Bin2Summary: Beyond Function Name Prediction in Stripped Binaries with Functionality-Specific Code Embeddings

ZIRUI SONG, The Chinese University of Hong Kong, Hong Kong
JIONGYI CHEN, National University of Defense Technology, China
KEHUAN ZHANG, The Chinese University of Hong Kong, Hong Kong

Nowadays, closed-source software only with stripped binaries still dominates the ecosystem, which brings obstacles to understanding the functionalities of the software and further conducting the security analysis. With such an urgent need, research has traditionally focused on predicting function names, which can only provide fragmented and abbreviated information about functionality. To advance the state-of-the-art, this paper presents Bin2Summary to automatically summarize the functionality of the function in stripped binaries with natural language sentences. Specifically, the proposed framework includes a functionality-specific code embedding module to facilitate fine-grained similarity detection and an attention-based seq2seq model to generate summaries in natural language. Based on 16 widely-used projects (*e.g.*, `Coreutils`), we have evaluated Bin2Summary with 38,167 functions, which are filtered from 162,406 functions, and all of them have a high-quality comment. Bin2Summary achieves 0.728 in precision and 0.729 in recall on our datasets, and the functionality-specific embedding module can improve the existing assembly language model by up to 109.5% and 109.9% in precision and recall. Meanwhile, the experiments demonstrated that Bin2Summary has outstanding transferability in analyzing the cross-architecture (*i.e.*, in x64 and x86) and cross-environment (*i.e.*, in `Cygwin` and `MSYS2`) binaries. Finally, the case study illustrates how Bin2Summary outperforms the existing works in providing functionality summaries with abundant semantics beyond function names.

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

Additional Key Words and Phrases: Code Summarization, Reverse Engineering, Machine Learning for Program Analysis, Transfer Learning

ACM Reference Format:
Zirui Song, Jiongyi Chen, and Kehuan Zhang. 2024. Bin2Summary: Beyond Function Name Prediction in Stripped Binaries with Functionality-Specific Code Embeddings. *Proc. ACM Softw. Eng.* 1, FSE, Article 3 (July 2024), 23 pages. https://doi.org/10.1145/3643729

1 INTRODUCTION

After decades, closed-source software, which is often released as executable binaries without its source code, still dominates the ecosystem [16]. Considering a piece of software is driven by its code consisting of various functions, recognizing the functionality offered by these functions in stripped binaries could significantly benefit many important applications including program comprehensions [7, 25, 34], software plagiarism detection [32, 62], decompilation [10, 28], software vulnerability identification [15, 51], and malware analysis [18, 33, 50, 58]. However, the existing

Authors' addresses: Zirui Song, The Chinese University of Hong Kong, Hong Kong, Hong Kong, sz019@ie.cuhk.edu.hk; Jiongyi Chen, National University of Defense Technology, Changsha, China, chenjiongyi@nudt.edu.cn; Kehuan Zhang, The Chinese University of Hong Kong, Hong Kong, Hong Kong, khzhang@ie.cuhk.edu.hk.

ACM 2994-970X/2024/7-ART3
https://doi.org/10.1145/3643729

works [12, 16, 21, 27] have primarily focused on predicting function names from stripped binaries, which are not always reliable in representing the functionality of the target function precisely because the function names are often made up of fragmented and overly abstracted abbreviations. Also, much effort has been devoted to generating comments based on the source code of software [1, 23, 26, 29, 31, 36, 63–65] that facilitate code comprehension, even the authentic function names are in presence. As such, it is urgent and necessary to go beyond function names and summarize the functions in stripped binaries.

Unfortunately, there is a knowledge gap between the function name prediction and the functionality summarization. The existing works that predict function names [12, 16, 21, 27] assume the semantic information of functionality is always uniformly distributed in the function, and they rely on the coarse-grained binary code similarity detection or encoding technique where only part of the functionality can be clearly represented and interpreted; therefore, it is non-trivial to comprehensively interpret functionality by fine-grained and functionality-specific similarity detection. At a high level, three major challenges should be addressed. First, it is difficult to conduct fine-grained similarity detection that captures the specific functionality in stripped binaries. Since the functionality represents the main execution behavior of the function, conducting the fine-grained similarity detection requires corresponding each part of the functionality to particular code snippets. Second, even though the semantic information of functionality could be captured in stripped binary, it is still challenging to present the functionality in natural language (*NL*) as a reliable summary instead of fragmented words, and such a summary should be sufficiently professional and accurate. Third, binaries compiled from the same source code could be different if compiled with different configurations (*e.g.*, cross-architectures), which could change related semantics or conceal the actual action flows within a function, making the functionality difficult to recognize.

Our Approach. In this paper, we present Bin2Summary, a learning-based framework that uses functionality-specific code embedding and an attention-based seq2seq model to reliably generate functionality summaries. Specifically, to capture the particular executions that represent the functionality in binaries, we propose a functionality-specific embedding method to capture the functionality-relevant code snippets and facilitate fine-grained code similarity detection. Next, we leverage representation learning techniques to embed the functions that are compiled into binaries and their associated comments, then further train a seq2seq model to handle the variable size of input function embeddings and the variable length of the output functionality summary. In addition, since the attention mechanism in the seq2seq model has the ability to find the relevance between two sequences, we propose an attention-based seq2seq model aiming to map particular semantic information of functionality extracted from the binaries to corresponding words in natural language. Moreover, to address the issue of analyzing the binaries compiled with different configurations, we adopt transfer learning to make Bin2Summary applicable in analyzing the binaries across architectures and environments. Finally, with the well-trained model, Bin2Summary summarizes the functionality as a natural language sentence for the function in stripped binaries.

Based on 16 open-source projects (*e.g.*, `Coreutils`), including these being widely used in recent five years of related works [13, 21, 27, 44, 59], we evaluate Bin2Summary with a total of 38,167 functions, which are filtered from 162,406 functions, and all of them have a high-quality comment. As the first work in binary code summarization, although it has difficulties in collecting samples that are in binaries and have high-quality comments, our dataset is generalizable since the selected projects are in various aspects of functionalities. Our evaluation results show the effectiveness of Bin2Summary by achieving 0.728 in precision and 0.729 in recall. To illustrate how effective the proposed functionality-specific code embedding techniques are, we also conduct an ablation study to show our module can improve the existing assembly language model by up to 109.5% and 109.9%

in precision and recall. Meanwhile, the transferability evaluation demonstrates that with only 30% training samples, the transferred models still perform well (0.710 in precision and 0.712 in recall) in analyzing the cross-architecture and cross-environment binaries. Going beyond such assessment, we also conduct a case study with widely accepted standards developed by CPC [61] to illustrate how Bin2Summary outperforms the state-of-the-art (*e.g.*, Debin, NFRE, and SymLM). As a result, in most cases (69.0%), the generated functionality summary of Bin2Summary is considered more semantic and helpful in facilitating binary code comprehension than the function name.

Contributions. Our paper makes three key contributions.

- We propose a novel system framework, Bin2Summary, to recognize the functionality of a function in stripped binaries, and summarize the functionality into a natural language sentence with abundant semantics instead of a simple function name.
- We present a suite of new techniques, including a new algorithm to capture the functionality-relevant snippets in binary codes, a novel approach to recognize functionality from program comments, and a unique method to transfer the pre-trained model applicable in analyzing the cross-architecture and cross-environment binaries.
- We develop a prototype of Bin2Summary by implementing the above new techniques. The evaluation shows the efficiency, effectiveness, and transferability of Bin2Summary, and further illustrates how it outperforms the state-of-the-art in practical usage scenarios.

Artifacts. We provide the artifacts available at GitHub[1].

Roadmap. The rest of this paper is organized as follows: §2 uses a running example to explain the challenges and insights of this study; §3 presents the overview of our work; §4 introduces the details of Bin2Summary design; §5 is the evaluation of our system; §6 discusses the limitations and future works; §7 reviews the related works, and §8 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Problem Definition

The functionality of a function is determined by the main execution behavior in the function, which is represented as a set of operation sequences in binaries. Hence, the semantics of such operation sequences as a whole constitute the functionality of a function. The basic unit of a function in stripped binaries is the instruction (*i.e.*, an operation). Several instructions often construct a basic block where these instructions are executed in a deterministic order. Typically, a sequence of basic blocks within a function generates a valid output.

Therefore, we denote the functionality of a function f as $\mathcal{F}_f$ and the semantics of every basic block B_i of function f as $\mathcal{M}_{\mathcal{B}}$, where $\mathcal{B} = \{B_1, B_2, ..., B_n\}$. As such, $\mathcal{F}_f = \{\mathcal{M}_{\mathcal{B}}\}$, which means our first step is to extract the semantic information from binaries to represent the functionality $\mathcal{F}_f$. On the other hand, since we aim to summarize the functionality of a function, *i.e.*, $\mathcal{F}_f$, in natural language, we define the ground truth of functionality summary $\mathcal{S}_f$ as a set of words $\mathcal{S}_f = \{w_1, w_2, ..., w_m\}$ ($w_i \subseteq \mathcal{V}$), where $\mathcal{V}$ is the vocabulary consisting of expressions often used by professionals to describe the functionality of a function. Then, we define the research problem of generating the functionality summaries as a multi-class multi-label classification problem. Specifically, we focus on constructing the mapping function $\Gamma(\cdot)$ that maps $\mathcal{M}_{\mathcal{B}}$ to functionality summary $\mathcal{S}_f$:

$$\Gamma(\mathcal{F}_f) = \Gamma(\mathcal{M}_{\{B_1,...,B_n\}}) = \mathcal{S}_f = \{w_1, ..., w_m\} \tag{1}$$

It is worth mentioning that, although the mapping between $\mathcal{M}_{\mathcal{B}}$ and $\mathcal{S}_f$ can be established, any of their individual units like B_i and w_j are not necessarily relevant.

