ICALLEE: Recovering Call Graphs for Binaries

Wenyu Zhu†, Zhiyao Feng†, Zihan Zhang†, Chao Zhang†, Zhijian Ou†, Min Yang*
† Tsinghua University, Beijing, China
* Fudan University, Shanghai, China

Abstract—Recovering programs’ call graphs is crucial for inter-procedural analysis tasks and applications based on them. The core challenge is recognizing targets of indirect calls (i.e., indirect callees). It becomes more challenging if target programs are in binary forms, due to information loss in binaries. Existing indirect callee recognition solutions for binaries all have high false positives and negatives, making call graphs inaccurate.

In this paper, we propose a new solution ICALLEE based on the Siamese Neural Network, inspired by the advances in question-answering applications. The key insight is that, neural networks can learn to answer whether a callee function is a potential target of an indirect callsite by comprehending their contexts, i.e., instructions nearby callsites and of callees. Following this insight, we first preprocess target binaries to extract contexts of callsites and callees. Then, we build a customized Natural Language Processing (NLP) model applicable to assembly language. Further, we collect abundant pairs of callsites and callees, and embed their contexts with the NLP model, then train a Siamese network and a classifier to answer the callsite-callee question. We have implemented a prototype of ICALLEE and evaluated it on several groups of targets. Evaluation results showed that, our solution could match callsites to callees with an F1-Measure of 93.7%, recall of 93.8%, and precision of 93.5%, much better than state-of-the-art solutions. To show its usefulness, we apply ICALLEE to two specific applications - binary code similarity detection and binary program hardening, and found that it could greatly improve state-of-the-art solutions.

I. INTRODUCTION

Indirect calls 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, indirect calls play an important role in program analysis and related tasks. One can complement Call Graphs (CGs) of programs by recognizing targets of indirect calls, 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 binary rewriting, αDiff [5] and DeepBinDiff [6] perform binary diffing with CG features, and TEEREX [7] requires precise CGs to perform symbolic execution. Conversely, imprecise indirect callee analysis will lead to obstacles in many applications, such as false positives in bug detection [8], [9], [10], path explosion in symbolic execution [11], [12], [13], code loss in deobfuscation, compatibility issues in binary rewriting and program hardening [14], [15], [16], [17].

In practice, it is common to utilize static analysis to infer indirect callees, because dynamic techniques have many false negatives due to poor code coverage, though they can precisely identify indirect callees. Given target programs with or without source code, applicable static analysis solutions are different. When the source code is given, points-to analysis [18], [19] and type-based analysis [20], [21] are the most common methods. If only binaries are given, statically determining targets of indirect calls 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 indirect callees. For instance, binary analysis tools that are widely used in practice (e.g., IDA Pro [22], Angr [13] and GHIDRA [23]) and PathArmor [24] identify indirect callees by constant propagation, and can only resolve very few targets. On the other hand, CCFIR [25] adopts the address-taken policy and treats all address-taken functions as potential targets for indirect calls, thus having high false positives. 1-CFI [26], TypeArmor [27] and its refinement [28] reduce targets of indirect calls 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 [29] performs a delicate pointer analysis based on a block memory model (not applicable to C++) and a special intermediate representation language (with only support for x86) to infer indirect callees, but still has relatively low precision. A better solution to recognize indirect callees in binaries is therefore demanded.

In this paper, we propose a DNN-based solution ICALLEE to recognize indirect callees at the binary level. Given an indirect callsite, ICALLEE will answer which callees could be its potential targets. The key insight is that, neural networks can learn to match callsites and callees by comprehending their contexts, i.e., instructions nearby callsites and of callees. Therefore, to build such a network, we employ three core modules, i.e., context extraction module, context embedding module, and callsite-callee matching module.

First, we utilize slicing to extract instructions related to callsites and callees, and pay specific attention to elements (e.g., argument registers, stack memory operations, return values, etc.) concerning function calling conventions. These slices determine the expectation of callsites and the semantics of callees, and thus could be utilized as contexts to determine whether a callee matches a callsite.

Second, we train a model to embed the slices into feature vectors, by adjusting a representation learning solution widely used in the NLP (Natural Language Processing) field and applying it to the assembly language. Specifically, we construct a large corpus of binary functions by collecting binaries from several datasets and large-scale programs, and train an assembly-centric doc2vec model [30] with the corpus to embed slices we collected.

