CALLEE: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning

Wenyu Zhu[†‡], Zhiyao Feng[†‡], Zihan Zhang[†‡], Jianjun Chen[†§], Zhijian Ou[†], Min Yang[⋆], Chao Zhang[†‡§*]
† *Tsinghua University, Beijing, China* ‡ *BNRist* § *Zhongguancun Laboratory*
⋆ *Fudan University, Shanghai, China*
* *Corresponding author*

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179482

***Abstract*—Recovering binary programs' call graphs is crucial for inter-procedural analysis tasks and applications based on them. One of the core challenges is recognizing targets of indirect calls (i.e., indirect callees). Existing solutions all have high false positives and negatives, making call graphs inaccurate. In this paper, we propose a new solution CALLEE combining transfer learning and contrastive learning. The key insight is that, deep neural networks (DNNs) can automatically identify patterns concerning indirect calls. Inspired by the advances in question-answering applications, we utilize *contrastive* learning to answer the callsite-callee question. However, one of the toughest challenges is that DNNs need large datasets to achieve high performance, while collecting large-scale indirect-call ground truths can be computational-expensive. Therefore, we leverage *transfer* learning to pre-train DNNs with easy-to-collect direct calls and further fine-tune DNNs for indirect-calls. We evaluate CALLEE on several groups of targets, and results show that our solution could match callsites to callees with an *F1-Measure* of 94.6%, much better than state-of-the-art solutions. Further, we apply CALLEE to two applications – binary code similarity detection and hybrid fuzzing, and found it could greatly improve their performance.**

1. Introduction

Indirect calls (*icalls* for short) allow programs to determine the choice of functions to call (i.e., callees) until runtime, enabling programmers to realize dynamic features, and thus are commonly used in object-oriented programming as well as some large-scale programs such as the Linux kernel. Meanwhile, icalls play an important role in program analysis and related tasks. One can complement Call Graphs (CGs) of programs by recognizing targets of indirect calls (*icallees* for short), and many tasks can benefit from precise CGs such as inter-procedural data-flow analysis [1], binary code similarity detection [2], and even test case generation for fuzzing [3]. For example, SelectiveTaint[4] relies on CG reconstruction for taint analysis, αDiff [5] and DeepBinDiff [6] perform binary diffing with CG features, and TEEREX[7] requires precise CGs to perform symbolic execution. Conversely, imprecise icallee analysis will lead to obstacles in many applications, such as false positives in bug detection [8], [9], [10] and path explosion in symbolic execution [11], [12].

In practice, it is common to utilize static analysis to infer icallees, because dynamic techniques can miss many legitimate callees due to poor code coverage . Given target programs with or without source code, applicable static analysis solutions are different. When the source code is available, points-to analysis [13], [14] and type-based analysis [15], [16] are the most common methods. *Otherwise statically determining icallees is much more challenging,* since much information (e.g., type) is missing in binaries.

Existing binary-level solutions in general apply an approximation algorithm to recognize icallees. For instance, binary analysis tools that are widely used in practice (e.g., IDA Pro [17], Angr [12], GHIDRA [18]) and PathArmor [19] identify icallees by constant propagation, and can only resolve very few targets. On the other hand, CCFIR [20] adopts the address-taken policy and treats all address-taken functions as potential icallees, thus having high false positives. τCFI [21], TypeArmor [22] and its refinement [23] reduce icallees to reduce false positives by first recovering function prototypes and then performing type-based matching, but have low guarantees of correctness. The state-of-the-art solution BPA [24] performs a delicate pointer analysis based on a block memory model and a special intermediate representation language (with only support for x86) to infer icallees, but the prototype did not support C++ binaries and still has relatively low precision. A better solution to recognize icallees in binaries is therefore demanded.

In this paper, we propose a deep-learning solution CALLEE to recognize icallees at the binary level. Given an indirect callsite (*icallsite* for short), CALLEE will answer which callees could be its potential targets. *The key insight is that, with sufficient data, DNNs can automatically identify patterns concerning icalls, which can be much more efficient than introducing approximation algorithms or heuristic rules to handle various cases.* Specifically, combining contrastive learning and transfer learning, DNNs can learn to match callsites and callees by comprehending their contexts, i.e., instructions nearby callsites and of callees.

Contrastive learning aims to represent similar inputs with similar embeddings in the latent space, and has been proved effective in question-answering scenarios [25], [26]. Thus regarding a callsite as a question and a callee as its corresponding answer, we build a contrastive-learning framework to match callsites with callees. Beforehand, we perform slicing to extract instructions for callsites and callees based on the calling convention, and embed the generated slices by adjusting a popular representation learning technique

DOI 10.1109/SP46215.2023.00112

doc2vec [27] to the assembly language. In addition, we propose a new symbolization policy to symbolize assembly tokens to improve the model performance and meanwhile handle the out-of-vocabulary (OOV) problem.

Moreover, DNNs need large datasets to achieve high performance, while collecting icall ground truths requires computational-expensive dynamic analyses . Whereas direct calls (*dcalls* for short) can be easily obtained with static analyses. Thus it would be exceedingly beneficial if we can train DNNs for icalls with dcalls, i.e., transfer learning, which reuses a pre-trained model for one task as the starting point for a model on another task. It has been proved efficient to transfer knowledge between languages, images and voices [28], and recently in program analysis [29]. Considering that dcalls and icalls share similar calling conventions, it is possible to transfer knowledge learned from dcalls to icalls. Therefore, we leverage transfer learning to train DNNs for icalls based on abundant dcalls. Specifically, we utilize contrastive learning to answer the callsite-callee question for both dcalls and icalls, while the icall DNN is initialized with a pre-trained dcall DNN.

We have implemented a prototype of CALLEE and evaluated it on targets that have abundant icalls such as the Linux kernel and the Firefox browser [30]. The evaluation results show that CALLEE could match callsites to callees with an *F1-Measure* (*F1*) of 94.6%, , recall of 90.9%, and precision of 97.3%, outperforming BPA [24], TypeArmor [22] as well as real-world binary analysis tools such as IDA Pro [17] .

Further, we have demonstrated that CALLEE can benefit down-stream applications based on call graphs. Firstly, we applied CALLEE to binary code similarity detection and greatly improved the state-of-the-art solution DeepBinDiff with an average increase of 4.6% *F1* in cross-version binary diffing and 13.7% in cross-optimization binary diffing. Moreover, CALLEE is applied to the widely-used hybrid fuzzing solution Driller [31]. In three 24-hour fuzzing campaigns, it can help the fuzzer find 50% more paths on average in all 8 CGC [32] challenges that have icalls.

Additionally, we have made an attempt to interpret the neural network with a case study where CALLEE surpassed other solutions. It showed that the proposed model can well capture semantic features of tokens in assembly instructions, and tokens related to arguments and return values contribute the most to icallee recognition, which is consistent with the domain knowledge of binary analysis.

In summary, we make the following contributions:

- We present the first transfer- and contrastive-learning approach CALLEE integrated with expert knowledge to recognize icallees and recover call graphs for binaries.
- We propose a new symbolization method for machine-learning solutions on assembly language, which can preserve data-flow information of assembly contexts and meanwhile does not introduce the OOV problem.
- We have collected the largest set of callsite-callee training data. The dataset and nerual model are available at https://github.com/vul337/Callee.
- We evaluate CALLEE with real-world programs and demonstrate that it outperforms state-of-the-art solutions on the callsite-callee matching task.
- We demonstrate that CALLEE is highly effective at promoting tasks based on CGs, e.g., binary code similarity detection or hybrid fuzzing.

2. Background and Related Work

2.1. Transfer Learning

Given a source domain $D_S = \{X_S, f_S(X, \theta_S)\}$ and learning task T_S, a target domain $D_T = \{X_T, f_T(X, \theta_T)\}$ and learning task T_T, transfer learning aims to help improve the learning of the target predictive function f_T in D_T using the knowledge in D_S and T_S, where $D_S \neq D_T$, or $T_S \neq T_T$. In general, one of the most common methods to perform transfer learning is to initialize f_T with parameters of the pre-trained f_S, i.e. using θ_S as the initial value of θ_T.

In fields of Natural Language Processing (NLP), transfer-learning techniques [33] have been proposed to transfer knowledge between two languages (e.g., English and Nepali). Recently, PLATO [29] proposed a cross-lingual transfer-learning framework for statistical type inference in source code. And StateFormer [34] utilized a pretrain-finetune architecture to recover function type information from assembly code, shedding light on applications of transfer learning on program analysis.

2.2. Contrastive Learning

Contrastive learning aims to teach a neural model to pull together the representations of matching samples in a latent space, and meanwhile separate non-matching ones. The most common method is through a Siamese network [35], which is a structure with two parallel networks to extract feature vectors of two input samples, and calculate the distance with another neural network or pre-defined norms. At first, the Siamese network was proposed to compare the similarity of two inputs. It consists of two identical networks with identical structures and weights. Distance between the feature vectors of inputs will be calculated and used as the similarity/difference score. Previous studies such as αDiff [5] and NMT [36] have shown that the Siamese network could be utilized to extract fine-grained semantic features of binary code, even if the code is from cross-version or cross-architecture binaries.

Recently, another type of Siamese network is introduced to address more complicated problems. The new structure, also called a pseudo-Siamese network, allows two networks to be different or not to share weights to adapt to application scenarios which require different categories of inputs. As

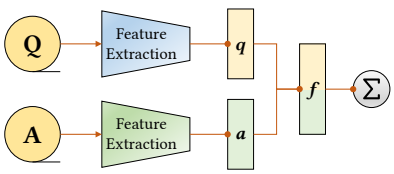


Figure 1: Illustration of the Siamese network

shown in Figure 1, in the question-answering scenario, two different networks can be utilized to extract features of a question (**Q**) and an answer (**A**) respectively. To calculate the similarity/difference, the extracted feature vectors $\boldsymbol{q}$ and $\boldsymbol{a}$ could be concatenated together as a feature vector $\boldsymbol{f}$, which will be further fed into a following classifier network Σ. The classifier will output a score indicating how much **Q** and **A** matches. This structure could be trained to match questions with answers, as shown in [37], [38], [39].

2.3. Applications based on Call Graphs

Binary program analysis applications often have to track data flow between functions to comprehend the semantics of programs, and thus have to conduct inter-procedural program analysis by traversing programs' Call Graphs (CGs) which represent functions calling relationships to track information flow or capture the semantics. Such applications include but are not limited to the followings.

Binary Similarity Detection. BinDiff [40] matches functions based on their position or neighborhoods in CGs. αDiff [5] extracts inter-function and inter-module features based on CGs, and further calculates feature distances with a Siamese neural network. DeepBinDiff [6] utilizes CGs to construct inter-procedural control-flow graphs (ICFGs) and performs random walks on them to embed each basic block.