[1]https://github.com/CupCupRay/Bin2Summary

2.2 A Running Example

We use a real-world function as a running example to illustrate the research problem defined above.

2.2.1 Textual Description in natural language. There are typically three professional textual materials that usually describe the function in natural language, *i.e.*, the function name, the comments, and the documentation. And Listing 1 presents an example of those three materials for the function x509_pubkey_decode in openssl. Obviously, among these materials, the comment has the most abundant semantics that introduces the function as it clearly describes the essential behavior and design purpose of the function; while its function name only partially describes the behavior of the function and the documentation focuses on explaining the usage of the function.

```
Function Name: x509_pubkey_decode
Comment: Attempt to decode a public key.
Documentation: Input: dst **ppkey, src *key. Returns 1 on success,
               returns 0 on a decode failure, returns -1 on a fatal error.
```

Listing 1. Descriptions of x509_pubkey_decode

2.2.2 Functions in source code and stripped binaries. The source code and assembly codes of the function x509_pubkey_decode are presented in Figure 1. We show the source code just for better understanding, and all the tools in this paper are used for binary analysis. By adopting a quick source code review, we can find some snippets that correspond to the functionality of this function, and these lines of code are displayed in bold font (line 4, line 9-10, line 15, and line 22) in Figure 1(a). Intuitively, those code snippets are functionality-relevant since they execute the operations that *"decode a public key"*. After compiling the function into binaries and adopting a binary analysis, we can see the control flow of this function as shown in Figure 1(b). In the CFG, some blocks (displayed in the dark background and bold borders) correspond to the functionality-relevant snippets shown in the source code, *i.e.*, these blocks are functionality-relevant at the assembly level. Meanwhile, as we can see in Figure 1(b), it is obvious the assembly instructions in the x64 version and x86 version are significantly different, even if they are compiled from the same source codes.

2.2.3 Our objective. Our ultimate objective is to summarize the functions in the stripped binary file, and the generated summary is supposed to explain the functionality of the function in natural language. To this end, our solution should achieve two sub-objectives. First, it must effectively capture the function's functionality in stripped binaries. For example, when analyzing the function x509_pubkey_decode, it is supposed to generate a set of words $\hat{\mathcal{S}}_f$ that close to the ground truth $\mathcal{S}_f$ and express the main executions (*a.k.a.*, the designed behavior) of the function (*i.e.*, with the meaning of *"decode a public key"*). Second, it also has to generate the semantically equivalent summary of functions sharing the same functionality in stripped binaries regardless of their compilation options. For instance, it has to produce the expression of *"decode a public key"* for function x509_pubkey_decode in binaries compiled under the x64 and the x86 architectures.

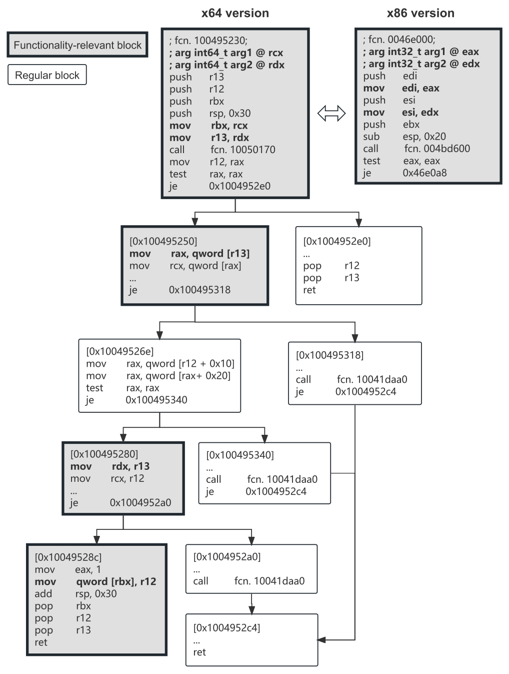
2.3 Challenges and Key Insights

2.3.1 Challenges. At a high level, there are three main challenges in addressing this problem.

Challenge 1*. How to precisely capture functionality-relevant snippets in stripped binaries.* To summarize the stripped binaries, it is non-trivial to understand the functionality offered by binaries and further recognize their features. However, when analyzing the binaries, the existing works try to understand the overall semantics in the whole functions, and none of them considers precisely recognizing and capturing the functionality in particular code snippets. As shown in the running

```
static int x509_pubkey_decode(EVP_PKEY
            **ppkey, X509_PUBKEY *key)
{
  EVP_PKEY *pkey = EVP_PKEY_new();
  if (pkey == NULL) {
    X509err(..., ...);
    return -1; }

  if (!EVP_PKEY_set_type(pkey,
    OBJ_obj2nid(key->algor->algorithm))) {
    X509err(..., ...);
    goto error; }

  if (pkey->ameth->pub_decode) {
    if (!pkey->ameth->pub_decode(pkey, key)) {
      X509err(..., ...);
      goto error; }
  } else {
    X509err(..., ...);
    goto error; }

  *ppkey = pkey;
  return 1;

  error:
    EVP_PKEY_free(pkey);
    return 0; }
```

(a) Source code in C language



```
; fcn. 100495230;
; arg int64_t arg1 @ rcx
; arg int64_t arg2 @ rdx
push    r13
push    r12
push    rbx
push    rsp, 0x30
mov     rbx, rcx
mov     r13, rdx
call    fcn. 10050170
mov     r12, rax
test    rax, rax
je      0x1004952e0
```

```
; fcn. 0046e000;
; arg int32_t arg1 @ eax
; arg int32_t arg2 @ edx
push    edi
mov     edi, eax
push    esi
mov     esi, edx
push    ebx
sub     esp, 0x20
call    fcn. 004bd600
test    eax, eax
je      0x46e0a8
```

```
[0x100495250]
mov     rax, qword [r13]
mov     rcx, qword [rax]
...
je      0x100495318
```

```
[0x1004952e0]
...
pop     r12
pop     r13
ret
```

```
[0x10049526e]
mov     rax, qword [r12 + 0x10]
mov     rax, qword [rax+ 0x20]
test    rax, rax
je      0x100495340
```

```
[0x100495318]
...
call    fcn. 10041daa0
je      0x1004952c4
```

```
[0x100495280]
mov     rdx, r13
mov     rcx, r12
...
je      0x1004952a0
```

```
[0x100495340]
...
call    fcn. 10041daa0
je      0x1004952c4
```

```
[0x10049528c]
mov     eax, 1
mov     qword [rbx], r12
add     rsp, 0x30
pop     rbx
pop     r12
pop     r13
ret
```

```
[0x1004952a0]
...
call    fcn. 10041daa0
```

```
[0x1004952c4]
...
ret
```

(b) Assembly code in x64 and x86 version

Fig. 1. Codes of x509_pubkey_decode, where the functionality-relevant codes/blocks are in bold font.

example (§2.2), we want to predict the keywords such as "decode" and "key" that represent the functionality of function x509_pubkey_decode, which has high relevance with particular code snippets. Therefore, it is necessary to "focus on" the functionality-relevant contents for code analysis.

Challenge 2. *How to automatically summarize reliable functionality summary in natural language sentences.* Even if successfully locating the functionality in binaries, it is also necessary to generate reliable summaries to represent such functionality. Basically, there are two sub-challenges. The first one is how to express functionality in natural language sentences precisely. Generally, a functionality summarization is supposed to provide more detailed information than the abstracted words in function names. As such, the program comment can be used as the reference, which has abundant semantics and good grammatical structure. Then, the second one is how to automate the function summarization process, *i.e.*, the proposed approach is supposed to handle a considerable amount of functions in stripped binaries and predict their functionality without manual assistance.

Challenge 3. *How to deal with the binaries compiled under different architectures or environments.* As shown in Figure 1, we can observe that it could result in dissimilarities if the same source code is compiled with different configurations. Basically, it is a long-standing challenge in binary similarity detection in the cross-architecture or cross-environment (*e.g.*, with different optimization levels, or with requiring an extra emulation layer) manner, which is yet to be well addressed. As such, it is also a definite challenge to address this issue to achieve our ultimate objective.