Third, we train a Siamese neural network to perform
callsite-callee matching. This problem is similar to classic question-answering scenarios [31], [32]. A common solution is the Siamese neural network [33], [34], which has two parallel feature extraction layers to process questions and answers respectively and a following classifier to match them. Analogically, iCALLEE regards a callsite as a question and a callee as its corresponding answer, and uses a Siamese network to match them.

We have implemented a prototype of iCALLEE and evaluated it on several groups of targets. Specifically, to measure the generality of our model, we compile source code of certain projects with different compilers and various compiler options, to acquire diversified binaries to test. We also evaluate our model on ground-truths dynamically collected from large real-world programs such as the Linux kernel and the Firefox browser [35]. The evaluation results show that iCALLEE could match callsites to callees with an F1-Measure (F1) of 93.7%, recall of 93.8%, and precision of 93.5%, much better than BPA [29], TypeArmor [27] as well as real-world binary analysis tools IDA Pro [22], Angr [13] and GHIDRA [23].

Further, we have demonstrated that applications relying on inter-procedural analysis can benefit from call graphs recovered by iCALLEE. Specifically, we applied iCALLEE to two specific applications. First, it is applied to the binary code similarity detection application, 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% recall in cross-optimization binary diffing. Second, iCALLEE is applied to the binary program hardening application, and greatly narrows down the number of candidate callees (i.e., potential hijacking operations’ landing locations) of indirect calls, i.e., 30% less than BPA [29], 80% less than τCFI [26], 72% less than TypeArmor [27] and 37% less than its refinement [28].

Additionally, we have made an attempt to interpret the neural network. It showed that, the proposed NLP model can well capture semantic features of tokens in assembly language, and DNNs can learn the features of operands of instructions in binaries can be viewed as tokens, which is consistent with the domain knowledge of binary analysis.

In summary, we make the following contributions:

- We present the first neural network based approach iCALLEE to recognize indirect callees for binaries and recover their call graphs.
- We demonstrate that, NLP techniques can learn features of the assembly language, and DNNs can learn the features of indirect function calls at the binary level.
- We evaluate iCALLEE with several groups of real-world programs, and reach up to 93.7% F1-Measure, 93.8% recall, and 93.5% precision on the callsite-callee matching task, much higher than existing solutions.
- We demonstrate that iCALLEE is highly effective at promoting tasks based on inter-procedural analysis, e.g., binary code similarity detection or program hardening.

II. BACKGROUND AND RELATED WORK

A. Program Slicing

Program slicing is a method used for extracting sequences of statements that have control or data dependencies with the target criteria [36]. By customizing the criteria, analysts can comprehend the program from different perspectives with slicing. Furthermore, it can be used in program differencing [37], optimization [38] and software security [39], [40].

Slicing techniques [41] can be divided into many categories. They can be divided into static [36] and dynamic slicing [42], whether or not considering influences of specific program inputs during slicing. According to the direction of control and data dependence analysis, they can be divided into backward [36] and forward slicing [43]. According to the analysis scope, they can be divided into intra-procedural slicing [36] and inter-procedural slicing [44].

B. Program Embedding

Program embedding refers to representing a program with a feature vector. Once the embedding correctly captures the semantic features of a program, we can use machine learning solutions to process the embedding vector instead of the original program in order to analyze this program.

Existing program embedding methods could be divided into two categories, i.e., dynamic and static. Dynamic program embedding methods will execute the target program to collect its input and output pairs, often used in fuzzing [45] and binary similarity detection [46], [47]. However, the effectiveness of these methods depends largely on the coverage of the training data of input and output, which are very difficult and time-consuming to collect. On the other hand, static program embedding methods [48], [49] often extract features, e.g., abstract syntax trees, control flow graphs and call graphs, from programs via static program analysis. However, it is very hard to learn precise semantic and behavior information from these static features. And thus static program embedding in general is not fit for machine learning solutions.

Nowadays, representation learning algorithms like doc2vec [30] and word2vec [50] are introduced to program embedding, and have been proved effective and efficient on program analysis [47], [51]. Analogous to words in text analysis, opcodes and operands of instructions in binaries can be viewed as tokens, which could be converted into unique vectors using doc2vec. As more applications of representation learning in program analysis are explored, many new learning methods tailored for programs have emerged. Most of them are refinements to the classic representation methods [49], [52], [48].