Hybrid Fuzzing. Driller [31] leverage symbolic execution engines to solve inputs for program paths when the AFL [41] fuzzer gets stuck, and SymQemu [42] further proposes a compilation-based symbolic execution policy to boost the speed of the symbolic executor. However, they do not resolve icallees due to the path-explosion problem. Thus by providing symbolic execution engines with a limited set of candidate targets, we can ease the path-explosion problem and thereby enable hybrid fuzzers to resolve icallees to improve the code coverage.

Except for aforementioned applications, CGs are also vital in malware detection [43], bug detection [44], [45] and many other scenarios [46], [47], [48].

Therefore the completeness and accuracy of CGs greatly affect the results of these applications. Otherwise, it may cause issues like false positives in bug detection, path explosion in symbolic execution, etc.

2.4. Recognizing Indirect Callees in Binaries

At the core, constructing a complete and accurate CG requires to precisely recognize icallees. Many solutions have been proposed to address this problem, but few can recognize icallees for binaries.

Type-based Analysis. Identifying icallees in binary programs in general requires type recovery analysis [49] which is error-prone, as shown in τCFI [21], TypeArmor [22] and its refinement [23]. Otherwise, a coarse-grained address-taken policy would be applied, as shown in CCFIR [20], in which arbitrary address-taken functions are marked as legitimate icallees, causing more false positives.

Pointer Analysis. SVF [13] leverages Andersen's algorithm and constructs an inter-procedural static single assignment (SSA) form to capture def-use chains of both top-level and address-taken variables, While whole-program analyses such as SVF and SUPA [14] have troubles on programs composed of separately compiled modules. K-Miner [50] splits kernel code based on system calls, and PeX [51] leverages the common programming paradigm used in kernel abstraction interfaces, but they have not scaled to user-mode binaries. Some binary analysis tools such as BAP [52] and Angr [12] leverage value-set analysis to resolve pointers, but face challenges on complex real-world programs. BDA [53] proposes a path sampling algorithm to perform dependency analysis while introducing huge runtime overhead, even more than dynamic testing on multiple targets from the SPECINT2000 benchmark [54], making it impractical. Recently, BPA [24] adds scalable pointer analysis support for binaries based on a special block memory model and intermediate representation (IR), while the prototype currently supports 32bit C programs only.

2.5. DNN-based Binary Analysis

Recent research has leveraged DNNs to solve many program analysis problems.

Function Recovery. Shin et al. [55] show that recurrent neural networks (RNNs) can identify functions in binaries precisely. It converts each byte into a vector with one-hot encoding, and concatenates vectors of all bytes as the representation of functions. Then it trains an RNN and uses the softmax function to predict whether a byte begins (or ends) a function. XDA [56] improves the performance by applying a BERT [57] model. EKLAVYA [58] and StateFormer [34] further recovers function signatures from assembly code. EKLAVYA embeds each instruction into a vector and concatenates them to represent functions, and predicts a type tuple for all the parameters of a function with an RNN. StateFormer [34] utilizes transfer learning with a transformer [59] model to learn type inference rules. However, they both cannot recover the signature of a callsite, and thus cannot recognize icallees.

Value-set Analysis (VSA). DEEPVSA [60] uses DNNs to facilitate VSA by learning semantics of instructions and capturing dependencies in contexts at the binary level, which can further assist alias analysis for crash diagnosis. But the application in resolving icallees needs further study.

Binary Similarity Detection. αDiff first utilizes a DNN to learns code features from raw bytes, then extracts inter-function and inter-module features and adopts a Siamese neural network to detect similarity between binaries. BinaryAI [61] uses BERT to pre-train the binary code on several tasks and adopts convolutional neural network (CNN) to extract the order information of CFG's nodes. NMT [36] proposes a DNN-based cross-lingual basic-block embedding model to measure the similarity of two blocks, which achieves cross-architecture similarity detection. By regarding instructions as words and basic blocks as sentences, they use word2vec [62] to embed instructions and use LSTM [63]

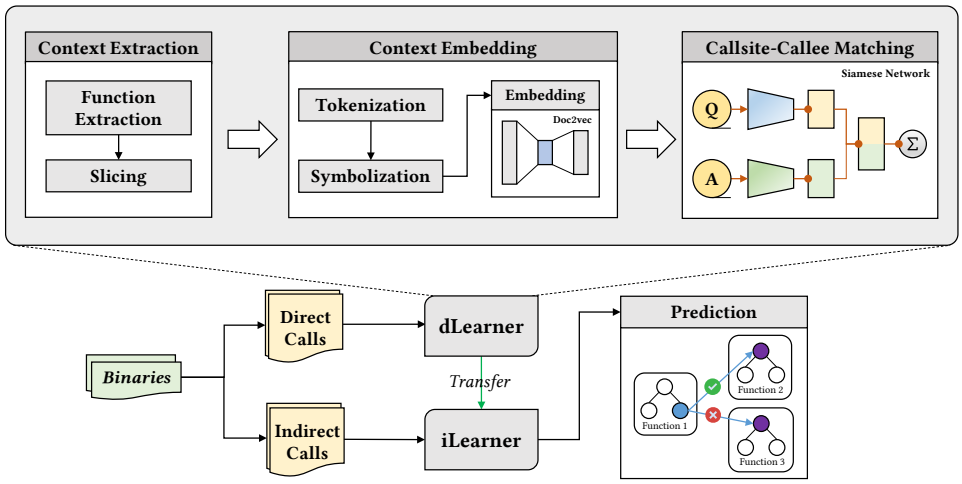


Figure 2: Overview of our solution CALLEE.

to embed basic-blocks. The state-of-the-art DeepBinDiff [6] uses both the code semantics and the program-wide control-flow information to generate basic block embedding.

To the best of our knowledge, *we are the first to use deep learning to comprehend contexts of call instructions and recognize icallees*, and utilize it to recover CGs for binaries with a high precision.

3. Overview

Our goal is to design a callsite-callee matching system that can automatically recognize which callees are potential targets for a given callsite. In this section, we describe the overview of our solution CALLEE.

Overall workflow. As shown in Figure 2, we first train a contrastive-learning framework **Learner** with dcalls (denoted by **dLearner**), and transfer the learned knowledge into a icall **Learner** (denoted by **iLearner**). In detail, parameters of the **iLearner** are initialized with the pre-trained **dLearner**. The **iLearner** will further be trained with icalls and used to perform icallee prediction. To build such a **Learner** framework, we employ three major modules, i.e., context extraction module, context embedding module, and callsite-callee matching module. The key insight is that, *neural networks can learn to match callsites with callees by comprehending their contexts, i.e., instructions nearby callsites and of callees.*

3.1. Core Modules of the Learner

3.1.1. Context Extraction. Contexts related to callsites and callees form the basis of decisions made by neural networks. Therefore, given a binary program, we first need to extract proper contexts from the binary. Full contexts, i.e., all instructions of a function, make it difficult to construct favorable embeddings of limited vector dimensions. Therefore, shrinking the contexts while keeping necessary information is critical. We adopt inter-procedural slicing with expert knowledge to extract related contexts.

3.1.2. Context Embedding. Since neural networks require vectors as inputs, contexts of callsites and callees have to be represented in the form of vectors. Existing studies [5] have shown that NLP solutions are effective at binary analysis. We thus utilize a popular NLP model doc2vec to embed program slices. Moreover, we adjust the doc2vec model with domain knowledge, i.e., differences between assembly and natual languages.

3.1.3. Callsite-callee Matching. Inspired by question-answering scenarios, CALLEE regards a callsite as a question and a callee as its corresponding answer. To compute the difference score of a callsite and a callee, CALLEE adopts contrastive learning, i.e., a Siamese neural network. The network takes a pair of callsite and callee embeddings as input, and generates their feature vectors, which will be concatenated together and fed into a classifier to calculate the difference score of the input pair.

3.2. Workflow of the Learner

The input to the **Learner** is plenty of binaries, and outputs are models that could be used to embed program slices and report difference scores. In total, there are 5 steps.

* **1: Collecting ground-truth callsite-callee pairs.** For dcalls, we simply extract callsite-callee pairs based on call instructions. For icalls, we dynamically run several testing programs with provided test suites and collect callsite-callee pairs at runtime. Specifically, we utilize Intel PT [64] to collect traces for user-mode binaries and PANDA [65] for the Linux kernel.
* **2: Statically extracting callsite-callee pair slices and functions from binaries.** With collected ground truths, we apply an inter-procedural slicing algorithm on binaries to extract slices for each callsite and its associated callee. Meanwhile, we build a function dataset from training binaries to train an embedding model later.
* **3: Slice preprocessing and embedding.** In this step, we symbolize instructions in the slices to reduce dimensions of data used in the following embedding model and

neural network to make those models converge faster. Meanwhile, we train a doc2vec model using the collected function dataset. The doc2vec model is then used to embed slices into vectors required by the neural network.

* **4: Establishing a vectorized callsite-callee dataset.** In this step, we vectorize positive (matching) and negative (non-matching) callsite-callee pairs with the trained doc2vec model. Subsequently, we label positive ones as 1 and negative ones as 0.
* **5: Training a Siamese neural network.** In this step, we construct a Siamese neural network with two parallel feature extraction layers, and train the network with the labeled dataset to produce difference scores.

3.3. Workflow of the Transfer Learning

With the proposed **Learner** framework, we perform transfer-learning between **dLearner** and **iLearner**.

* **1: Pre-training the dLearner.** With collected binaries, we first train the **dLearner** with statically-extracted dcall pairs, following the standard train-validation-test procedure. After pre-training, we select the best-performance models for transfer learning.
* **2: Initializing the iLearner.** We initialize parameters of models in **iLearner** with values of corresponding parameters of the selected models, including the parameters of the doc2vec model and the Siamese network.
* **3: Fine-tuning the iLearner.** Finally, we train the models of **iLearner** with dynamically-collected icall pairs after initialization, i.e., fine-tuning.

4. Methodology

We first introduce the contrastive **Learner** in detail, i.e., context extraction, context embedding and callsite-callee matching, and then describe the transfer learning.

4.1. Context Extraction via Slicing

Recent studies have shown that DNNs trained in a completely data-driven way without domain knowledge may be non-explainable and unpredictable, whose results may even conflict with prior expert knowledge. However, a system based completely on expert knowledge may have limitations in the scope and capability of solving problems, due to insufficient knowledge or improper inference logic.

Therefore, we integrate expert knowledge into the deep learning system. Specifically, we perform program slicing in advance. The slicing step aims at using expert knowledge to preliminary extract useful information for matching callsite and callee pairs. Besides, shorter code gadgets after slicing are more favorable for embedding.