2.3.2 *Prior Efforts.*

Although BIN2SUMMARY is the first work focused on binary code summarization, binary analysis and source code summarization are not new research problems. However, as shown

Table 1. Comparison with related works in addressing challenges and the metrics reported in their paper

	CodeNN [24]	Code Attention [65]	DeepCom [23]	CodeRNN [31]	Astattgru [29]	Rencos [64]	NCS [1]	T5-Code [36]	CoditT5 [63]	Trex [45]	Stateformer [44]	Debin [21]	Nero [12]	NFRE [16]	SymLM [27]	Our Work
C1	✗	✓✗	✗	✗	✓✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
C2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓✗	✓✗	✓✗	✓✗	✓
C3	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓✗	✗	✓✗	✓✗	✓✗	✓
Analysis Object[2]	S	S	S	S	S	S	S	S	S	**B**	**B**	**B**	**B**	**B**	**B**	**B**
Precision (%)	/	/	/	/	/	/	/	/	/	/	/	29.5	40.2	39.4	63.4	**72.8**
Recall (%)	/	/	/	/	/	/	/	/	/	/	/	29.3	39.9	34.2	67.7	**72.9**
BLEU-4 (%)	20.5	24.6	38.2	/	11.4	20.7	44.6	17.3	66.8	/	/	/	/	/	/	**41.4**

[2] *i.e.*, the target of the work, "S" indicates it focuses on the source code analysis, "B" indicates it focuses on the binary code analysis.

in Table 1, none of the existing works comprehensively considers the three challenges mentioned above, which directly motivates us to propose Bin2Summary.

Specifically, among the previous works, only Code Attention [65] and Astattgru [29] consider extracting features from specific code snippets (*a.k.a.*, **C1**), but they didn't treat it systematically nor apply such an idea in the binary analysis. Also, only the works in source code analysis [1, 23, 29, 31, 36, 63–65] have effectively addressed the **C2**, and none of the binary analysis works [12, 16, 21, 27, 44, 45] can generate natural language sentences. Moreover, when facing the binaries compiled under different settings, most works (*e.g.*, Nero [12], NFRE [16], SymLM [27]) need to train a new model and perform evaluation individually, which only partially solve the **C3**. Although Trex [45] can deal with such binaries, it is not designed for code summarization or information recovery.

2.3.3 Insights. Fortunately, we have observed the following insights to address the challenges.

Insight 1*. Identifying the functionality-relevant code snippets by function slicing using function arguments.* We observe that the behavior of a function is *usually* related to the computation and transformation of arguments, and the code not associated with the data flow of arguments is *often* less relevant to the main execution behavior. For example, in the running example (shown in Figure 1), by analyzing the data flow of function arguments **key* and ***ppkey*, the functionality-relevant snippets (*e.g.*, the codes in bold font) can be clearly recognized. Based on this insight, we propose to perform a binary-level static function slicing using function arguments to identify the functionality-relevant snippets for each function. In particular, the above insight does not apply to the function without argument; in this case, we will treat all its blocks equally.

Insight 2*. Summarizing function via attention-based machine learning with the help of NLP expertise.* We would like to apply machine learning expertise to automatically generate the reliable functionality summary of a function in stripped binaries. To this end, we must build a ground truth dataset to learn the mapping from the binary codes to the functionality descriptions in natural language. As such, we utilize NLP expertise to obtain the functionality descriptions and attention network to find the relevance between semantics in binaries and descriptions in natural language.

Insight 3*. Addressing the binary similarity analysis issue across compilation settings with transfer learning.* Since we leverage machine learning expertise and build a neural network model to interpret functionality, the pre-trained model could provide abundant prior knowledge for the original task. Considering that we may not always be able to obtain a sufficiently large dataset

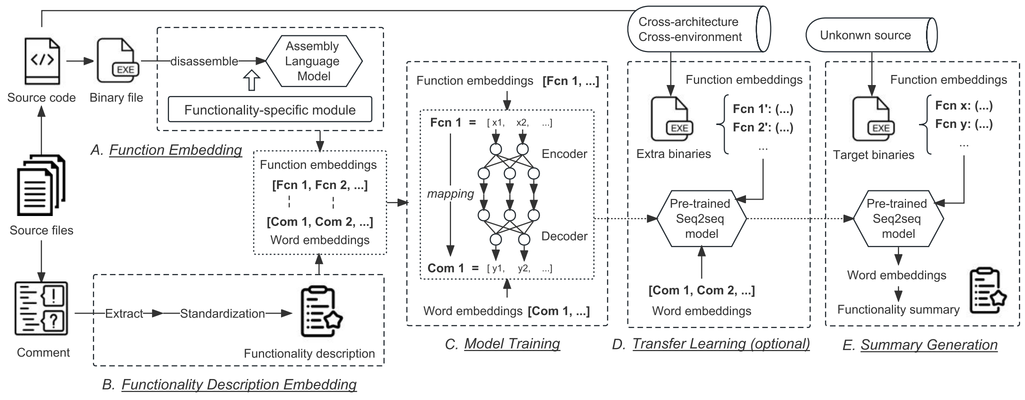


Fig. 2. Overview of Bin2Summary

with files in target format (*e.g.*, binaries compiled with different settings), we choose to investigate a transfer learning approach. Such an approach can solve the domain shift problem of adapting models to a new task (*i.e.*, analyzing the binaries across architecture and environment). Meanwhile, with abundant prior knowledge in the pre-trained model, we can use limited labeled samples (*e.g.*, 30% training samples) to achieve satisfactory performance.

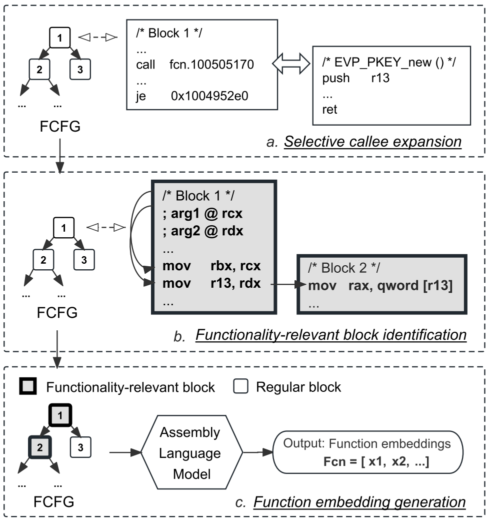
3 OVERVIEW OF BIN2SUMMARY

Figure 2 presents the overview of Bin2Summary, which consists of three stages: offline data processing, offline model training, and online testing. There are four main components and an optional component in different stages. Specifically, at the stage of offline data processing, Bin2Summary first takes a set of open-sourced software as input and then sends the compiled binary code of each software to *Function Embedding* component, which embedding every function into $\mathcal{M}_{\mathcal{B}}$, and also passes the associated program comments to *Functionality Description Embedding* component that extracts and embedding the functionality descriptions. Next, at the stage of offline model training, all pairs of embedded functions and their associated embedded descriptions are collected by *Model Training* component, which then leverages these pairs to train a seq2seq model and learn the semantic mapping $\Gamma(\cdot)$. Additionally, there is an optional component *Transfer Learning* at the stage of offline model training, which leverages the pairs of embedded functions with different compilation configurations and their associated embedded descriptions to re-train a seq2seq model and learn a new mapping $\Gamma'(\cdot)$. Finally, at the online testing, the pre-trained model is used by *Summary Generation* that takes a target binary, embedding its functions in the same way as *Function Embedding*, and generates the functionality summary $\mathcal{S}_f$.

4 DESIGN DETAILS

4.1 Function Embedding

Since our key objective is to predict the functionality of an unknown function by comparing its functionality similarity with known functions at the binary code level, we have to propose a mechanism to extract the functionality from a function. As discussed in §2.3(a), Bin2Summary adopts the essential idea in the representation learning to represent each function by embedding it as a vector preserving its functionality semantics. To this end, Bin2Summary designs the first component, and its workflow is presented in Figure 3.



(a) Workflow of *Function Embedding*

```
Algorithm 1: Identify Functionality-relevant Block
Input: function control-flow graph $G(V, E)$
       function entry arguments $Args$
Output: list of functionality-relevant blocks $\mathbb{V} \subseteq V$
1 Initialize register states $S_c$ represented by $Args$
2 for $v_i \in V$ do
3     Update current states $S_c$ with $(G, v_i)$
4     for $instruction \in v_i$ do
5         Get next states $S_n$ with $(S_c, instruction)$
6         for $arg \in Args$ do
7             if $arg \in changes\ (S_c, S_n)$ then
8                 Append $v_i$ to $\mathbb{V}$
              end
          end
9         $S_c = S_n$
      end
  end
```

(b) Functionality-relevant block identification alg.

Fig. 3. Workflow of *Function Embedding* and algorithm for functionality-relevant block identification.

4.1.1 *FCFG Generation and Selective Callee Expansion.*

CFG is known as effective and reliable in differentiating binaries [3, 13, 14, 30, 60], with the help of disassembly tools (*e.g.*, Radare [52], IDA Pro [49], Binary Ninja [8]), *Function Embedding* first would like to generate an inter-procedure control flow graph (*ICFG*) for the program and obtain FCFG for each function to recognize and capture their functionality semantics. Since FCFG is derived from ICFG, the calling context is retained. Having generated FCFG for each function, *Function Embedding* then considers the function call context by adopting selective callee expansion, which is a technique used to extend the original function and enrich its semantics. BinGo [9] and Asm2vec [13] propose this technique to selectively inline the callee function into the caller function for static analysis. BinGo and Asm2vec both set a coupling metric to capture the ratio of in-degree and out-degree of each callee function, *i.e.*, only consider the utility function for expansion. But *Function Embedding* does not apply such a strategy, since it may mistakenly filter out the functionality-relevant information. For instance, the function `EVP_PKEY_new`[3] (*i.e.*, the callee function shown in Figure 1) is supposed to be ignored according to the coupling metric proposed by BinGo, however, it provides the semantics of "new" and "key", which are essential in enriching the caller's semantics in terms of functionality that related to "key".