C. Siamese Neural Networks

Siamese network [53] is a structure with two parallel networks to extract input features, concatenate them and generates one output vector. At first, the Siamese network was proposed to compare the similarity of two inputs. It consists of two identical networks with identical structure and weights, which read two inputs respectively and extract their feature vectors. Distance between the two feature vectors will be calculated and used as the similarity/difference score. Previous studies such as oDiff [5] and NMT [47] 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 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, and the extracted feature vector q and a will be concatenated together as a feature vector f, which will be fed into a following classifier 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 [54], [55], [34].

D. Applications based on Call Graphs

Program analysis applications often have to track data flow between functions in order to comprehend the semantics of programs, and thus have to conduct inter-procedural program analysis. Such analysis tasks will traverse Call Graphs (CGs) of programs which represent calling relationships between functions to track information flow or capture the semantics. Such applications include but not limited to the followings.

Binary Similarity Detection. BinDiff [56] 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.

Program hardening. MULTIVERSE [15] relies on CGs to implement a shadow stack by inserting instructions for every call and ret instruction and allocating memory. IFCC [20] generates legitimate callee tables according to CGs and instruments programs to determine whether indirect callees are legal during runtime. Other Control-flow Integrity (CFI) solutions, such as τCFI [26], TypeArmor [27] and its refinement [28] also need precise CGs, and they also rely on CG recovery as well, as discussed in the next section.

Binary Rewriting. SelectiveTaint [4] generates a function clone for every acyclic path through a CG, when performing context-sensitive value-set analysis to identify instructions to be instrumented, and then uses static binary rewriting to instrument taint analysis logic.

Malware Detection. SMIT [57] establishes a malware database by representing each malicious program in terms of its CG, and translates the problem of finding a malware sample’s closest kin in a malware database into one that searches for a graph’s nearest neighbor in a graph database.

Directed Fuzzing. Directed grey-box fuzzers such as AFLGo [58], HawkEye [59] and CAFL [60] all rely on CGs to calculate target site distances of the basic blocks for instrumentation. AFLGo calculates Dijkstra shortest distance on the CG, HawkEye further assigns adjacent-function distance as the weight of the edges in the CG. As for CAFL, it calculates the target site distances and inserts checkpoint calls by recursively crawling up the CG.

Bug Detection. iDEA [61] builds ICFGs based on CGs starting from each entry point of a driver. Along the ICFG, iDEA invokes detection functions of a checker before and after each instruction to detect security bugs in Apple drivers. NEUEX [62] queries the CG to identify all the candidate vulnerability points which are statically reachable from the latest symbolic state. Similar dependencies also exist in other bug detection solutions, e.g., [8], [9], [10].

Except for aforementioned applications, CGs are also vital in symbolic execution [7], [11], [12], [13], kernel exploitation [63], [64], and many other scenarios [65], [66], [67].

The completeness and accuracy of CGs therefore greatly affect the results of these applications. Otherwise, it may cause issues like: false positives in bug detection, path explosion in symbolic execution, code loss in deobfuscation, compatibility issues in binary rewriting and program hardening.

E. Recovering CGs via Recognizing Indirect Callees

At the core, constructing a complete and accurate CG requires to precisely recognize targets of indirect calls. Many solutions have been proposed to address this problem. But few can recognize indirect callees for binaries.

Type-based Analysis. If the source code of a program is available, then type-based analysis [20], [68], [69], [21] is the first choice. However, type information is not available for binary programs such as most commercial off-the-shelf (COTS) software. Thus, identifying indirect callees in binary programs in general requires type recovery analysis [70], as shown in τCFI [26], TypeArmor [27] and its refinement [28], which is error-prone. Otherwise, a coarse-grained address-taken policy will be applied, as shown in CCFIR [25], in which arbitrary address-taken functions are marked as legitimate targets of indirect calls, causing high false positives.