The principle of slicing is to identify and preserve instructions related to data dependencies between icallsites and icallees, including local variables that passed between functions (arguments and return values) and global variables. To get as much information as possible, we perform a depth-first traversal of all basic blocks in callsite and callee function's control-flow graph (CFG). For global data dependencies, we keep instructions whose operands are related to values in the data segment. For inter-procedural local data dependencies, we keep those concerning stack memory and registers used for function arguments and return values, based on rules of data passing [66] shown in Table 1. To be conservative, we do not drop control-flow instructions. Details of slicing algorithms are presented in Section 5.2.

TABLE 1: Data passing rules in the calling convention of the System V AMD64 Application Binary Interface (ABI).

Data Type	Example	Passing
INTEGER, POINTER	`char`, `short`, `int`, `long`	Argument: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` Return value: `rax`, `rdx`
SSE, SSEUP	`float`, `double`	Argument: `xmm0` to `xmm7` Return value: `xmm0`, `xmm1`
X87, X87UP, COMPLEX_X87	`long double`	Argument: stack Return value: `st0`, `st1`
MEMORY	struct, array, union	Argument: stack Return value: (address in) `rax`

4.2. Context Embedding

Required by most neural networks, inputs need to be embedded into vectors or tensors. Therefore, we adopt doc2vec, a common approach in the field of NLP, to embed slices.

Before embedding, instructions should be tokenized to avoid nonexistent tokens caused by punctuation. For instance, instruction `mov rax, [rdi]` should be tokenized into `"mov", "rax", ",", "[", "rdi", "]"`. Moreover, instructions from a fresh binary may have tokens unseen in the trained doc2vec model, known as the Out-of-Vocabulary (OOV) phenomenon. Thus we need to symbolize slices before embedding.

4.2.1. Symbolization. The general idea of symbolization is to replace open-set tokens with closed-set tokens. Open-set tokens are tokens that can have many variants, including immediate operands, user-defined function names, user-defined variables, and so on. Contrastively, closed-set tokens refer to tokens that have limited variants. For example, `20h` is an open-set token in instruction `mov eax, 20h`. It can be replaced by `num`, which is a closed-set token.

Further, the intensity of symbolization should be taken into account. We compare two symbolization policies: *strict* symbolization and *loose* symbolization. By strict, it means that the symbolization process transforms open-set tokens in the same kind into a single closed-set token. For instance, given an open set of user-defined function names foo_0, foo_1, ..., foo_∞, any token in it will be replaced by the same closed-set token `fun`. Strict symbolization is the most commonly used policy in preprocessing, because it can eliminate OOV. However, strict symbolization may lose data-flow information, which often contributes to the determination of the function call targets. For example, strict symbolization turn all strings into one token `"str"`.

Hence we propose *loose symbolization* to preserve data-flow information and meanwhile maintain a finite-size token corpus. Through modulo arithmetic, an open set like

TABLE 2: Symbolization Rules.

Symbolization	loc_ABCD	arg_ABCD	sub_ABCD	var_ABCD	struct_ABCD	unk_ABCD	byte_ABCD	off_ABCD	*word_ABCD	flt_ABCD	dbl_ABCD	a_String
Strict	loc	arg	fun	var	struct	unk	byte	offset	word	flt	dbl	str
Loose	loc+ABCD%N	arg+ABCD%N	fun+ABCD%N	var+ABCD%N	struct+ABCD%N	unk+ABCD%N	byte+ABCD%N	offset+ABCD%N	*word+ABCD%N	flt+ABCD%N	dbl+ABCD%N	str+len(String)

{foo_0, foo_1,...,foo_∞} can be transformed into {foo_0, foo_1,...,$foo\_(N-1)$} where N is a hyperparameter. As for strings, we simply take the length of a string as a suffix, and replace the string with `str_len`. Additionally, several kinds of tokens are symbolized according to their semantics. For example, operands of a dcall instruction are considered to be a function, and thus we replace them with `"fun"`. Detailed rules of symbolization are summarized in Table 2.

4.2.2. Vectorization. After symbolization, CALLEE adopts doc2vec, a popular model used in NLP, to embed slices into vectors. A doc2vec model takes paragraphs of tokens as input and calculates the distributions of both paragraphs and tokens. To capture the semantic information of low-frequency tokens, we choose the Distributed Bag of Words of Paragraph Vector (PV-DBOW) model [27], and adjust it to apply to assembly language. Note that, compared with word2vec and PalmTree [67] (For detailed evaluation of different embedding techniques, please refer to Appendix C.), doc2vec is able to calculate the word embedding and paragraph embedding at the same time, and the paragraph embedding is shared during multiple training of word embeddings in one paragraph. Thus the generated word embedding in fact involved both inter-token and inter-instruction information.

Formally, an m-token callsite slice $\vec{\pi_i} = \{u_0, u_1, ..., u_m | u \in R^j\}$ and an n-token callee slice $\vec{\alpha_i} = \{t_0, t_1, ..., t_n | t \in R^j\}$ are mapped into

$$G(\vec{\pi_i}) \to \vec{Q_i} = \{\vec{E}_{u_0}, \vec{E}_{u_1}, ..., \vec{E}_{u_m} | \vec{E} \in R^k\}, and$$

$$G(\vec{\alpha_i}) \to \vec{A_i} = \{\vec{E}_{t_0}, \vec{E}_{t_1}, ..., \vec{E}_{t_n} | \vec{E} \in R^k\}$$

where G is the doc2vec model as a mapping $G : \mathbf{X} \to \mathbf{Z}$ between the token space $\mathbf{X} : R^j$ and the embedding space $\mathbf{Z} : R^k$. Note that embeddings for each token in a paragraph are concatenated together, i.e. $\vec{Q_i} \leftarrow \vec{E}_{u_0} \oplus \vec{E}_{u_1} \oplus ...\vec{E}_{u_m}$; $\vec{A_i} \leftarrow \vec{E}_{t_0} \oplus \vec{E}_{t_1} \oplus ...\vec{E}_{t_n}$.

However, doc2vec is designed to be applied to natural languages (e.g., English). But the prior knowledge of natural languages is quite different from the assembly. Thus CALLEE adjusts two parameters of doc2vec intuitively.

- `sample`: In natural languages, high-frequency tokens are mostly function words. Therefore, these tokens are usually downsampled to reduce their frequency. Yet high-frequency tokens in assembly language can carry much information (e.g., `comma` to distinguish operands). As a result, we do not downsample high-frequency tokens.
- `min_count`: Low-frequency words caused by wrong segmentation results of sentences are often ignored during training an embedding model of natural languages. On the contrary, low-frequency tokens in program analysis scenarios can be semantically deterministic. Hence we set the `min_count` parameter to 0.

4.3. Structure of the Matching Network

For embedded callsites and callees, we further build a Siamese neural network to predicate their difference scores.

An embedded callsite slice $\vec{Q_i}$ will pass through feature extraction layers ϕ that output a feature vector $\vec{q_i} = \{\phi(\vec{Q_i}) | \phi : \mathbf{Z} \to \mathbf{F}\}$, where $\mathbf{F} : R^f$ is the feature space. Similarly, for an embedded callee slice $\vec{A_i}$ we can obtain a feature vector $\vec{a_i} = \{\phi'(\vec{A_i}) | \phi' : \mathbf{Z} \to \mathbf{F}\}$ with another set of feature extraction layers ϕ'. Then to calculate the matching score, we concatenate two feature vectors together, considering that currently there is no theoretical proof of which distance measure is optimal for feature vectors. In other words, different data/scenarios may need different distance measures. Therefore we utilize a fully-connected network (FCN) to predict a score with the concatenated vector, i.e., "let the data talk". The FCN σ is essentially an adaptive (trainable) "distance": $d_i = \sigma(\vec{q_i} \oplus \vec{a_i})$.

The contrastive loss [68] is used as the optimization goal of our Siamese network:

$$L = \frac{1}{2N} \sum_{i=1}^{N} [y_i d_i^2 + (1 - y_i) \max\{1 - d_i, 0\}^2]$$

where N is the number of input pairs, y_i (i.e., 1 or 0) is the label of the input pair (i.e., match or not). The optimization goal indicates that, if the input pair match ($y_i = 1$), then the output (difference score) d_i should be close to 0; otherwise, the output should be close to 1.

According to the output d, we can set a threshold to determine whether the callsite and callee match:

$$matching = \begin{cases} yes & d < \text{threshold} \\ no & \text{otherwise} \end{cases}$$

4.4. Transfer Learning

With the proposed **Learner** framework, we utilize a two-stage transfer-learning training mechanism, i.e., pre-training with dcalls and fine-tuning with icalls. Specifically, given two Siamese neural networks

$$\Lambda_d = \sigma_d(\phi_d(\vec{Q_i}, \theta_d) \oplus \phi'_d(\vec{A_i}, \theta'_d))$$

and

$$\Lambda_i = \sigma_i(\phi_i(\vec{Q_i}, \theta_i) \oplus \phi'_i(\vec{A_i}, \theta'_i))$$

for dcalls and icalls respectively, where θ indicates parameters of ϕ. we first train Λ_d with dcall pairs, and then initialize ϕ_i and ϕ'_i with ϕ_d and ϕ'_d, and further fine-tune Λ_i with icall pairs. Note that σ_i is trained from scratch, and the doc2vec model follows the same training mechanism.

5. Implementation

5.1. Dataset Collection

The datasets that CALLEE used require two kinds of data: assembly functions for training the doc2vec model and callsite-callee pairs for training the Siamese neural network.

Algorithm 1: Slicing of a callsite

```
Input: CallsiteSet
Output: CallsiteResult
CallsiteResult ← {}
foreach Callsite in CallsiteSet do
    StackSet, RegSet, GlobalVarSet, CrtlFlowSet ← {}
    for Insn ← FuncStart : Callsite do
        if isStackInsn(Insn) then
            StackSet ← StackSet ∪ {Insn}
        else
            foreach Op in InsnOperands do
                if isArgRegInOp(Op) then
                    RegSet ← RegSet ∪ {Insn}
    for Insn ← Callsite : FuncEnd do
        foreach Op in InsnOperands do
            if isRetRegInOp(Op) then
                RegSet ← RegSet ∪ {Insn}
    GlobalVarSet ← getGlobalVarXref(Function)
    CrtlFlowSet ← getCrtlFlowInsn(Function)
    SliceResult ← StackSet ∪ RegSet ∪ GlobalVarSet ∪
      CrtlFlowSet
    CallsiteResult ← CallsiteResult ∪ {SliceResult}
return CallsiteResult
```

Algorithm 2: Slicing of a callee

```
Input: CalleeSet
Output: CalleeResult
CalleeResult ← {}
foreach Callee in CalleeSet do
    Function ← makeFunction(Callee)
    StackSet, RegSet, GlobalVarSet, CrtlFlowSet ← {}
    for Insn ← FuncStart : FuncEnd do
        if isStackInsn(Insn) then
            StackSet ← StackSet ∪ {Insn}
        else
            foreach Op in InsnOperands do
                if isArgRegInOp(Op) then
                    RegSet ← RegSet ∪ {Insn}
                else if isRetRegInOp(Op) then
                    RegSet ← RegSet ∪ {Insn}
    GlobalVarSet ← getGlobalVarXref(Function)
    CrtlFlowSet ← getCrtlFlowInsn(Function)
    SliceResult ← StackSet ∪ RegSet ∪ GlobalVarSet ∪
      CrtlFlowSet
    CalleeResult ← CalleeResult ∪ {SliceResult}
return CalleeResult
```

5.1.1. Functions. In analogy with natural languages, we regard functions as the "paragraphs", instructions as "sentences", opcodes and operands as "words", and train a doc2vec model to embed slices into vectors.