Also, same as Asm2vec, *Function Embedding* only expands the first-order callees in the call graph to avoid too many callees occupying the body of the caller. Meanwhile, *Function Embedding* adopts the same equation that was proposed in Asm2vec to filter out lengthy callee:

$$\delta(f_{er}, f_{ee}) = \frac{\text{length}(f_{ee})}{\text{length}(f_{er})} \tag{2}$$

where f_{er} is the caller function, and f_{ee} is the callee function. The high δ means the length of the callee function is increasingly comparable to the caller. According to the algorithm, such a callee should be discarded when δ is higher than a threshold value.

[3] *i.e.*, `fcn.100505170`, as its name is stripped in stripped binaries.

4.1.2 Functionality-relevant Block Identification. After obtaining the FCFG with callee expansion, *Function Embedding* needs to identify the blocks which are functionality-relevant. As discussed in §2.3, we propose to perform a binary-level static function slicing using function arguments to identify the functionality-relevant blocks for each function, first of all, we define the functionality-relevant block as follows:

Definition 1. ***Functionality-relevant Block*** *refers to a block that (1) is in a function with no arguments or (2) contains at least one operation that directly or indirectly involves any arguments.*

Specifically, the operations that directly involve a function argument including reading and writing the data in the argument; and the operations that indirectly involve a function argument including reading and writing the data which can be traced back to the argument. In particular, Bin2Summary first needs to recover the information of the function argument from stripped binaries. Fortunately, the disassemblers (*i.e.*, Radare [52], Binary Ninja [8]) that are used in Bin2Summary can recover the register information of function arguments (as demonstrated in Figure 1 and Figure 3).

With the register information of function arguments, Bin2Summary next needs to trace the data flow dependence of function arguments and slice the instructions that directly or indirectly involve one or more function arguments. As shown in Figure 3(b), Bin2Summary first initializes the current register states S_c to make it represented by the function arguments, and backtrack values of all registers in a basic block and stores the results in S_c, then for each instruction in the block, Bin2Summary obtains the S_n (*i.e.*, the next register states). Finally, the algorithm will find if there are any changes related to registers that involve the function arguments. Once there exists a path indicating a function argument is involved, the block will be marked as functionality-relevant. To demonstrate how identification works, we illustrate three real-world cases as follows.

Case 1 (no argument): Function `EVP_PKEY_new` (the callee in the running example (§2.2) has no function argument, hence we treat all its blocks as functionality-relevant blocks.

Case 2 (directly read): For the first block of function `x509_pubkey_decode` (as shown in Figure 1), there are two instructions: *mov rbx, rcx* and *mov r13, rdx* that execute the operations that pass the value of the function argument to another register (*a.k.a.*, read the arguments). Hence this block is identified as functionality-relevant block.

Case 3 (indirectly write): For the block *0x10049528c* in function `x509_pubkey_decode` (as shown in Figure 1), there is an instruction: *mov qword [rbx], r12* that executes an operation that passes a parameter to the contents pointed by *rbx* (which stores the value of function argument ***ppkey*). Hence this block is identified as functionality-relevant block.

As a result, the identified functionality-relevant blocks perfectly match the observations from the source code reviewing, *i.e.*, the functionality-relevant blocks well correspond to the functionality-relevant code snippets shown in Figure 1.

4.1.3 Function Embedding Generation. After identifying the functionality-relevant blocks, *Function Embedding* starts to generate function embeddings by using representation learning techniques. Specifically, *Function Embedding* first pre-process the FCFG, then trains an assembly language model for instruction embedding and further FCFG embedding.

(*I*) Function Control-flow Graph Pre-processing. Since the FCFG has both semantic and structural information, the first step of FCFG pre-processing is serializing the graphs of these functions to convert the structural information of control flow into sequential information. Similar to the essential idea of Word2vec [39] to represent the co-occurrence of words obtained from sequences of words in paragraphs of natural language texts, the serialization technique in FCFG pre-processing treats each execution path as a sequence of words, and each path should contain the co-occurrence of its composed basic blocks. For example, the random walk sampling [46] can be used as the serialization technique in FCFG pre-processing, which is also widely adopted in the existing binary code representation learning works [13, 14]. After the FCFG serialization, *Function Embedding* then normalizes the extracted execution paths to eliminate a variety of differences of binaries resulting from different compilation choices. For example, the specific address value in the assembly instructions will be replaced by the token "addr". The normalization is expected to enhance the generality of the assembly language model, further facilitate *Model Training* and improve the performance of *Summary Generation*.

(*II*) Function Control-flow Graph Embedding. After serializing and normalizing the FCFG, *Function Embedding* next trains an assembly language model for FCFG embedding. Similar to Word2vec [39] where the first step is embedding the basic unit (*i.e.*, word in natural language text) as a numeric vector representing its co-occurrence with other basic units, the assembly language model starts from embedding the opcodes and operands, which are the basic unit constructing an instruction. The existing binary code representation learning works have adopted kinds of advanced models. For instance, Asm2vec [13] adopts the PV-DM model, DeepBinDiff [14] adopts the Word2vec model, and PalmTree [30] adopts the BERT model. Bin2Summary is able to utilize any of them as a plugin module in *Function Embedding* and further generate functionality-specific embeddings for functions (the details of re-implementation and evaluation are pushed to §5.3).

With the pre-trained assembly language model, *Function Embedding* then embeds each instruction, which consists of an opcode and potential multiple operands, by concatenating overall operands embedding with the opcode embedding instead of simply summing them up because the opcodes and operands have different semantics. Then, depending on whether the block is functionality-relevant, *Function Embedding* calculates the block embedding at different granularities. Specifically, for a basic block B_i, its semantics (which is represented as an embedding) are calculated as

$$\mathcal{M}_{B_i} = \begin{cases} [e_1, e_2, e_3, ..., e_k] & if \quad B_i \in \mathcal{B}_F \\ \sum_{j=1}^{k} e_j & if \quad B_i \notin \mathcal{B}_F \end{cases} \tag{3}$$

Where e_j indicates the embedding of j_{th} instruction in block B_i, k means the total instruction number in block B_i, and $\mathcal{B}_F$ is the list of functionality-relevant blocks. As a result, if B_i is functionality-relevant, the embeddings $\mathcal{M}_{B_i}$ is supposed to be a matrix of instruction embeddings; if B_i is a regular block, the embeddings $\mathcal{M}_{B_i}$ is supposed to be a vector calculated by summing up its instruction embeddings. As a result, *Function Embedding* embeds a FCFG as a set of basic blocks to represent its executions as $\mathcal{F}_f = \{\mathcal{M}_{\mathcal{B}}\}$.

4.2 Functionality Description Embedding

Having embedded functions, Bin2Summary next extracts the functionality descriptions from their associated comments and embeds these functionality descriptions to assist in constructing the mapping between the binary codes and the words in natural language. To this end, as the second component of Bin2Summary, *Functionality Description Embedding* is proposed and its workflow is illustrated in Figure 4.

4.2.1 Functionality Description Extraction. Since program comments may contain not only the functionality description of our interest but also other information, it is necessary to recognize and extract the words that express the main function behavior from comments. Fortunately, we have noticed two unique properties among pieces of functionality description, *i.e.*, *(i)* functionality description often contains a meaningful predicate (normally as a verb) and *(ii)* sometimes the conditional clause also provides abundant semantics. As such, *Functionality Description Embedding* applies two strategies to extract the functionality description. Specifically, it first extracts the