Pointer Analysis. SVF [18] 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. Whole-program analyses such as SVF and SUPA [19] are non-scalable when applied to programs composed of separately compiled modules. Ptr-Tracker [71] uses heap graphs for pointer analysis in the context of bug finding, but does not guarantee soundness. K-Miner [72] splits kernel code based on system calls, but provides poor soundness. PeX [73] leverages the common programming paradigm used in kernel abstraction interfaces, but only applicable to kernels. VIP [74] turns to type-based pointer analysis to handle separately compiled modules in C++ programs, and faces the same issues of type-based solutions: imprecision and unsoundness. Some binary analysis tools such as BAP [75] and Angr [13] leverage value-set analysis to resolve pointers, but cannot scale to large programs. Recently, BPA [29] adds scalable pointer analysis support for binaries based on a special block memory model and IR, which is only applicable to x86 programs written in C.
F. DNN-based Program Analysis

DNNs have been proved efficient in various recognition and regression tasks, e.g., image recognition and machine translation. Recent research has leveraged DNNs to solve many program analysis problems.

Function Recovery. ByteWeight [76] shows 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 [77] improves the performance by applying a BERT [78] model. EKLAVYA [51] and StateFormer [79] further recover 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 [79] utilizes transfer learning with a transformer [80] model to learn type inference rules. However, they both cannot recover the signature of a callsite, and thus cannot recognize indirect callees.

Value-set Analysis. DEEPVSA [81] utilizes DNNs to facilitate value-set analysis by learning semantics of instructions and capturing dependencies in contexts at the binary level, and can further assist alias analysis for crash diagnosis. While the application in resolving indirect callees 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 [82] 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 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 [50] to embed instructions and use LSTM [83] to embed basic-blocks. The state-of-the-art DeepBinDiff uses both the code semantic information 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 targets of indirect calls, and utilize it to recognize indirect callees and recover CGs with high precision.

III. OVERVIEW

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

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. Therefore, to build such a network, we employ three major modules, i.e., context extraction module, context embedding module, and callsite-callee matching module. As a supervised learning solution, iCALLEE also has two phases: a learning phase and a recognition phase, as shown in Figure 2.

A. Core Modules

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, e.g., all instructions of a callee, 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.

2) Context Embedding: Since neural networks require vectors as inputs, contexts of callsites and callees have to be represented in the form of vectors. Note that, we perform the slicing algorithm on disassembly code, whose features are similar to those of natural languages. Therefore, we believe NLP solutions could be utilized to process slices as well. Existing study such as NMT also showed that NLP solutions are effective at binary analysis. We thus use a popular NLP model doc2vec to embed program slices.
3) Callsite-callee Matching: Inspired by question-answering scenarios, our solution iCALLEE regards a callsite as a question and a callee as its corresponding answer. To compute the difference score of a callsite and a callee, iCALLEE adopts a Siamese neural network. The network takes a pair of callsite vector and callee vector as input, and generates their feature vectors. Then, instead of computing the distance between them, these two feature vectors are concatenated together and fed into a classifier, which outputs the difference score of the input pair. A score close to 0 indicates that the callee is a possible target of the callsite.

B. Workflow

There are two phases of the overall workflow: the learning phase and the recognition phase.

1) The learning phase: The input to the learning phase is a large number of binaries (with collected ground-truth), and outputs of the learning phase are models that could be used to embed program slices and report difference scores. As highlighted in Figure 2, there are 5 steps in the learning phase.

* Step I.1: Collecting ground-truth callsite-callee pairs. We dynamically run several testing programs with provided benchmarks, and collect callsite-callee pairs at runtime. Specifically, we utilize Intel PT [84] to collect traces for user-mode binaries and utilize a record and replay framework PANDA [85] to collect information from kernel execution.

* Step I.2: Statically extracting callsite-callee pair slices and functions from binaries. Given collected ground-truths, we apply an inter-procedural slicing algorithm to extract slices for each callsite and its associated callee. We additionally extract slices of direct calls for robustness evaluation. Meanwhile, we build a function dataset from training binaries, in order to train an embedding model later.

* Step I.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, in order to make those models converge faster. Meanwhile, we also train a doc2vec model using the collected function dataset. The doc2vec model is then used to transform slices into vectors required by the following neural network.

* Step I.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.

* Step I.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.

2) The recognition phase: Given a target binary, we identify all indirect callsites and candidate callees, and extract their slices. The slices are then symbolized and transformed into vectors with the trained doc2vec model. Finally, the trained Siamese network predicts whether pairs are positive or negative. As shown in Figure 2, this phase has 4 steps.

* Step II.1: Extracting indirect callsite slices and candidate callee slices. This step analyses the target binary, identifies all indirect callsites and takes all address-taken functions as candidate callees. Afterward, we perform the same slicing algorithm as in Step I.2 to extract the corresponding slices.

