyj Danyluk (Eds.). 282–289.

[51] A. LeClair, S. Jiang, and C. McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 795–806. https://doi.org/10.1109/ICSE.2019.00087

[52] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Association for Computational Linguistics, Minneapolis, Minnesota, 3931–3937. https://doi.org/10.18653/v1/N19-1394

[53] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS.*

[54] Qimai Li, Zhichao Han, and Xiao ming Wu. 2018. Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16098

[55] Zhibo Liu and Shuai Wang. 2020. 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 (ISSTA 2020).* Association for Computing Machinery, 475–487. https://doi.org/10.1145/3395363.3397370

[56] 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.* 1412–1421. https://doi.org/10.18653/v1/D15-1166

[57] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment,* Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). 309–329.

[58] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR,* Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[59] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (Nov. 1995), 39–41. https://doi.org/10.1145/219717.219748

[60] Palo Alto Networks, Inc. 2021. Domain Generation Algorithm (DGA) Detection. https://docs.paloaltonetworks.com/pan-os/9-1/pan-os-admin/threat-prevention/dns-security/domain-generation-algorithm-detection.html.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[62] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14).* 701–710. https://doi.org/10.1145/2623330.2623732

[63] PNF Software, Inc. 2021. IDA Pro. https://www.pnfsoftware.com/.

[64] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* 45–50. http://is.muni.cz/publication/884893/en

[65] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27,* Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). 3104–3112. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30,* I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[67] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 1875–1892. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational

[68] Yanlin Wang and Hui Li. 2021. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *The Thirty-Fifth AAAI Conference on Artificial Intelligence.*

[69] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17).* Association for Computing Machinery, New York, NY, USA, 363–376. https://doi.org/10.1145/3133956.3134018

[70] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.* The Internet Society.

[71] Yaniv David, Uri Alon and Eran Yahav. 2021. The Dataset of Nero. https://doi.org/10.5281/zenodo.4081641.

[72] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net. https://openreview.net/forum?id=Sy8gdB9xx

[73] Hong Zhao, Zhaobin Chang, Guangbin Bao, and Xiangyan Zeng. 2019. Malicious Domain Names Detection Algorithm Based on *N*-Gram. *J. Comput. Networks Commun.* 2019 (2019), 4612474:1–4612474:9. https://doi.org/10.1155/2019/4612474

[74] Lingxiao Zhao and Leman Akoglu. 2020. PairNorm: Tackling Oversmoothing in GNNs. In *International Conference on Learning Representations.* https://openreview.net/forum?id=rkecl1rtwB

[75] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *26th Annual Network and Distributed System Security Symposium, NDSS.*