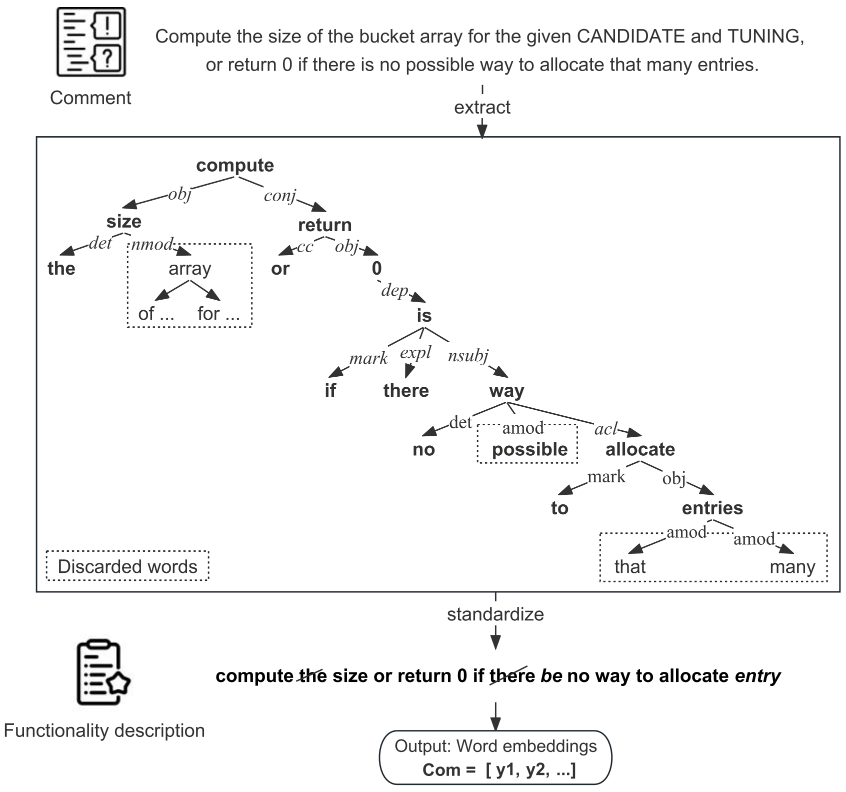

Comment
Compute the size of the bucket array for the given CANDIDATE and TUNING, or return 0 if there is no possible way to allocate that many entries.
extract
compute
obj
conj
size
return
det
nmod
cc
obj
the
array
or
0
of ...
for ...
dep
is
mark
expl
nsubj
if
there
way
det
amod
acl
no
possible
allocate
mark
obj
to
entries
amod
amod
Discarded words
that
many
standardize
Functionality description
compute ~~the~~ size or return 0 if ~~there~~ be no way to allocate entry
Output: Word embeddings
Com = [y1, y2, ...]

Fig. 4. Workflow of *Functionality Description Embedding*

main clause of a sentence if its predicate is meaningful, then only keeps conditional clauses but ignores other types of clauses within the sentence. To implement proposed strategies, *Functionality Description Embedding* utilizes the Stanford CoreNLP [35] to conduct natural language grammatical analysis and semantic analysis. Specifically, dependencies parsing is used to analyze grammatical relations between the words and extract the syntactic structure of the sentence, and Part of Speech (*POS*) tagging can help to find the lexical terms of the words.

For example, as presented in Figure 4, since there are two meaningful verbs (*i.e.*, "compute" and "return") in the main clause and the conditional clause also contains a meaningful verb (*i.e.*, "allocate"), we use such verbs as the root and extract corresponding pieces while keeping the main structure of these clauses. Particularly, we will discard a function of our dataset if its comment does not have a main clause or any meaningful predicate, since its comment fails to express the function's execution behavior.

4.2.2 Functionality Description Standardization. Having extracted functionality description, *Functionality Description Embedding* next standardizes these texts due to the varied commenting styles in different projects, which may introduce the label sparsity problem reducing the quality of the training data. Specifically, *Functionality Description Embedding* applies the lemmatization technique [35] in Stanford CoreNLP [35], which identifies and returns the base form of a word (*e.g.*, compute) if given its families of derivational words sharing similar meanings (*e.g.*, computes, computed, and computing), to address the label sparsity problem. In addition, to further reduce the noise, *Functionality Description Embedding* also filters some function words with little semantic meaning (*e.g.*, definite and indefinite articles). For example, as shown in Figure 4, the extract sentence is standardized as "compute size or return 0 if be no way to allocate entry", and such sentence can fully express the functionality with as few words as possible.

4.2.3 Description Sentence Embedding. Similar to *Function Embedding*, the final step of *Functionality Description Embedding* is embedding the standardized functionality description. Considering the extensive vocabulary and the semantic requirement which makes other word representation techniques (*e.g.*, one-hot encoding) less effective and efficient, *Functionality Description Embedding* chooses Word2vec [39] and follows its standard procedure to embed each word, and generates a sequence of word embeddings representing the description.

4.3 Model Training

After embedding both functions and their functionality description, Bin2Summary will construct the mapping between these two pieces of information. Considering the length of its input and its output may not be in a fixed size, Bin2Summary would like to train a seq2seq model to construct the mapping. In addition, given the insight that a piece of functionality description always corresponds to a specific code snippet, *Model Training* applies the attention mechanism to improve the model.

4.3.1 Constructing Encoder and Decoder. Considering a seq2seq model is trained through an encoder and a decoder, in this work, the input of the encoder is the function embeddings that are produced by *Function Embedding* (§4.1), which can be represented as $\mathcal{M}_{\mathcal{B}}$. As for the decoder, its input is the output of the encoder alongside the word embeddings generated by *Functionality Description Embedding* (§4.2), which can be represented as $\mathcal{S}_f$. In addition, the encoder can also be used to learn the transferred knowledge in *Transfer Learning*. To this end, our seq2seq model is built on gate recurrent unit (*GRU*), which uses its internal state to process the input sequence, and can also avoid the legacy problem in traditional RNN.

4.3.2 Training with Attention. Recall a key observation (§2.3) that a certain functionality description of a function often maps a specific snippet of code, we can leverage the attention mechanism in the model training to make it focus on the local features (*i.e.*, the structure and semantics of specific snippets) to improve its performance. Particularly, *Model Training* applies the Bahdanau attention [6] to calculate the attention alignment score as an adaptive weight dynamically highlighting the relevant features of the function embeddings, which is necessary to find the relevance between functionality semantics in binaries and functionality descriptions in natural language. Finally, the whole training process is designed to construct the mapping function $\Gamma(\cdot)$ that maps the $\mathcal{M}_{\mathcal{B}}$ to summary $\{w_1, w_2, ..., w_m\}$ and minimizes the loss:

$$L = \sum_{\hat{w}_i} D(e_{w_i}, e_{\hat{w}_i}) \tag{4}$$

where $e_{\hat{w}_i}$ is the predicted word embedding, e_{w_i} is the embedding of reference word (*a.k.a.*, the ground truth), and $D(\cdot)$ is the function calculates the distance of two embeddings.

4.4 Transfer Learning

As demonstrated in Figure 1, the binaries as the analysis targets may vary significantly regarding the compilation settings (*e.g.*, different environments, and architectures), leading to poor performance of the prediction model trained with a particular type of binaries. When facing the binaries compiled under a different architecture or environment, all the existing works [12, 16, 21, 27] choose to train a new model from scratch, which not only wastes the prior knowledge in the pre-trained model but also requires massive extra training data. Unlike the existing works, we propose to use transfer learning and apply Bin2Summary in analyzing the binaries compiled with different settings.

Therefore, we propose a *Transfer Learning* component to help construct a new mapping function $\Gamma'(\cdot)$ using the knowledge in $\Gamma(\cdot)$. Specifically, we need to learn a new mapping $\Gamma'(\cdot)$ that maps $\mathcal{F}_{f'} = \{\mathcal{M}_{\mathcal{B}'}\}$ to functionality summary $\mathcal{S}_{f'}$ using the pre-trained seq2seq model from *Model Training*, where $\mathcal{S}_{f'}$ is the the same as $\mathcal{S}_f$, while the semantics $\mathcal{M}_{\mathcal{B}'}$ are different from $\mathcal{M}_{\mathcal{B}}$. During the *Transfer Learning*, the design of the pre-trained seq2seq model will not be changed. Specifically, it remains an encoder-decoder structure, and only the parameters within the network unit will be adjusted. For example, by freezing the weights of the encoder's higher layers and all the layers in the decoder, encoder's lower layers in our pre-trained model can be fine-tuned, which is also a widely applied method in related transfer learning works [17, 22, 40, 48]. Also, the *Transfer Learning* is optional as it is used when there is a need to apply Bin2Summary for cross-architecture or cross-environment binary analysis.

4.5 Summary Generation

After obtaining the pre-trained seq2seq model in *Model Training*, Bin2Summary can then take an arbitrary binary file as input and generate the functionality summary of each function. Specifically, *Summary Generation* first embeds each function in the given binary with the same method in *Function Embedding* (§4.1), and then generates the embedded functionality summary $\hat{\mathcal{S}}$ as a set of embedded words $\{\hat{e}_{w_1}, \hat{e}_{w_2}, ...\}$ for each inputted function embedding. Finally, it interprets the generated embedded functionality summary from numeric vectors to human readable natural languages based on the Word2vec model trained in *Functionality Description Embedding* (§4.5). Specifically, for any predicted embedded words $\hat{e}_{w_i}$, we can retrieve the closest word in the vocabulary using the Word2vec model by computing the similarity between the target and candidates.

5 EVALUATION

In this section, we first introduce the details of our dataset and our experiment environment (§5.1), then evaluate the efficiency (§5.2), effectiveness (§5.3), and transferability (§5.4) of Bin2Summary. Meanwhile, we provide a user study (§5.5) to illustrate how Bin2Summary outperforms the existing function name prediction works.