* Step II.2: Slice Embedding. We preprocess slices corresponding to target callsites and callees in the same way as Step I.3 and embed them with the trained doc2vec model.

* Step II.3: Assembling candidate pairs to query. In this step, we assemble indirect callsite slice vectors and candidate callee slice vectors together, to create candidate pairs to query.

* Step II.4: Recognition. In this step, we query the trained Siamese neural network with candidate pairs to get their difference scores. Multiple positive pairs could share the same callsite, i.e., a callsite could have multiple callees.

### IV. Methodology

iCALLEE has three core modules, i.e., context extraction, context embedding and callsite-callee matching, and has to solve three essential challenges respectively. First, the context of callsites and callees should be represented in a proper granularity, to fit in the limited embedding space. We thus perform an inter-procedural slicing algorithm on input binaries. Second, slices need to be embedded into vectors but they may have many unseen tokens (e.g., constants or addresses) in the training data. We thus preprocess slices and symbolize some tokens in slices before embedding. Third, the performance of the Siamese neural network heavily depends on the feature extraction layers. We thus evaluate different layers to process embedded slices in Section VI.

#### A. Context Extraction via Slicing

Recent studies have shown that a deep neural network trained in a completely data-driven way without domain knowledge may be non-explainable and unpredictable. Its 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.

We identify and preserve instructions related to data dependencies between indirect callsites and callees, including local variables that passed between functions (arguments and return

| Data Type | Example | Passing |
|-----------|---------|---------|
| INTEGER, | char, short |Argument: rdi, rsi, rdx, rcx, r8, r9 |
| POINTER  | int, long | Return value: rax, rdx |
| SSE      | float, SSEUP | Argument: xmm0 to xmm7 |
| SSEUP    | double | Return value: xmm0, xmm1 |
| X87, X87UP | long double | Argument: stack |
| COMPLEX_X87 | Return value: zt0, zt1 |
| MEMORY   | struct, array, union | Argument: stack |
|          |         | Return value: (address in) rax |
values) and global variables. For global variables, all encountered instructions related to them need to be kept, because global variables are allowed to be set or used globally in the program. For local variables, we use rules of data passing [86] shown in Table I to determine whether an instruction is related to data passing between callers and callees, and then further decide to keep it or not.

To get as much information as possible, we perform a depth-first traversal of all basic blocks in caller and callee function’s control-flow graph (CFG), then analyze instructions within each basic block. We keep instructions whose operands are related to values in the data segment (for global data dependencies), registers used for function arguments and return values, and stack memory (for inter-procedural local data dependencies). To be conservative, we do not drop control-flow instructions.

### B. 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", ",]". Furthermore, when we embed instructions from a target binary, they may have tokens unseen in the trained doc2vec model, known as the Out-of-Vocabulary (OOV) phenomenon. To avoid this problem, we need to symbolize slices before embedding.

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 to a great extent. However, strict symbolization may lose data-flow information to a certain extent, which often contributes to the determination of the function call targets.

Hence we propose *loose symbolization* to preserve as much information as possible and meanwhile limit the size of the token corpus. Since it is reasonable to assume that tokens have limited possibilities in one binary, we can set an upper bound size for the token corpus. Through modulo arithmetic, an open set like `{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 direct call instruction are considered to be a function, and thus we replace them with "fun". Detailed rules of symbolization are shown in Table II.

2) **Vectorization:** After symbolization, iCALLEE 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 [30], and adjust it to apply to assembly language (V-C2). Note that, compared with word2vec, doc2vec is able to train 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.

Utilizing the binary analysis tool, IDA Pro, we collect abundant functions from various binaries (details in Section VI). After tokenization and symbolization, each function is regarded as a paragraph, and fed into the doc2vec model.

After vectorization, we concatenate vectors for each token in a paragraph. If the paragraph is too long to concatenate, we have to truncate it. Note that we have to ensure that callsite instructions are not lost after truncation.

### C. Structure of the Matching Network

With the embedded callsites and callees, we further build a Siamese neural network to predicate their difference score. First, the extracted callsite and callee slices are transformed into vectors by the trained doc2vec model. Second, we assemble positive and negative callsite-callee pairs using these vectors. Finally, the Siamese neural network takes vectors as input and is trained to output their difference scores (0 represents matched).