We write a Python script for IDA Pro to extract functions from binaries. Note that only functions in the `.text` section are extracted. As for those in other sections, we have to identify which shared libraries they are in. All involved shared libraries are analyzed later to extract their functions.

5.1.2. Callsite-callee pairs. The primary goal is to record addresses of callsite-callee pairs in binaries.

Direct-call pairs can be easily obtained with IDA Pro by simply traversing binaries and recording addresses of callsites and callees. For icalls, we utilize dynamic anaylses to collect ground truths. For user-mode binaries, we instrument all icallsites with an LLVM pass to output the callee at runtime. With coverage-guided fuzzers such as AFL [41] and program test suites as fuzzing seeds, we can cover most functional code. After fuzzing, the indirect callsite-callee pairs are collected by running the program with generated inputs. For the kernel, we emulate it in PANDA [69]. By parsing emulation logs, we can obtain the icall pairs. For more details, please refer to Appendix A.

5.2. Slicing

We implement the slicing algorithm with the IDAPython [70] SDK provided by IDA Pro. Before slicing, we filter out cases where IDA Pro fails or goes wrong.

We extract slices from callsites (Algorithm 1) and callees (Algorithm 2), then combine them according to the requirements of training or testing. First, we get the function where the callsite or callee address is located. Since the function boundary of a target function called in an indirect way may not be correctly recognized by static analysis, we force callee addresses to be starts of functions when slicing callees. Then, we walk through instructions of the function, deciding whether to keep them based on operands. To preserve local variables' inter-procedural data dependencies, we identify and retain the information about function signatures. For arguments, we extract instructions concerning stack memory and registers used for arguments from the first half of the callsite function (i.e., instructions before this `call` instruction) and the whole callee function. For return values, we extract instructions containing registers used for return values from the second half of the callsite function (i.e., instructions after this `call` instruction) and in callees. To preserve global variables' data dependencies, we get cross-reference instructions of global variables in both callsite and callee functions. Finally, we gather control-flow instructions, and the union of those parts is taken as the result.

5.3. Embedding

CALLEE utilizes IDA Pro to disassemble instructions, so we take advantage of its naming rules to symbolize instructions. By default, data structures are named according to their addresses. For example, a user-defined function at address `0x43B9D0` in the `.text` section is named as `sub_43B9D0`. Therefore, we can symbolize the function as `fun` (strict) or `func0` (loose), assuming that the hyper-parameter N is set to 10. As shown in Table 2, we consider 12 situations in total.

6. Evaluation

We evaluate CALLEE from the following aspects:

- **Performance of icallee recognition.** We compare CALLEE with SOTA solutions, conduct ablation studies, and discuss its generalization and time efficiency.
- **Applications of CALLEE.** We apply CALLEE to binary similarity detection and hybrid fuzzing to examine whether it can promote their performance.

- **Interpretability of CALLEE.** We interpret the **Learner** framework used by CALLEE.

6.1. Evaluation Setup

Experiments are performed on a machine equipped with Ubuntu 18.04 LTS. The machine has an Intel CPU (Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz), four NVIDIA GPUs (A100 PCIE) and 768GB RAM, and is installed with LLVM 12.0.1, GCC 7.5.0, libipt 2.0.0 (commit `892e12c5`), a docker image of PANDA (git tag: `0729fd0d`), IDA Pro 7.6, and Python 3.6.9. The Python is equipped with gensim 4.2.0 and PyTorch 1.9.0.

TABLE 3: Dataset Statistics.

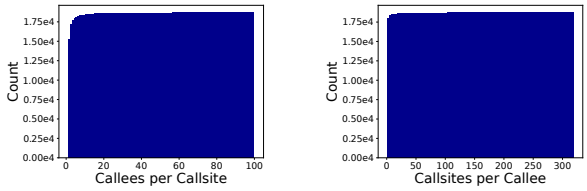
Dataset	# Projects	# Binaries	# Functions	# Pairs
Direct Call	19K	261K	68M	406M
Indirect Call	52	183	343K	30K
GNU Binutils*	1	694	963K	5M

* For cross-compiler and cross-version evaluation.

6.1.1. Datasets. Table 3 shows statistics on the number of projects, binaries, functions and callsite-callee pairs of the datasets we collect. All binaries are in the x64 architecture. For the dcall dataset, we first build enormous binaries automatically with the apt package manager and then extract functions and direct callsite-callee pairs. For the icall dataset, we collect binaries rich in icalls, including the Linux kernel (v5.3.11), the Firefox browser (v72.0a1), and corresponding shared libraries. After dynamic testing, we extract functions from them and perform the slicing. To build a balanced dataset, we set the ratio of positive pairs to negative pairs to 1:1 and assemble negative pairs by randomly choosing unmatched callsites and callees from the ground truths. To avoid negative pairs that are actually positive pairs not covered by dynamic testing, we additionally check the source-level type of the unmatched pairs with the help of debug information, which contains type information of function calls. Note that different projects usually have different contributors, whose coding styles can be varied, e.g., Firefox has over 100 contributors in the last 90 days [71], and thus we believe the datasets are diversified based on the number of projects.

Additionally, we study the distribution of callsites and callees in the icall dataset. For a malformed dataset whose callsites generally have the same small set of callees, almost any algorithm will do well by just guessing those callees the majority of the time. As shown in Figure 3, the number of "callsites per callee" is small for the majority of callees, indicating that the callsites with common callees will be few. And the number of "callees per callsite" is small for the majority of callsites, further demonstrating the diversity of the dataset.

Dataset split. A common split method is cross-validation: randomly choosing, e.g., 70% pairs for training, 20% for validation and 10% for testing, without considering the distribution of the data (e.g., the originating binaries). But it can lead to severe overfitting issues, i.e., the model overfits patterns of data from binaries in the dataset and cannot generalize to data from binaries outside the dataset.



(a) Callee Distribution (b) Callsite Distribution

Figure 3: Distribution of callees per callsite (left) and callsites per callee (right) in the icall dataset.

We have conducted an experiment following this split method. The final F1 scores of the model on the icall dataset are 98.9% for training and 94.6% for testing. However, when we apply the trained model to data extracted from binaries outside the dataset, the F1 drops sharply to about 53.7%, indicating that the trained model's generalization ability is poor. In other words, the model overfits the dataset.

To acquire a better generalization ability across binaries, *we extract pairs from different binaries for training and testing to evaluate the generalization performance across different binaries.* Therefore, we first randomly choose 80% of the binaries for training, 10% for validation, and 10% for testing. Then pairs are further extracted from these binaries. Since the dataset consists of binaries by different authors, thus there is little shared code across the split datasets.

6.1.2. Hyperparameters. We set the `batch_size` to 512, and train the network 20 epochs. The optimizer is `rmsprop`, the learning rate is 0.001, the threshold for the final decision is 0.5, and the embedding dimension of the doc2vec model is 100. The final classifier network of the Siamese neural network is an FCN consisting of three layers with 512, 512, and 1 neuron(s) respectively. The sigmoid function is used as the final activation function. We adopt Batch Normalization [72] and Dropout [73] to help the network converge, and the dropout rate is set to 0.2. The hyper-parameter N of loose symbolization is set to 10. Note that, these hyperparameters are selected based on several rounds of dry-run experiments.

6.1.3. Evaluation Metrics. We choose the common metrics *Precision*, *Recall* and *F1-Measure* (*F1*) to evaluate the performance of models. These metrics are computed from the number of *True Positives* (*TP*), *True Negatives* (*TN*), *False Positives* (*FP*), and *False Negatives* (*FN*). An FP is a pair classified as match but actually does not match. An FN is a pair classified as unmatch but actually matches.

6.2. Performance of CALLEE

6.2.1. Overall Performance. Overall, we choose the loose symbolization method and FCN feature extraction layers and train the Siamese neural network on sliced contexts with the transfer-learning technique. As shown in Table 4, CALLEE has an *F1* of 94.6%, recall of 90.9%, and precision of 97.3%.

TABLE 4: Performance of CALLEE (in bold) on the icall dataset. Results not in bold are presented for ablation studies.

Setting	Context	Symbolization	Siamese Network	Mode	Train			Test		
					Precision	Recall	F1	Precision	Recall	F1
0	Sliced	Loose	FCN	*dcall*	93.4%	87.9%	90.6%	93.8%	87.5%	90.5%
1	Sliced	Loose	FCN	*icall*	76.8%	75.6%	76.2%	70.3%	63.7%	66.8%
2	**Sliced**	**Loose**	**FCN**	***transfer***	**99.2%**	**96.8%**	**98.0%**	**97.3%**	**90.9%**	**94.6%**
3	Sliced	Loose	FCN	*zero-shot*	-	-	-	93.0%	85.9%	89.3%
4	*Full*	Loose	FCN	icall	74.1%	72.9%	73.5%	57.4%	53.0%	55.1%
5	Sliced	*Strict*	FCN	icall	75.3%	74.2%	74.7%	61.9%	56.6%	59.1%
6	Sliced	Loose	*LSTM*	icall	71.0%	70.1%	70.5%	67.8%	61.5%	64.5%
7	Sliced	Loose	*TextCNN*	icall	73.7%	72.3%	73.0%	69.5%	63.0%	66.1%
8	Sliced	Loose	*1dCNN*	icall	77.3%	75.7%	76.5%	68.4%	61.7%	64.9%

6.2.2. Comparison with state-of-the-art solutions. We compared CALLEE with several closely relevant solutions which recognize icallees as well. Since the refinement of TypeArmor fails to discuss their precision/recall in recognizing icallees and has not open-sourced yet, we only compare CALLEE with BPA and TypeArmor as well as popular binary analysis tools such as IDA Pro, Angr and GHIDRA. We use the same binaries as BPA: the SPEC CPU 2006 benchmark and 4 server applications (memcached-1.5.4, lighttpd-1.4.48, exim-4.89, and nginx-1.10).