5.1 Evaluation Setup

Dataset. We select a total of 16 projects for the experiments, which are widely used in recent five years of research works [13, 21, 27, 44, 59] that are related to the software and program analysis. Specifically, the collected projects are relevant to utilities (`Attr`, `Bash`, `Binutils`, `Coreutils`, `Diffutils`, `Findutils`, `Make`, `Sg3_utils`, `Tar`), database (`Sqlite3`), and networking (`Httpd`, `Openssh`, `Openssl`, `Putty`, `Tmux`, `Wget`). Then the stripped binaries in our main dataset are obtained based on `Cygwin` [11]. Among various environments, we choose 64-bit `Cygwin` because it can provide more well-maintained program comments along with the source codes than other environments (*e.g.*, `MSYS2`). In addition, the binaries in other formats are also prepared in further experiments (*e.g.*, §5.5). Next, we clean the data to improve the quality of datasets. In total, there are 581 binary files. Since the comments are necessary as the raw reference, we filter the functions with meaningless comments or without comment, then we collect 38,167 functions as the starting point, which are filtered from 162,406 functions, and all of them have a high-quality comment. Finally, as in the previous function name prediction works [12, 21, 27], we split the dataset into training, validation, and test sets with a ratio of 8:1:1 in terms of the binary files, which guarantees there is no overlapping at the file level. More statistics related to our datasets are presented in the supplementary materials.

Implementation. Bin2Summary was implemented with more than 3,000 lines of Python code. In the prototype of Bin2Summary, we re-implement Asm2vec , DeepBinDiff , and PalmTree with more than 500 lines of our own code to utilize their encoding technique and adopt their assembly language model as a module in *Function Embedding*. Meanwhile, the hyper-parameter configurations are engineered based on empirical evaluation by ourselves and in the literature [12, 16, 24, 65], more details are also shown in the supplementary material and included in the artifact.

Evaluation Environment. All the evaluations are performed on a server running Ubuntu 16.04, equipped with a 32-core Inter E5-2640 processor, 256 GB physical memory, 64 GB RAM, and one GeForce GTX TITAN X GPU.

5.2 Efficiency Evaluation

Even if it is ordinarily insensitive to time consumption for offline analysis, Bin2Summary can perform efficiently. For *Function Embedding*, Bin2Summary takes 5.41 seconds to identify the functionality-relevant blocks with data-flow dependency analysis for each function on average. In the part of embedding generation, the time consumption for assembly language model (*e.g.*, PV-DM model from Asm2vec, Word2vec model from DeepBinDiff, BERT from PalmTree) offline training varies from 25 to 60 hours, and the embedding prediction takes 0.07 seconds for each function on average. For *Functionality Description Embedding*, it takes an average of 0.04 seconds to extract and embed the functionality description of each function. In addition, Bin2Summary needs around 60 to 70 hours to train a seq2seq model until loss converges. Meanwhile, it takes about 0.15 seconds to generate a functionality summary in *Summary Generation*.

Comparing to more than 90 seconds per function by Debin [21] and Nero [12] which perform heavyweight global data-flow analysis, and around 4 seconds per function by NFRE [16] to reassign a function name, which is shorter in length and quantitatively more minor input and output in model training and complexity, Bin2Summary outperforms them in terms of efficiency and shows the top-tier capability in handling large-scale binaries.

5.3 Effectiveness Evaluation

We evaluate the accuracy of Bin2Summary in predicting the functionality summaries with *precision* and *recall* which comply with related works [4, 5, 12, 16, 27]. Meanwhile, we also adopt the *BLEU score* to measure the quality of the generated summaries which comply with related works in source code summarization [1, 23, 24, 29, 36, 63–65].

Precision and Recall. Given a generated sequence $\hat{S} : \{\hat{w}_1, ..., \hat{w}_i\}$ (*i.e.*, the predicted functionality) and the reference sequence $S : \{w_1, ..., w_j\}$ (*i.e.*, the collected functionality description), we have:

$$\mathrm{TP} = \sum_{k=1}^{i} \mathbb{I}\{\hat{w}_k \in S\}, \mathrm{FP} = \sum_{k=1}^{i} \mathbb{I}\{\hat{w}_k \notin S\}, \mathrm{FN} = \sum_{k=1}^{j} \mathbb{I}\{w_k \notin \hat{S}\} \tag{5}$$

$$\text{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \text{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{6}$$

where $\hat{w}_k$ indicates an individual word in the generated sequence, w_k indicates an individual word in the reference sequence, and $\mathbb{I}\{\cdot\}$ is indicator function, namely $\mathbb{I}\{True\} = 1$ and $\mathbb{I}\{False\} = 0$.

BLEU Score. BLEU socre [43] is a commonly used automatic metric for NMT and has been used in the evaluation of machine translation tasks [1, 23, 24, 26, 29, 54, 64]. Generally, the BLEU-*n* indicates an algorithm has applied a maximum of n-gram precision. It calculates the similarity between the candidate and the reference. A higher BLEU score indicates higher relevance between the candidate and the reference. The score is computed as follows:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} \frac{1}{N} \log P_n) \tag{7}$$

where P_n is the ratio of length n sub-sequences in the generated sequence that are also in the reference. Assume that we apply the BLEU-4, then N should be set as 4, which is the maximum value of grams. BP is the brevity penalty, which is computed as:

$$BP = \begin{cases} 1 & if \quad l_c > l_s \\ e^{1-\frac{l_s}{l_c}} & if \quad l_c \leq l_s \end{cases} \tag{8}$$

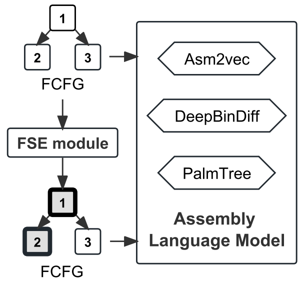
where l_c is the length of the generated sequence and l_s is the effective length of the reference sequence. The brevity penalty is used to ensure the high-scoring candidate matches the reference in length, word choice, and word order. In our experiments, the BLEU score calculation algorithm is implemented using smoothing method$_0$ based on NLTK$_{3.2.4}$, which is widely used [64].

Experiments. As introduced in *Function Embedding* (§4.1), our approach can adopt any one of the existing binary code representation learning techniques as a plugin module to train an assembly language model and further generate the function embedding. In this section, we would like to evaluate the performance of Bin2Summary with different function embedding modules and illustrate how different function embedding techniques contribute to the functionality summary generation tasks. To this end, we re-implement Asm2vec [13], DeepBinDiff [14], and PalmTree [30] and utilize their assembly language model to generate different function embeddings for further *Model Training* and *Summary Generation*. For a fair comparison, we set the embedding dimension as 128 for each model, which is also the hyper-parameter used in these works, and we did not set any limitation on the vocabulary size of these assembly language models. Meanwhile, to evaluate the contributions of the "functionality-specific embedding (short as FSE) module" in *Function Embedding*, we also compare the performance of Bin2Summary with and without it (as shown in Figure 5). There are two configurations for each adopted technique:

- **-Base**: only uses basic function embedding technique
- **-FSE**: uses basic function embedding technique together with FSE module

In addition, to assess Bin2Summary's performance in generating natural language sentences, we utilized two sets of references to calculate the BLEU score: the "processed reference" represents the pre-processed description after the extraction and standardization (*e.g.*, the functionality description shown in Figure 4); and the "raw reference" represents the raw comment collected from source code (*e.g.*, the comment shown in Figure 4). Furthermore, in order to eliminate the randomness in model training, we have trained ten models (with different initial states) for each version of Bin2Summary with different techniques and modules.

Results. Figure 5 shows the performance of Bin2Summary models. We can observe that (1) among different models, the model adopting PalmTree module achieves the best precision (0.659 with base technique and 0.728 with FSE module on average), best recall (0.660 with base technique and 0.729 with FSE module on average) and best BLEU score (0.361 with base technique and 0.414 with FSE module on average) for processed reference, it is because the function embeddings generated by PalmTree contain more information (*e.g.*, internal formats, contextual dependency) than the embeddings generated by other techniques. Also, (2) the FSE module can significantly improve the performance of the Bin2Summary models. Specifically, regarding the precision and recall, the FSE module improves the model adopting Asm2vec by 76.4% and 76.3%; improves the model



(a) Function embedding with different techniques and modules

Modules(s)	Prec. (%)	Rec. (%)	BLEU-4 (%) Processed Ref.	BLEU-4 (%) Raw Ref.
Asm2vec-Base	21.2 ± 4.2	21.5 ± 4.3	6.9 ± 2.6	0.1 ± 0.0
Asm2vec-FSE	**37.4 ± 3.9**	**37.9 ± 4.1**	**17.5 ± 3.7**	**4.1 ± 0.8**
DeepBinDiff-Base	32.5 ± 4.3	32.3 ± 4.2	13.6 ± 3.8	2.7 ± 0.6
DeepBinDiff-FSE	**68.1** ± **2.2**	**67.8** ± 2.1	**37.2 ± 4.1**	**13.0 ± 2.0**
PalmTree-Base	65.9 ± 2.1	66.0 ± 2.2	36.1 ± 5.1	12.7 ± 2.0
PalmTree-FSE	**72.8 ± 1.3**	**72.9 ± 1.2**	**41.4 ± 5.7**	**14.2 ± 2.1**

(b) Performance of Bin2Summary with different techniques, modules and different references on different metrics

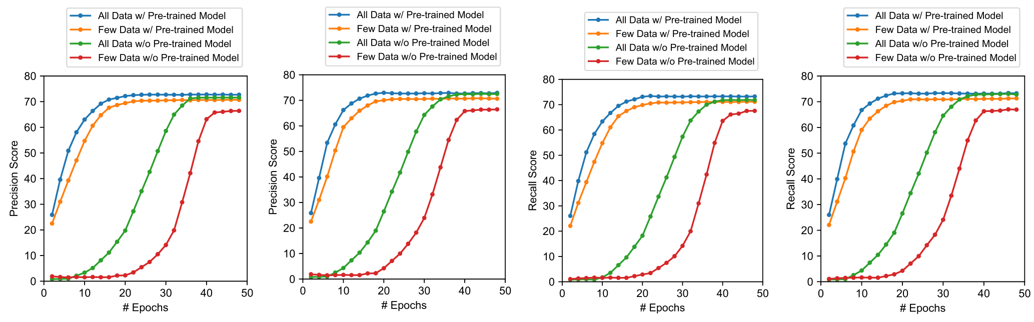
Fig. 5. Effectiveness evaluation strategy and performance of Bin2Summary.

adopting DeepBinDiff by 109.5% and 109.9%; improves the model adopting PalmTree by 10.5% and 10.5%. From the perspective of the BLEU score, the FSE module is also effective in improving sentence generation, *i.e.*, it improves the model adopting Asm2vec by 153.6% and 400.0% in terms of processed reference and raw reference; improves the model adopting DeepBinDiff by 173.5% and 381.5% in terms of processed reference and raw reference; improves the model adopting PalmTree by 14.7% and 11.8% in terms of processed reference and raw reference.

Therefore, we can conclude that (1) the FSE module is effective in improving existing function embedding techniques and particularly useful for the lightweight ones (*e.g.*, Asm2vec and DeepBinDiff), which proves the functionality-relevant snippets are critical in recognizing the execution behavior. (2) Although the FSE module is extremely effective in improving the existing function embedding modules, their performance is still constrained by their own design framework, for instance, the prototype with "PalmTree-Base" is still better than the prototype with "Asm2vec-FSE" since PalmTree is more heavyweight than Asm2vec. Furthermore, to fully evaluate Bin2Summary, in the following experiments, we will use our best model which adopts the "PalmTree-FSE" module as the prototype.

5.4 Transferability Evaluation

To reveal the transferability of Bin2Summary, we examine whether the pre-trained model in Bin2Summary can be transferred to analyze the cross-architecture and cross-environment binaries even with 30% training samples. The current prototype of Bin2Summary is trained on the dataset from `Cygwin64`. Therefore, to transfer the pre-trained model in new tasks and evaluate its performance on different binaries, we first build two evaluation datasets containing the binaries compiled under `Cygwin32` (*i.e.*, cross-architecture) and `64-bit MSYS2` (*i.e.*, cross-environment), where the `Cygwin32` is the 32-bit version of `Cygwin64` while remaining the same toolchain and corresponding configurations; and the `64-bit MSYS2` is another 64-bit run-time environment running natively on Windows OS, whose compilation environment (*e.g.*, without requiring a POSIX emulation layer) are significantly different from the ones in `Cygwin`. We guarantee that the collected binaries are compiled from the same source code. Next, we try to feed only 30% training samples for Bin2Summary to conduct *Transfer Learning*. Specifically, since the encoder in our seq2seq model is naturally responsible for generating the context vector that encapsulates the input embeddings, we fine-tune the encoder's lower layers in our pre-trained model by freezing the weights of the



(a) On cross-arch dataset (b) On cross-env dataset (c) On cross-arch dataset (d) On cross-env dataset

Fig. 6. Results (Precision & Recall) of transfer learning on different binary datasets and with different strategies

encoder's higher layers and all the layers in the decoder. Such a method is also widely applied in related transfer learning works [17, 22, 40, 48].

Figure 6 shows the evaluation results of different models trained with different strategies in terms of precision and recall. As can be noticed, (1) if we only use 30% training samples to train a model from scratch, the performance of the trained model is worse than the model trained with all training data or trained with a pre-trained model; (2) even with 30% training samples, the transferred model converges faster than any model trained from scratch and only has a very small performance degradation (2.5% in precision and 2.4% in recall) comparing to the model trained with the whole training set. Meanwhile, in the case study (§5.5), we will use the transferred model, which is aligned with the baseline techniques to conduct the comparison and evaluation on the same datasets. In conclusion, Bin2Summary can be smoothly transferred to summarize the functionality of cross-architecture and cross-environment binaries without requiring massive training data.

5.5 Case Study: Binary Code Comprehension

To evaluate the performance of Bin2Summary in providing semantic information for binary code comprehension, we perform a comparison and conduct a user study with state-of-the-art (*i.e.*, Debin [21], NFRE [16], and SymLM [27]) that predict the function names, and some examples shown in the supplementary materials. As Bin2Summary does not share the same ground truths (*i.e.*, the functionality description extracted from the comments) with those works (*i.e.*, the function name), it could be non-sense to compare the precision and recall. As such, to understand whether our functionality summary is more helpful than the predicted function name for analysts in binary code comprehension, we invited 10 participants to conduct a user study. Among the 10 participants, 2 of them are employed in distinct information technology companies, 8 of them are students (MSc and Ph.D.) and researchers from three universities located in two different countries. Most importantly, we ensure that each participant we invite has a background in C/C++ programming. Specifically, the participants who work in the IT company have C/C++ application development experience, the students have attended at least one course on C/C++ programming, and the researchers have written papers related to C/C++ application development.

During the user study, each participant is asked to go through the source code, the predicted names, and the generated functionality summaries of 100 randomly selected functions shown in a questionnaire. It is worth mentioning, Debin, NFRE, and SymLM can only analyze the ELF files, hence we apply *Transfer Learning* to make Bin2Summary analyze the same ELF files (all complied under x64 and with -O0 optimization) to avoid the deviation of binaries. Participants

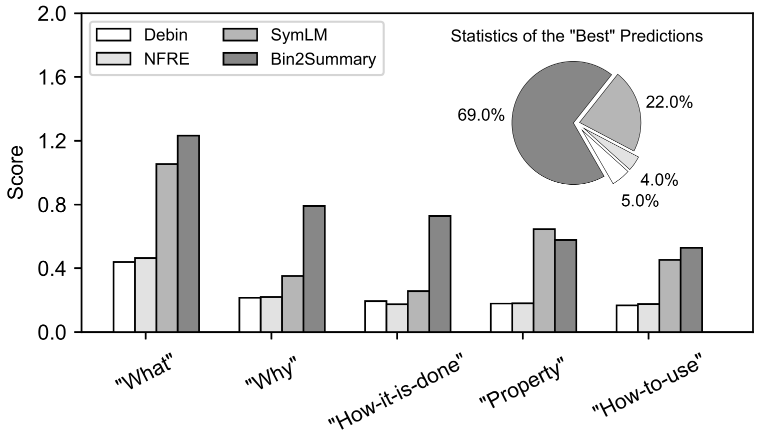


Fig. 7. User study results on program comprehension metrics and statistics of the "best" predictions

are then required to score the given textual descriptions from five different perspectives: "What" (*i.e.*, a definition of the main executions), "Why" (*i.e.*, the reason why the function is provided), "How-it-is-done" (*i.e.*, details of the function implementation), "Property" (*i.e.*, the settings, attributes, or characteristics) and "How-to-use" (*i.e.*, a description about the expected set-up and usage), and such metrics are based on the comprehensive taxonomy developed by CPC [61]. By summing up the scores, we can know the "best" predictions chosen by the participants for each function. A more concrete guide for the user study is presented in supplementary material.

According to the statistics, the participants spend an average of 3 minutes analyzing an individual function including reading its source code and scoring the predicted function names and the generated summary. After analyzing the questionnaires that were finished independently, we present the statistical results in Figure 7. In most cases (69.0%), the generated summary of Bin2Summary is considered better than the function name predicted by existing techniques, and there is a high degree of agreement between these participants with the value of Krippendorff's alpha [20] of 0.95 on the "best" predictions. Particularly, compared to the predicted function names, the generated summaries have significantly better performance in explaining the implementation details of the function and the reason why the function is provided. Therefore, these results show that the functionality summaries generated by Bin2Summary are more intuitive, accurate, and reliable than the predicted function name by state-of-the-art.

6 DISCUSSION

In this section, we discuss different cases that can bias our evaluation results and the limitations of our work together with future works.

Alternative Components in the Framework. We implement the prototype of Bin2Summary on top of the existing disassembler (*e.g.*, Radare [52], Binary Ninja [8]) and assembly language models (*e.g.*, DeepBinDiff [14], PalmTree [30]) because these techniques are state-of-the-art in binary code embedding. However, Bin2Summary is not fundamentally specific to these tools, and our framework should conceptually work with any tools that can generate function embeddings for stripped binaries. Similarly, the components (*e.g.*, the word2vec model in *Functionality Description Embedding*, the seq2seq model in *Model Training*) in our proposed framework can also be replaced by other state-of-the-art techniques if they are designed for the same task. More specifically, we can utilize the Transformer [53] as the seq2seq model in *Model Training*, and theoretically, the performance of Bin2Summary should be further improved by doing so.