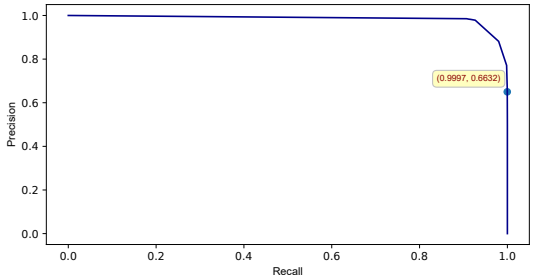


Figure 4: Precision-Recall Curve of CALLEE.

Since BPA is not open-sourced, we adopt the results from their paper: based on a dynamically collected dataset [24], BPA and TypeArmor have precision of 57.6% and 35.1%, recall of 100% and 99.9%, and thus F1-measures of 73.1% and 51.9% respectively. For fair comparison, we report CALLEE's precision-recall (PR) curve in Figure 4. As shown, the precision drops as the recall increases, and the precision remains 66% when recall reaches 99.9%. As for real-world binary analysis tools such as IDA Pro, Angr, GHIDRA, etc., they identify icall targets by constant propagation. Although constant propagation can avoid false positives, i.e. has a 100% precision rate, it can only resolve very few targets and has high false negatives, i.e. has a recall rate close to zero, and thus has an F1-measure close to 50%. For icallsites of subject binaries in Table 5, constant propagation can at most recognize 8 targets in `403.gcc`, and cannot recognize any target(s) in over half of the binaries.

We also calculate the average indirect call target (AICT) metric that TypeArmor and BPA used, and the results are shown in Table 5. Additionally, we include a source-level type analysis solution LLVM-CFI [15] as a reference. Column **#Functions** indicates the number of all functions in a binary, and columns **#iCallsites** and **#AT** indicate the number of icallsites and address-taken functions respectively. For fair comparison, we use the set of icallees when CALLEE's recall is 99.9%. We assume the recovering results of TypeArmor are absolutely correct, though the accuracy of TypeArmor in identifying argument numbers is about 83%, and much lower in identifying the usage of return value (less than 20%). Nonetheless, it shows that CALLEE has smaller AICTs than state-of-the-art solutions on most binaries, and can reduce 40.1% icallees than TypeArmor on average, which is better than BPA and the refinement solution. While as expected, LLVM-CFI still outperforms CALLEE, since it is a source-level solution which could utilize function type information.

TABLE 5: AICT evaluation results. **#AT** indicates numbers of address-taken functions and **#CP** indicates numbers of callees found by constant propagation.

Binary	#Functions	#iCallsites	#AT	#CP	AICT			
					TypeArmor	BPA	CALLEE	LLVM-CFI
nginx	1118	220	744	4	420.5	525.1	383.0	21.5
lighttpd	360	56	279	0	24.7	33.9	31.7	7.0
exim	622	78	344	0	38.0	30.6	22.4	5.7
memcached	244	50	109	0	21.6	1.4	11.3	1.1
400.perlbench	1793	117	664	6	536.6	363.7	354.0	24.0
401.bzip2	79	22	2	0	1.0	2.0	1.4	1.0
403.gcc	4678	44	1050	8	581.3	427.8	338.0	9.3
433.milc	245	6	3	0	2.0	2.0	2.0	2.0
445.gobmk	2537	46	1672	1	1,413.3	1,297.2	672.4	600.9
456.hmmer	506	12	20	1	22.0	2.8	7.2	10.0
458.sjeng	145	3	8	0	7.0	7.0	7.0	7.0
464.h264ref	533	354	40	0	28.9	26.4	20.9	2.1
482.sphinx	336	10	7	0	1.9	0.7	5.6	5.0
Average	1,015.1	78.3	380.2	1.5	-	-	-	-

6.2.3. Ablation Studies. To evaluate how key parts of CALLEE influence the performance, we perform ablation studies of transfer learning, slicing, symbolization and feature extraction layers of the Siamese network.

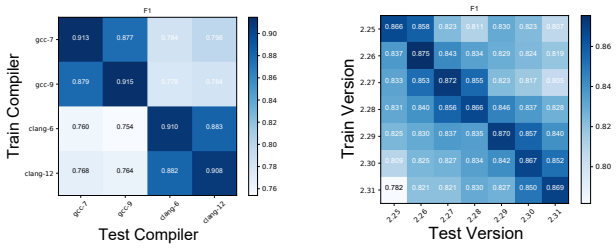
Effect of Transfer learning. To evaluate the effect of transfer learning, we perform model training with 4 modes: training and testing with dcall and icall datasets respectively, training on dcall dataset first and fine-tuning with icall dataset (i.e. transfer-learning), and training on dcall dataset and testing on icall dataset (i.e. zero-shot learning). Table 4 shows that merely training with icall dataset can only achieve a 66.8% *F1* on the test set, and meanwhile suffers from over-fitting (*F1* drops 9.4% from training to testing). While transfer-learning can boost the *F1* during testing to over 94%. Even in the zero-shot learning setting, where we test the pre-trained dcall model with the icall dataset without fine-tune, the *F1* can still reach 89%, indicating that dcall and icall pairs can share many common patterns, and thereby transfer-learning can greatly improve CALLEE's performance.

Effect of Slicing. To evaluate the effect of slicing, we first fixate other parts of CALLEE. Based on the icall dataset, we compare two situations: full context and sliced context (Settings 1, 4 in Table 4). As shown, the model trained with full context suffers from severe over-fitting: *F1* drops 18.4% from training to testing, showing that processing binaries with slicing could greatly help the Siamese network comprehend the context. It also indicates that full contexts of one binary can significantly differ from those in another binary, considering that different binaries in the icall dataset are compiled with different compilers and there is manually written assembly code in the Linux kernel. Therefore the network overfits code patterns in training binaries. Whereas performing slicing can "uniform" the assembly context from different sources, and thus can restrain the overfitting.

Effect of Symbolization. Similarly, we fixate the Siamese neural network (FCN feature extraction layers) of CALLEE, and compare different symbolization policies on the icall dataset (Settings 1, 5 in Table 4). As shown, strict symbolization has worse performance than loose symbolization. It confirms that the strict symbolization discards too much data-flow information, as discussed in Section 4. Additionally, the performance of strict symbolization degrades steeply (15.6% *F1*) from training to testing, which means that strict symbolization leads to worse over-fitting. In other words, strict symbolization leads to poor generalization performance. *Therefore, embedding with loose symbolization could better preserve data-flow information.*

Effect of Feature Extraction Layers of the Siamese Neural Network. We have tested the performance of Siamese networks with different feature extraction layers on the icall dataset (Settings 1, 6, 7, 8 in Table 4). The FCN we test has 3 hidden layers with 512 neurons. The LSTM model has 512 neurons. The 1dCNN has 1 convolutional layer with 512 filters. The TextCNN is adopted from [74]. We use ReLU as the activation function for these models. As shown, Siamese networks with FCN layers have the best performance, achieving an *F1* of 66.8%. TextCNN layers perform slightly worse than FCN layers, with an *F1* of 66.1%. 1dCNN layers perform best on the training set but have the worst overfitting, leading to relatively poor performance on the testing set. The *F1* drops 11.6% from training to testing. LSTM layers have the worst performance. One explanation is that Recurrent Neural Networks such as LSTM usually take longer to converge due to the vanishing and exploding gradient problems [75], even if LSTM tried to ease gradient problems by introducing gates [63]. Overall, we choose FCN layers as feature extraction layers.

6.2.4. Generalization across Compilers and Program Versions. Apart from the generalization ability across binaries, we also evaluate the generalization ability across compilers and program versions. The zero-shot learning results have shown that icall pairs share common patterns with direct ones, and we thus believe they have common behavior in generalization. Therefore, we perform experiments based on the large-scale dcall dataset. Specifically, we build 7 versions (from 2.25 to 2.31) of GNU Binutils with



(a) Cross-Compiler (b) Cross-Version

Figure 5: Generalization performance on GNU Binutils.

4 compilers (gcc-7, gcc-9, clang-6, clang-12) and further extract dcall pairs from them. To evaluate the generalization ability across compilers, we train the model on pairs from binaries compiled with one compiler (e.g. gcc-7) and test on pairs from binaries compiled with another (e.g. gcc-9). Figure 5(a) shows the *F1* of the cross-compiler setting. Data in the diagonal line indicates the upper limit of the model, where training set and testing set are the same. In most difficult scenarios such as clang-12 vs gcc-7, whose generated assembly can have huge differences, the model can still achieve an *F1* of 76%, and in easier scenarios such as gcc-7 vs gcc-9, the model achieves a substantial performance with only a 2%-4% drop of *F1*. The model behaves likewise in the cross-version setting, as shown in Figure 5(b). Across two most different versions 2.31 vs 2.25, to which 44 contributors have pushed over 5,000 commits [76], the model still achieves a 78% *F1*. Thus CALLEE has a substantial generalization performance in both cross-compiler and cross-version settings.

6.2.5. Time Efficiency. Suppose a binary has M icallsites and N candidate callees, CALLEE pair the callsites with each candidate callee and output a score for each input pair, so the time complexity is $O(MN)$. However, only address-taken functions are considered as possible candidates. And modern machine learning frameworks such as PyTorch provide batch inference, which takes advantage of scalable computation resources to generate many predictions at once. Suppose the batch_size is B, the time complexity will be $O(\frac{MN}{B})$. Ideally, if the RAM is sufficient to load all pairs, i.e. $B{=}MN$, the model only needs to infer once.

After the one-time-effort pre-train, we measure the time consumption of key parts of CALLEE with merely CPU. It takes about 23s to fine-tune the doc2vec model and 2,407s to fine-tune the Siamese network. After fine-tuning, on average, it takes about 0.0027s to perform slicing for a callsite-callee pair, 0.0042s to embed a slice with the doc2vec model and 0.0011s to infer one pair with the Siamese network. For binaries in Table 5, it takes 4~30 seconds in total to analyze a binary with CALLEE and 6~45 seconds with TypeArmor. However, as a pointer analysis, BPA needs more than 100 seconds to analyze small programs such as `lighttpd`, and more than 7 hours to analyze large programs like `gcc`.

In summary, we could draw the following conclusion: CALLEE *is more efficient and effective at recognizing*

icallees than state-of-the-art solutions such as BPA, TypeArmor as well as binary analysis tools.

6.3. Applications of CALLEE

6.3.1. Promoting binary similarity detection. With the final network trained and fine-tuned with pairs of all optimization levels, we utilize CALLEE to promote a fundamental task in binary similarity detection: binary diffing.