Cross-platform Analysis. Bin2Summary is designed as a cross-platform (*i.e.*, cross-operating system) framework to work for associated executable binaries, such as the PE files for the Windows OS and the ELF files for the Linux OS, though our prime datasets are developed with binaries targeting the Windows OS in this paper. By applying the *Transfer Learning*, Bin2Summary is capable of analyzing the ELF file, which is also demonstrated in the user study (§5.5), which proves Bin2Summary can be smoothly utilized in cross-platform binary analysis.

Obfuscation. Bin2Summary assumes its inputs are the binaries without obfuscation. Even though many binaries have been protected with different techniques, tools such as unpacker and de-obfuscator are sufficient to handle these binaries, and these tools can be adopted in our framework and work independently with Bin2Summary to preprocess the obfuscated code. As such, this assumption is believed not to limit our scalability significantly, and we will address the problem of analyzing the obfuscated functions in our future work.

Selection of Projects in the Dataset. As a learning-based approach, Bin2Summary relies on the dataset in the *Model Training* to summarize the functions in stripped binaries, and the scope of the dataset may constrain the performance of Bin2Summary. The current prototype is trained on a dataset with functions related to utilities, database management, and networking, so it may mispredict functionalities outside those categories. Fortunately, Bin2Summary has good transferability, *i.e.*, it is able to learn other kinds of functionalities with limited extra training data. Meanwhile, a large-scale dataset may further improve the performance of Bin2Summary. Therefore, one of our future works is to extend the functionality scope by enlarging our dataset.

Binary Code Summarization with LLM. As the state-of-the-art in NLP, the large language models (*i.e.*, LLM) such as ChatGPT[4] have shown great performance in summarizing the decompiled or disassembled functions. However, with several test cases (more details of the test cases are provided in supplementary material), we observe that ChatGPT is not robust and highly dependent on the syntactic features (*e.g.*, the strings). Also, as its interpretability analysis is out of scope, we leave these problems in our future research. Meanwhile, the reinforcement learning part in ChatGPT also needs massive data, we believe Bin2Summary can be used to facilitate and improve the ChatGPT training in future work.

7 RELATED WORK

Bin2Summary involves a set of techniques and applications. There are two types of related works that share similar tasks with Bin2Summary, so we would like to compare our work with existing techniques in the information recovery for reverse engineering and code summarization for software analysis. In addition, since Bin2Summary has utilized transfer learning techniques, we will discuss the similarities and differences between Bin2Summary and the existing works in this area.

7.1 Information Recovery in Binaries

Software in the wild is usually released as stripped binaries that contain no debug information such as function names [12, 16, 21, 27], function types [10, 21], and variable names [7, 10, 21, 25] to help facilitate some reverse engineering tasks (*e.g.*, binary code comprehension). Among the existing works, the function name is regarded as the most helpful information for developers to understand the information of a given function, which makes the function name recovery meaningful in binary analysis. Debin [21], Nero [12], NFRE [16], and SymLM [27] are recent studies on this

[4]https://chat.openai.com

issue to recover the function name by adopting deep learning techniques. However, the limited semantics in function names constrain the performance of these works. Meanwhile, all of them rely on coarse-grained binary code similarity detection or encoding techniques. Different from the existing techniques, *BIN2SUMMARY uses a fine-grained code embedding method to capture the specific functionality in binaries and generates functionality summaries with abundant semantics.*

7.2 Source Code Summarization

Many works aim to generate comments or descriptive summaries for the source codes or code changes (*e.g.*, patch). With the advance of deep learning, the most straightforward approaches that automatic generation upon the manually crafted templates [37, 38, 41] are gradually exceeded by other state-of-the-art approaches. Based on the methodologies, there are two widely used frameworks: (1) information retrieval-based framework [19, 56, 57, 61] and (2) machine translation based framework [1, 2, 23, 24, 26, 29, 31, 36, 54, 63–65]. In a word, information retrieval approaches aim to generate comments for a source code snippet by cloning the comments of similar code in the corpus. And the machine translation-based frameworks use learning techniques to establish the mappings from the expected comments to the source code snippet. Unlike these existing works, *BIN2SUMMARY focuses on the stripped binary code instead of the source code*, which is a more challenging task because there is only limited semantic information in stripped binaries and the binaries will be influenced by different compilation settings (*e.g.*, architecture, environment).

7.3 Transfer Learning in Related Areas

There are some works that successfully apply transfer learning in software engineering or NLP such as software defect classification [42], text-to-text prediction [48], text sentiment classification [55], and multi-language text classification [47]. However, all of them focus on leveraging the existing well-trained model in new tasks, because none of them face the challenges in binary analysis (*i.e.*, the binary code compiled under different architectures or environments are significantly different). Compared to these prior works, *BIN2SUMMARY uses transfer learning to make it applicable in cross-architecture and cross-environment binary analysis*, which is also the first binary analysis framework that can use limited extra training data in transfer learning.

8 CONCLUSION

In this work, we focus on a question about interpreting the functionality of a function in stripped binaries. To address the challenges, we have presented the design, implementation, and evaluation of BIN2SUMMARY, a tool that can analyze the binaries and generate functionality summaries of the functions in natural language. We presented a set of new techniques, including new approaches that identify the functionality-relevant code snippets in binaries and transform the recognized semantic information of functionality into natural language sentences. The evaluation results show the efficiency, effectiveness, and transferability of BIN2SUMMARY outperforms the state-of-the-art. Meanwhile, the case studies demonstrate BIN2SUMMARY's top-tier performance in practical usage that provides abundant semantics for facilitating binary code comprehension.

ACKNOWLEDGEMENT

We want to thank Dr. Qingchuan Zhao for his enormous help with this paper, as well as our anonymous reviewers for their valuable comments. This work was supported in part by National Key Research & Development Project of China (Grant No. 2019YFB1804400), and Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14209720. Jiongyi Chen was supported by the Natural Science Foundation of China (Grant No. 62302508) and the Natural Science Foundation of Hunan Province, China (Grant No. 2022JJ40553).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. arXiv:2005.00653

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. PMLR, 2091–2100.

[3] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. arXiv:1808.01400

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473

[7] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. 2021. Variable Name Recovery in Decompiled Binary Code using Constrained Masked Language Modeling. arXiv:2103.12801

[8] Binaryninja [n.d.]. Binary Ninja. https://binary.ninja/.

[9] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.

[10] 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 *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.

[11] Cygwin [n.d.]. Cygwin. https://www.cygwin.com/.

[12] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[13] Steven HH Ding, Benjamin CM Fung, and Philippe 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)*. IEEE, 472–489.

[14] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. *DeepBinDiff: Learning program-wide code representations for binary diffing.* eScholarship, University of California.

[15] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh

[16] 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*. 607–619.

[17] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. 2019. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4805–4814.

[18] Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems* 85 (2018), 88–96.

[19] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, Vol. 2. IEEE, 223–226.

[20] Andrew F Hayes and Klaus Krippendorff. 2007. Answering the call for a standard reliability measure for coding data. *Communication methods and measures* 1, 1 (2007), 77–89.

[21] 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*. 1667–1680.

[22] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[25] 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*. 20–30.

[26] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[27] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1631–1645.

[28] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.

[29] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[30] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251.

[31] Yuding Liang and Kenny Q. Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (New Orleans, Louisiana, USA) *(AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 641, 8 pages.

[32] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 389–400.

[33] Xin Ma, Shize Guo, Haiying Li, Zhisong Pan, Junyang Qiu, Yu Ding, and Feiqiong Chen. 2019. How to make attention mechanisms more practical in malware classification. *IEEE Access* 7 (2019), 155270–155280.

[34] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–37.

[35] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[36] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[37] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[38] Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. arXiv:1301.3781

[40] Sangwoo Mo, Minsu Cho, and Jinwoo Shin. 2020. Freeze the discriminator: a simple baseline for fine-tuning gans. arXiv:2002.10964

[41] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[42] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2017. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering* 44, 9 (2017), 874–896.

[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[44] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702.

[45] 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).

[46] 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*. 701–710.

[47] Peter Prettenhofer and Benno Stein. 2010. Cross-language text classification using structural correspondence learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*. 1118–1127.

[48] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 140 (2020), 1–67.

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

[50] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 11–20.

[51] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.. In *NDSS*, Vol. 1. 1–1.

[52] Radare2 Team. 2017. Radare2 GitHub repository. https://github.com/radare/radare2.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[54] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[55] Chang Wang and Sridhar Mahadevan. 2011. Heterogeneous domain adaptation using manifold alignment. In *Twenty-second international joint conference on artificial intelligence*.

[56] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389.

[57] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.

[58] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. 2018. Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 127–134.

[59] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: a tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244.

[60] Shih-Yuan Yu, Yonatan Gizachew Achamyeleh, Chonghan Wang, Anton Kocheturov, Patrick Eisen, and Mohammad Abdullah Al Faruque. 2023. Cfg2vec: Hierarchical graph neural network for cross-architectural software reverse engineering. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 281–291.

[61] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1359–1371.

[62] Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Program logic based software plagiarism detection. In *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 66–77.

[63] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[64] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.

[65] Wenhao Zheng, Hong-Yu Zhou, Ming Li, and Jianxin Wu. 2017. Code attention: Translating code to comments by exploiting domain features. arXiv:1709.07642

Received 2023-09-13; accepted 2024-01-23