The state-of-the-art solution DeepBinDiff [6] leverages the program-wide control flow information to generate basic block embeddings. Specifically, it relies on an interprocedural CFG (ICFG) generated by Angr, which is a combination of CGs and CFGs, to provide program-wide contextual information. Given two binaries, DeepBinDiff first generates an ICFG for each binary, merges them based on library functions, and runs the Text-associated DeepWalk (TADW) algorithm [77] to embed basic blocks. With generated embeddings, DeepBinDiff utilizes a k-hop greedy matching algorithm to match basic block pairs. In principle, if two icallsites in two binaries have similar callees, the two basic blocks they belong to should be similar too. Therefore, we can speculate that, *with the CGs recovered by* CALLEE*, DeepBinDiff would have better performance.*

Our experiments are performed on the same set of binaries used by DeepBinDiff, i.e., `printenv`, `md5sum`, `split`, `uniq`, `ls`, `who`, `cp`, `rmdir`, `yes`, `tty` from five versions of GNU Coreutils (v5.93, v6.4, v7.6, v8.1, v8.3) with four optimization options (O0, O1, O2, O3). The binaries are compiled with the same compiler Clang, and we adopt the same metric used by DeepBinDiff, which is Precision, Recall, and F1-score of basic block matching. Parameters of DeepBinDiff are fixed to *k=4*, *threshold=0.6*, which are the optimal parameters according to their paper. To eliminate the influence introduced by randomness in TADW, we repeat each experiment three times and calculate the average metrics.

We compare the performance of DeepBinDiff in diffing binaries across different versions and optimization levels, based on the original CGs and the CGs recovered by CALLEE respectively. To further verify the usefulness of CGs recovered by CALLEE, we also tested DeepBinDiff on crafted CGs that are generated by adding random edges between icallsites and potential callees.

Cross-optimization-level diffing. Table 6 shows the F1-scores of cross-optimization-level diffing. We compile Coreutils-v7.6 and setup 6 experiments (O3 vs O2, O3 vs O1, O3 vs O0, O2 vs O1, O2 vs O0, O1 vs O0). As shown, compared to the original CGs, adding random edges would cause DeepBinDiff drop a 1.9% F1-score (i.e., from 44.8% to 42.9%) on average, while adding edges recovered by CALLEE would cause DeepBinDiff to increase the F1-score by 13.7% (i.e., from 44.8% to 58.5%) on average. Detail statistics of the F1 scores of DeepBinDiff in different settings on different binaries are presented in Appendix B.

TABLE 6: Cross-optimization-level binary diffing F1 scores of DeepBinDiff on the original CGs, on CGs with random edges, and on CGs recovered by CALLEE.

Optimization Levels	DeepBinDiff	+Rand	+CALLEE
O3 vs O2	89.0%	85.3%	**93.7%**
O3 vs O1	69.7%	67.8%	**78.4%**
O3 vs O0	10.8%	9.3%	**25.6%**
O2 vs O1	74.5%	72.0%	**92.1%**
O2 vs O0	11.2%	9.9%	**28.6%**
O1 vs O0	13.7%	12.8%	**32.6%**
Average	44.8%	42.9%	**58.5%**

TABLE 7: Cross-version Binary Diffing Results.

Versions	DeepBinDiff	+Rand	+CALLEE
v5.93 vs v8.3	72.5%	70.6%	**78.2%**
v6.4 vs v8.3	75.9%	73.3%	**85.8%**
v7.6 vs v8.3	95.5%	93.3%	**96.7%**
v8.1 vs v8.3	97.1%	94.6%	**98.8%**
Average	85.3%	83.0%	**89.9%**

Note that, adding random edges decreases all settings' F1-scores, because it would significantly change the contexts of basic blocks that ought to be similar. Whereas adding edges recovered by CALLEE increases all settings' F1-scores, showing that precise CGs are useful for binary diffing and CALLEE is effective at recovering CGs.

Cross-version diffing. Table 7 shows the F1-scores of cross-version diffing. We fix the Coreutils' optimization level to O1, and perform 4 experiments (v5.93 vs v8.3, v6.4 vs v8.3, v7.6 vs v8.3, v8.1 vs v8.3). Compared with DeepBinDiff, adding random edges leads to a 2.3% F1-score decrease on average, while adding edges recovered by CALLEE increases the F1-score by 4.6% on average. Detailed statistics in different settings on different binaries are presented in Appendix B. Consistent with the cross-optimization-level diffing results, we can see that, adding random edges decreases all settings' F1-scores and adding CALLEE edges behaves in contrast.

Additionally, the evaluation shows that, compared with cross-version diffing, cross-optimization-level diffing is more difficult, and larger increments appear in the cross-optimization-level settings involving the O0 level, i.e. O3-O0, O2-O0, O1-O0, compared with other settings. It indicates that optimization levels' effect is larger than versions', which is consistent with conclusions of DeepBinDiff and BINKIT [78]. Thus we can obtain larger promotion in cross-optimization-level diffing by complementing the ICFG.

In summary, CALLEE can improve the performance of DeepBinDiff by a large margin, especially in the cross-optimization-level diffing task.

6.3.2. Promoting hybrid fuzzing. We further apply CALLEE to hybrid fuzzing. Driller [31] is a hybrid fuzzer that augments the famous grey-box fuzzer AFL [41] with symbolic execution when AFL gets stuck. Specifically, driller takes all untraced paths in AFL's queue and looks for basic block transitions AFL failed to find satisfying inputs for. Driller will then use Angr to solve inputs for these transitions and pass them to AFL. However, driller does not monitor transitions invoked by icalls, and thus *we could speculate that augmenting driller with the CGs recovered by*

TABLE 8: Hybrid Fuzzing Results.

Challenge	# Paths			Found Crash?		
	Driller	+Rand	+CALLEE	Driller	+Rand	+CALLEE
NRFIN_00026	26	25	20	✘	✘	✘
LUNGE_00002	39	37	120	✓	✓	✓
YAN01_00007	45	45	125	✘	✘	✘
NRFIN_00074	412	404	489	✓	✓	✓
KPRCA_00017	246	221	283	✘	✘	✓
KPRCA_00003	11	10	9	✓	✓	✓
KPRCA_00060	140	113	394	✓	✓	✓
NRFIN_00076	45	42	47	✘	✘	✘
Average	120.5	112.1	185.9	-	-	-

CALLEE *can help driller cover more paths, i.e., improve the code coverage.* We have modified driller to solve symbolic constraints to generate testcases when it hits an icall.

Our experiments are performed on the same binaries used by driller, i.e., the DARPA CGC chanllenges [32]. We choose all 8 challenges that involve icall in the code, and fuzz the binaries for 24 hours, Experiments are repeated 3 times and we calculate the average number of results. We compare the number of triggered paths of each challenge between the vanilla driller and driller with icall resolving based on the CGs recovered by CALLEE. Analogically, we also include a driller with icall resolving based on the CGs with added random edges.

As shown in Table 8, on average, adding random edges to CGs decreases the number of paths by 8, while adding edges recovered by CALLEE can increase the numbers by over 50%, because adding random edges could misguide the symbolic execution engine to solve unreachable edges. Note that the fuzzer may spend more time solving symbolic constraints introduced by icalls than conditional branches, known as the exploration-exploitation trade-off problem, as shown in `NRFIN_00026` and `KPRCA_00003`. But overall adding edges recovered by CALLEE can increase the code coverage, demonstrating the effectiveness of CALLEE. We additionally examine the crashes found by the fuzzers, and results show that adding icall edges can also benefit slightly. For example, on the `KPRCA_00017` challenge, vanilla driller and driller+Rand failed to trigger crash within 24 hours while driller+CALLEE can.

In summary, we could draw the following conclusion: CALLEE *can promote binary similarity detection and improve the code coverage in hybrid fuzzing.*

6.4. Interpretability of CALLEE

To examine whether CALLEE has learned interpretable knowledge, we visualize the embedding model as well as the weights of the Siamese neural network.

6.4.1. Embedding Model. We use T-SNE [79] to project high-dimensional vectors to a 2D space to examine whether the embedding model could group semantically-close tokens together. There are 3,330 tokens after Loose symbolization. The smaller the distance between tokens, the more similar their semantic features are. For example, token `jb` and `jnb` are both instructions related to conditional jump, so they are clustered together in Figure 6. Therefore, word vectors generated by the doc2vec model can well capture semantic features of tokens in assembly instructions.

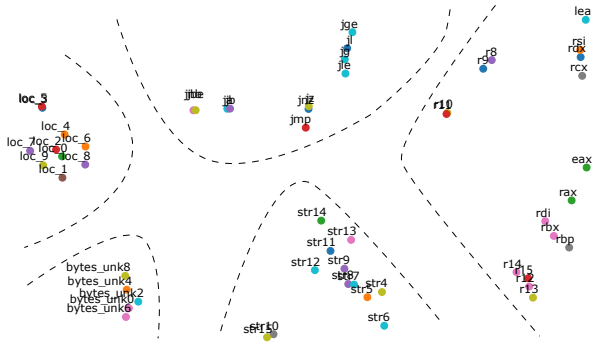


Figure 6: T-SNE visualization of tokens in doc2vec

6.4.2. Siamese network. We utilize the saliency map to interpret the network to deduce the sensitivity of output regarding input vectors. First, we compute partial derivatives for input pairs. Given a callsite or callee slice (after vectorization) $x \in R^{l \times d}$, l is the length of the slice, and d is the dimension of a token's embedding. $f(x)$ is the output of the Siamese network. The partial derivatives is given by:

$$\nabla_x f(x) = \frac{\partial f}{\partial x} = [\frac{\partial f}{\partial x_{i,j}}]_{i \in 1...l, j \in 1...d}$$

This partial derivative consists of gradients of each input token. To measure the sensitivity of each token, we further compute the magnitude of gradient. The saliency map $S(x)$ is defined as:

$$S(x)[i] = \sqrt{(\frac{\partial f}{\partial x_{i,1}})^2 + (\frac{\partial f}{\partial x_{i,2}})^2 + ... + (\frac{\partial f}{\partial x_{i,d}})^2}$$

With the saliency map to interpret CALLEE, we present a case study of a pair from `lighttpd` on which CALLEE surpasses TypeArmor. With the help of debug info, we could map the assembly pair to source code: the callsite is `a->data[i]->fn->free(a->data[i])` in function `array_free_data`, and the callee is function `void array_data_string_free(ptr *p)`. However, TypeArmor wrongly reports the callee as "non-void" function, and thus could lead to type-matching mistakes. CALLEE predicted the pair as "match", and the saliency map is shown in Figure 7. In the saliency map, a token with darker color means a larger $S(x)[i]$, i.e. a greater contribution to model decision, according to the definition of saliency map. Thus in the slices of the callsite and callee, the most important tokens are all related to the argument register `rdi`, and meanwhile tokens concerning the return value register `rax` has little contribution. It demonstrates that the network indeed can capture important features of the calling convention. In other words, the network has learned patterns consistent with domain knowledge.

In summary, we could draw the following conclusion: *The embedding model reasonably represents tokens in a high-dimensional space, and the Siamese neural network can learn patterns consistent with domain knowledge.*

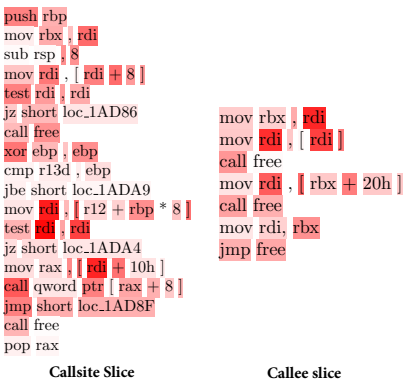


Figure 7: Saliency map of the pair from `lighttpd`.

7. Discussion and Limitations

Cross-optimization-level evaluation of CALLEE. Cross-optimization-level callsite-callee matching, e.g. training with GCC-O0 pairs only and testing with GCC-O3 pairs, is not common in production environments. Instead, CALLEE trains *one unified* model with pairs of functions that are compiled with all optimization levels before deployment to answer users' callsite-callee matching questions. Nevertheless, we have compared the performance of this unified model with the performance of multiple models for individual optimization levels, i.e., each model is only trained with pairs of functions compiled with the same optimization level, and results show that our unified model has very close performance (less than 1% F1). Thus we use the unified model for downstream applications.

Mechanism of neural networks. Although we have used T-SNE to visualize the distribution of token embeddings and calculated the saliency map of the Siamese network, CALLEE is designed to provide a *reference for*, rather than teaching human experts to analyze binaries, because the robustness of interpretation of neural networks has not been theoretically proved [80], and currently there is no standard method to interpret DNNs for binary analysis.

Indirect jumps. Currently, CALLEE only handles icalls and does not support indirect jumps. In general, indirect jumps are used for `switch` statements or tail calls. For the former, their targets can be recovered from the associated jump table generated by compilers [16]. For the latter, they are almost the same as icalls. Our solution could be extended to support them in the same way, i.e., slicing, preprocessing, embedding and matching with a Siamese network.

Variadic functions. Type-based solutions cannot well support variadic functions, i.e. functions with a variable number of arguments. While CALLEE matches callsites with callees by apprehending their contexts and has no requests on the arguments. As long as the instructions concerned with arguments are all kept in the context, the network can extract features automatically from the context.

Applicability to programs with other calling conventions or in other architectures or obfuscated. The software ecosystem has various calling conventions and architectures. For example, for 32-bit programs using the x86 cdecl calling convention, function arguments are passed via the stack. Another example is that, smart contracts written in Solidity run in a stack-based virtual machine. To apply CALLEE to programs with other calling conventions, one needs to adjust the current policies of slicing and symbolization. In the same way can one apply CALLEE to programs in other architectures or obfuscated ones. Overall, the idea of comprehending contexts of callsites and callees and matching them in a question-answering way is theoretically reasonable for all programs. We leave it as future work.

Applicability to tasks that require a 100% recall. Tasks such as Control-flow integrity (CFI) and binary rewriting usually require a 100% recall to avoid compatibility issues caused by false negatives. However, due to the random nature of neural networks, one cannot ensure neural networks achieve a 100% recall, therefore to apply CALLEE to those tasks, additional efforts are required to eliminate false negatives. Actually, even TypeArmor can have false negatives as well [24], and BPA achieves a 100% recall on top of binary profiling. Except for binary profiling, one can ease the false-negative problem by increasing the matching threshold, while introducing more false positives.

Working on assembly rather than IR. Lifting binaries to IR actually relies on indirect control-flow resolution [81]. Besides, existing binary lifting tools can generate redundant or even incorrect IR [82]. Therefore, we believe that lifting binaries to IR may lead to more information loss, enlarging the difficulty for neural networks to comprehend the context.

8. Conclusion

In this paper, we present CALLEE, a transfer- and contrastive-learning approach that effectively recognizes icallees at the binary level. By slicing the contexts of callsites and callees, CALLEE trains an assembly-centric doc2vec model and a Siamese neural network to match callsites with callees. Evaluation results show that, CALLEE can recognize icallees with high precision and recall, and can recover call graphs to promote downstream applications, e.g., binary code similarity detection and hybrid fuzzing. By interpreting the embedding model and the Siamese neural network, we demonstrate that CALLEE learns knowledge similar to human experts, and thus can apprehend the assembly language to some extent. Therefore, we believe that transfer-learning approaches are promising for binary program analysis tasks.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their insightful feedback, especially the suggestion of transfer learning. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000, 2021YFB3101200), National Natural Science Foundation of China (61972224, 62272265, U1836213), and Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 693–706.

[2] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "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*, 2017, pp. 363–376.

[3] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *Network and Distributed System Security Symposium*, 2020.

[4] S. Chen, Z. Lin, and Y. Zhang, "Selectivetaint: Efficient data flow tracking with static binary rewriting," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[5] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

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

[7] T. Cloosters, M. Rodler, and L. Davi, "Teerex: Discovery and exploitation of memory corruption vulnerabilities in {SGX} enclaves," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 841–858.

[8] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 345–362.

[9] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 472–482.

[10] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[13] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[14] ——, "Value-flow-based demand-driven pointer analysis for c and c++," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 812–835, 2018.

[15] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[16] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[17] Hex-Rays SA, "IDA Pro: a cross-platform multi-processor disassembler and debugger." http://www.hex-rays.com/products/ida/index.shtml.

[18] NSA, "Ghidra Software Reverse Engineering Framework." https://ghidra-sre.org/.

[19] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[20] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 559–573. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.44

[21] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τ cfi: Type-assisted control flow integrity for x86-64 binaries," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 423–444.

[22] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[23] Y. Lin and D. Gao, "When function signature recovery meets compiler optimization," in *2021 IEEE Symposium on Security and Privacy*, 2021.

[24] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *Network and Distributed System Security Symposium*, 2021.

[25] L. Yu, K. M. Hermann, P. Blunsom, and S. Pulman, "Deep learning for answer sentence selection," in *NIPS Deep Learning and Representation Learning Workshop, Montreal*, 2014. [Online]. Available: http://www.dlworkshop.org/accepted-papers

[26] D. Wang and E. Nyberg, "A long short-term memory model for answer sentence selection in question answering," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 2015, pp. 707–712.

[27] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[28] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[29] Z. Li, X. Xie, H. Li, Z. Xu, Y. Li, and Y. Liu, "Cross-lingual transfer learning for statistical type inference," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[30] Mozilla , "Mozilla firefox," https://hg.mozilla.org/mozilla-central, accessed: 2020-04-24.

[31] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[32] D. DARPA, "Cyber grand challenge," *Retrieved June*, vol. 6, p. 2014, 2014.

[33] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 8342–8360. [Online]. Available: https://aclanthology.org/2020.acl-main.740

[34] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," in *IEEE S&P*, 2021.

[35] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in neural information processing systems*, 1994, pp. 737–744.

[36] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[37] S. Minaee and Z. Liu, "Automatic question-answering using a deep similarity neural network," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 923–927.

[38] M. Yu, W. Yin, K. S. Hasan, C. dos Santos, B. Xiang, and B. Zhou, "Improved neural relation detection for knowledge base question answering," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*,

2017, pp. 571–581.
[39] W. Zhao, T. Chung, A. Goyal, and A. Metallinou, "Simple question answering with subgraph ranking and joint-scoring," 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)*, 2019, pp. 324–334.
[40] Zynamics, "BinDiff." https://www.zynamics.com/bindiff.html.
[41] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018, online: accessed 01-May-2018.
[42] S. Poeplau and A. Francillon, "Symqemu: Compilation-based symbolic execution for binaries," in *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.
[43] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 611–620.
[44] X. Bai, L. Xing, M. Zheng, and F. Qu, "idea: Static analysis on the security of apple kernel drivers," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1185–1202.
[45] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints." in *Network and Distributed System Security Symposium*, 2019.
[46] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 149–165.
[47] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μrai: Securing embedded systems with return address integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.
[48] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2421–2436.
[49] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Network and Distributed System Security Symposium*, 2011.
[50] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux." in *Network and Distributed System Security Symposium*, 2018.
[51] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1205–1220.
[52] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.
[53] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.
[54] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "{X-Force}:{Force-Executing} binary programs for security applications," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 829–844.
[55] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 611–626.
[56] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.
[57] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT (1)*, 2019.
[58] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116.
[59] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[60] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "{DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1787–1804.
[61] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.
[62] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.
[63] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[64] Intel Inc., "Processor tracing," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.
[65] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Conf. on Computer and Communication Security*, 2013.
[66] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface," *AMD64 Architecture Processor Supplement*, 2018.
[67] X. Li, Q. Yu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," *arXiv preprint arXiv:2103.03809*, 2021.
[68] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1735–1742.
[69] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015, pp. 1–11.
[70] IDAPython Team, "Idapython project for hex-ray's ida pro," https://github.com/idapython/src.
[71] Mozilla, "Mozilla top contributors," https://support.mozilla.org/en-US/community/top-contributors/questions?product=firefox, 2022, online: accessed 18-August-2022.
[72] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.
[73] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
[74] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.
[75] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
[76] GNU, "Gnu binutils diff," https://github.com/bminor/binutils-gdb/compare/68b975a...af127c2, 2022, online: accessed 15-August-2022.
[77] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, vol. 2015, 2015, pp. 2111–2117.
[78] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *Transactions on Software Engineering*, 2021.
[79] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
[80] A. Ghorbani, A. Abid, and J. Zou, "Interpretation of neural networks is fragile," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3681–3688.
[81] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida *et al.*, "Binrec: dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
[82] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *2017*

32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, pp. 353–364.
[83] Intel Inc., "libipt," https://github.com/intel/libipt.
[84] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
[85] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *KSII the 9th international conference on internet (ICONI) 2017 symposium*, 2017.
[86] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

Appendix

1. Callsite-callee pair collection

User-mode binaries. For user-mode binaries, we first turn off the Address Space Layout Randomization (ASLR) for convenience, then we have tried the following methods:

- LLVM. We instrument all indirect callsites by an LLVM machine pass. When compiling binaries, this pass identifies all indirect call instructions, and inserts a one-byte int3 instruction before them. We then write a debugger script to automatically catch breakpoints caused by this instruction and record runtime information, including callsite addresses, the callee addresses, and virtual memory maps of the binaries (to recognize addresses resided in shared libraries).
- Fuzzing & Intel Processor Tracing (PT). We first use coverage-guided fuzzers such as American Fuzzy Loop (AFL) [41] to get inputs that can cover as much code as possible. Then run the program with these inputs, and use Intel PT [64] to record execution traces. Finally, with the libipt [83] decoder library, we extract indirect call instructions from the trace, take their next instructions as targets and make pairs.

```
...
0x00000000fffee96a:  mov    eax,ebx
0x00000000fffee96c:  call   0xfffedac7

----------------
IN:
0x00000000fffedac7:  mov    ecx,edx
...
0x00000000fffedae8:  ret

----------------
IN:
...
0x00000000fffee979:  mov    eax,ebx
0x00000000fffee97b:  call   0xfffedac7

----------------
IN:
0x00000000fffee980:  mov    DWORD PTR [esp+0x4],eax
0x00000000fffee984:  mov    edx,ebp
...
```

Figure 8: Logging optimization of PANDA. Function call `call 0xfffedac7` is continuously invoked twice at address `0x00000000fffee96c` and address `0x00000000fffee97b`, but the function body (instructions) is only recorded once.

The Linux kernel. Likewise, we turn off Kernel Address Space Layout Randomization (KASLR) when compiling the kernel for the convenience of implementation. If KASLR is on, addresses recorded during runtime are complicated to be mapped back to the static addresses in the binary. Afterward, the kernel is emulated in an open-source record and replay platform PANDA [65], which is built upon the QEMU [84] whole system emulator. We enable the "`-d in_asm`" option of PANDA to log the target assembly code and instruction addresses.

Kernel traces are stored in a log file, from which we can extract the addresses of callsite-callee pairs. Usually, the next instruction of a callsite should be the target callee, however, there are two challenges in parsing the kernel trace log:

- Hardware interrupt. When a hardware interrupt is encountered right after an indirect call, we do not record the current pair, since we have no knowledge of hardware interrupts.
- Logging optimization of PANDA. As shown in Figure 8, when a function is invoked multiple times, PANDA may log function body texts only once in the trace. Hence we check indirect calls which are continuously invoked. To avoid false callees, we only record the target of the first indirect call (i.e. address of the first callsite's next instruction).

Rational behind the dynamic analysis to collect ground truths. Recall that data as ground truths should all be true positives. And an icall that can be invoked during runtime without violating sanitizers is always legitimate and thus dynamically collected icall pairs are all true positives. Although potential legitimate pairs might be missed during dynamic analysis, the collected ground truths are 100% accurate. Besides, although dynamically-collected icall pairs can be easy-to-trigger, it is orthogonal to the callsite-callee matching because the complexity of a callsite's control-flow constraints does not influence the validity of its callees.

2. Detail statistics of DeepBinDiff

The performance of DeepBinDiff depends on the call graphs it can get. In this section, we present the detailed F1 scores of DeepBinDiff on different binaries in different settings.

In the cross-optimization-level binary diffing setting, the F1 scores of DeepBinDiff based on the original CGs, CGs with random edges and CGs with edges recovered by CALLEE are shown in Table 9, Table 10 and Table 11 respectively.

In the cross-version binary diffing setting, the F1 scores of DeepBinDiff based on the original CGs, CGs with random edges and CGs with edges recovered by CALLEE are shown in Table 12, Table 13 and Table 14 respectively.

TABLE 9: Cross-optimization-level binary diffing F1 scores of DeepBinDiff, based on **original** CGs.

Optimization Levels	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
O3 vs O2	87.8%	89.4%	91.4%	87.8%	84.8%	91.9%	92.1%	90.7%	87.6%	86.8%	89.0%
O3 vs O1	72.7%	72.9%	75.0%	69.8%	60.5%	65.8%	72.4%	64.6%	72.0%	71.6%	69.7%
O3 vs O0	9.0%	13.2%	10.6%	14.0%	8.0%	11.0%	11.8%	8.0%	11.4%	10.6%	10.8%
O2 vs O1	78.4%	78.1%	79.4%	75.7%	67.8%	68.6%	74.8%	66.9%	77.0%	77.8%	74.5%
O2 vs O0	12.9%	11.5%	11.6%	14.7%	8.6%	9.2%	14.0%	7.4%	11.4%	10.3%	11.2%
O1 vs O0	13.4%	13.7%	14.2%	15.8%	9.8%	14.2%	14.1%	10.0%	15.8%	16.4%	13.7%
Average	45.7%	46.5%	47.0%	46.3%	39.9%	43.5%	46.5%	41.3%	45.9%	45.6%	44.8%

TABLE 10: Cross-optimization-level binary diffing F1 scores of DeepBinDiff, based on CGs instrumented with **random edges**.

Optimization Levels	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
O3 vs O2	83.8%	86.5%	87.1%	85.8%	81.4%	85.9%	87.9%	85.4%	84.1%	85.4%	85.3%
O3 vs O1	69.1%	71.7%	70.8%	70.1%	59.1%	64.7%	67.6%	62.6%	71.2%	71.5%	67.8%
O3 vs O0	8.1%	10.2%	8.8%	9.5%	7.8%	8.8%	11.8%	7.0%	11.8%	8.7%	9.2%
O2 vs O1	75.4%	74.1%	77.8%	73.9%	66.3%	66.2%	69.4%	67.1%	75.5%	74.6%	72.0%
O2 vs O0	10.9%	11.1%	9.4%	12.5%	8.3%	8.2%	10.3%	7.0%	10.3%	11.4%	9.9%
O1 vs O0	12.1%	14.1%	10.3%	15.6%	9.6%	14.7%	11.1%	8.8%	16.7%	15.3%	12.8%
Average	43.2%	44.6%	44.0%	44.6%	38.8%	41.4%	43.0%	39.6%	44.9%	44.5%	42.9%

TABLE 11: Cross-optimization-level binary diffing F1 scores of DeepBinDiff, based on CGs recovered by **CALLEE**.

Optimization Levels	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
O3 vs O2	89.7%	96.5%	99.0%	90.4%	93.0%	98.1%	96.1%	95.3%	89.4%	89.6%	93.71%
O3 vs O1	76.4%	76.0%	74.5%	76.2%	87.5%	81.5%	76.7%	81.6%	78.0%	75.4%	78.38%
O3 vs O0	27.7%	30.1%	25.6%	27.6%	18.7%	23.9%	28.7%	18.5%	27.1%	27.9%	25.58%
O2 vs O1	87.6%	93.6%	92.2%	92.7%	93.6%	94.3%	95.7%	97.6%	87.3%	86.1%	92.07%
O2 vs O0	26.8%	34.0%	28.3%	36.3%	20.4%	26.6%	30.4%	15.7%	32.7%	34.5%	28.57%
O1 vs O0	36.7%	33.3%	31.4%	37.0%	26.1%	32.3%	32.7%	24.7%	35.8%	35.7%	32.57%
Average	57.5%	60.6%	58.5%	60.0%	56.6%	59.5%	60.1%	55.6%	58.4%	58.2%	58.5%

TABLE 12: Cross-version binary diffing F1 scores of DeepBinDiff, based on **original** CGs.

Versions	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
v5.93 vs v8.3	61.7%	68.0%	74.3%	79.5%	76.5%	84.5%	75.5%	67.0%	68.2%	70.0%	72.5%
v6.4 vs v8.3	67.8%	77.2%	79.7%	82.2%	80.5%	87.3%	76.2%	69.5%	67.4%	71.4%	75.9%
v7.6 vs v8.3	92.5%	94.0%	97.0%	97.5%	94.5%	98.6%	93.7%	96.9%	94.7%	95.9%	95.5%
v8.1 vs v8.3	97.9%	97.9%	97.3%	98.4%	95.1%	96.7%	95.5%	97.6%	97.7%	97.2%	97.1%
Average	80.0%	84.3%	87.1%	89.4%	86.7%	91.8%	85.2%	82.8%	82.0%	83.6%	85.3%

TABLE 13: Cross-version binary diffing F1 scores of DeepBinDiff, based on CGs instrumented with **random edges**.

Versions	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
v5.93 vs v8.3	59.6%	68.9%	75.3%	79.1%	71.8%	82.5%	74.0%	59.7%	65.1%	69.8%	70.6%
v6.4 vs v8.3	65.5%	71.2%	72.5%	83.5%	78.9%	81.7%	73.2%	70.8%	66.7%	69.2%	73.3%
v7.6 vs v8.3	91.9%	92.8%	92.7%	96.4%	92.2%	96.7%	90.0%	93.4%	94.1%	93.1%	93.3%
v8.1 vs v8.3	97.0%	97.3%	92.6%	97.6%	93.8%	92.0%	92.3%	93.2%	95.8%	94.8%	94.6%
Average	78.5%	82.5%	83.3%	89.1%	84.2%	88.2%	82.4%	79.3%	80.4%	81.7%	83.0%

TABLE 14: Cross-version binary diffing F1 scores of DeepBinDiff, based on CGs recovered by **CALLEE**.

Versions	printenv	md5sum	split	uniq	ls	who	cp	rmdir	yes	tty	Average
v5.93 vs v8.3	65.6%	79.7%	81.4%	83.5%	87.4%	84.8%	80.7%	70.7%	73.7%	74.7%	78.2%
v6.4 vs v8.3	79.5%	84.2%	88.7%	90.4%	89.1%	93.1%	85.0%	83.4%	81.2%	83.0%	85.8%
v7.6 vs v8.3	93.5%	95.4%	98.6%	96.0%	99.0%	98.0%	97.7%	95.3%	97.0%	96.3%	96.7%
v8.1 vs v8.3	98.7%	98.7%	99.1%	98.3%	99.0%	99.1%	99.5%	98.0%	99.0%	98.9%	98.8%
Average	84.3%	89.5%	92.0%	92.0%	93.6%	93.8%	90.7%	86.9%	87.7%	88.2%	89.9%

3. Impact of embedding techniques

To evaluate the generalization ability of different embedding techniques, we report the zero-shot performance, i.e., precision, recall and F1 scores when training with dcall pairs and testing with icall pairs, of 5 common embedding methods: Instruction2Vec [85], word2vec, PalmTree, Asm2Vec, and doc2vec. Since Instruction2Vec and word2vec cannot generate embeddings for instruction sequences directly, we have averaged instruction/token embeddings to obtain the sequence embedding.

As shown in Table 15, word2vec has the worst performance because it does not consider the internal structures of instructions; Instruction2Vec achieves an acceptable performance for its fine-grained instruction embedding model. Asm2Vec [86] performs better than Instruction2Vec but worse than PalmTree and doc2vec becasue it generates instruction sequences by random walk, which may lead to illegitimate control-flow sequences and cannot guarantee code coverage. F1-scores of PalmTree and doc2vec are close, while PalmTree has a higher recall and doc2vec has a higher precision. However, since PalmTree is a transformer-based solution, it will have a relatively low runtime efficiency. Overall, we choose doc2vec as the embedding technique for CALLEE.

TABLE 15: Zero-shot evaluation of embedding methods.

Embedding	Precision	Recall	F1
word2vec	72.5%	78.4%	75.3%
Instruction2Vec	79.8%	82.7%	81.2%
Asm2Vec	92.6%	83.3%	87.7%
PalmTree	88.2%	90.1%	89.1%
doc2vec	93.0%	85.9%	89.3%