After extracting features, each parallel network will output a feature vector. Then we concatenate two feature vectors and use multiple fully-connected layers to derive the final difference score. The contrastive loss [87] is used as the optimization goal of our Siamese network, which is defined as follows.

\[
L = \frac{1}{2N} \sum_{n=1}^{N} [yd^2 + (1 − y) \max\{1 − d, 0\}^2]
\]

where, N is the number of input pairs, y (i.e., 1 or 0) is the label of the input pair (i.e., match or not), d is the distance or difference score output by the final layer. The optimization goal indicates that, if the input pair match (y = 1), then the output (difference score) 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 pair matches:

\[
\text{matching} = \begin{cases} 
\text{yes} & d < \text{threshold} \\
\text{no} & \text{otherwise}
\end{cases}
\]
V. IMPLEMENTATION

We establish datasets of iCALLEE based on Intel PT, IDA Pro etc., preprocess the data with IDA Pro, implement the doc2vec embedding model with gensim [88], and train the Siamese neural network with Keras [89].

A. Dataset Collection

iCALLEE needs two datasets in total. One function dataset for training the doc2vec model and one callsite-callee pair dataset for training the Siamese neural network. For the latter, we further denote indirect-call pairs as the ground-truth dataset, and additionally extract direct-call pairs to evaluate the influence of compilers and optimization levels. We collect binaries from various sources including the SARD dataset [90], the GNU Binutils [91], the GNU Coreutils [92], the Firefox browser [35] and the Linux kernel.

1) function dataset: 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 in the SARD dataset, GNU Binutils, GNU Coreutils, and Firefox. 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.

2) Callsite-callee pair dataset: 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 call sites and callees. While for indirect-calls, since it is difficult to recognize their targets statically, we additionally utilize dynamic methods because indirect-call pairs collected by them are 100% legitimate. Note that, although dynamically-collected indirect-call pairs can be easy-to-trigger, it is orthogonal to the callsite-callee matching task, because the complexity of a callsite’s control-flow constraints have no influence on the validity of its callees.

For user-mode binaries, we instrument all indirect call sites by an LLVM machine pass, which will output the callees at runtime. With coverage-guided fuzzers such as AFL [93], we can get inputs that can cover as much code as possible. Afterward, the indirect call site-callee pairs are collected by running the program with these inputs. For the kernel, it is emulated in an record and replay platform PANDA [94]. We enable the "-d in_asm" option of PANDA to log the target assembly code and instruction addresses. By parsing the log file, we can get the indirect callsite-callee pairs. For more details, please refer to Appendix A.

Besides, indirect-call pairs are much less than direct-call pairs, so we augment the ground-truth dataset with "noises", which are essentially useless instructions, e.g. nop.

B. Slicing

We implement the slicing algorithm with the IDAPython [95] 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 callsites and callee functions. Finally, we gather control-flow instructions, and the union of those parts is taken as the result.

C. Embedding

iCALLEE utilizes IDA Pro to disassemble instructions, so we take advantage of its naming rules to symbolize instructions. As for embedding, we adjust model parameters based on the difference between natural languages and assembly.

1) Symbolization: By default, IDA Pro names data structures automatically 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 func0 (strict) or func0 (loose), assuming that the hyper-parameter N is set to 10. As shown in Table II, we consider 12 situations in total.

2) Doc2vec: 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. iCALLEE adjusts two parameters of the doc2vec model during training.

One parameter is the 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.

The other parameter is the 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.

VI. EVALUATION

Here, we evaluate iCALLEE from the following aspects:

- Performance of indirect callee recognition. We discuss the efficiency of iCALLEE and how it is affected by key parts of iCALLEE, including the granularity of context, the intensity of preprocessing, parameters of the embedding model as well as the Siamese neural network.
- Applications of iCALLEE. We apply iCALLEE in the binary similarity detection scenario to see whether the state-of-the-art binary diff solution DeepBinDiff can benefit from CGs recovered by iCALLEE. We additionally evaluate the effectiveness of iCALLEE by applying it on reducing indirect call targets and comparing it with state-of-the-art solutions and binary analysis tools.
- Interpretability of iCALLEE. We interpret the deep-learning-based model used by iCALLEE from two aspects: distribution of token embeddings and feature weights of the Siamese neural network.

A. Evaluation Setup

Experiments are performed on a machine equipped with Ubuntu 16.04 LTS. The machine has an Intel CPU (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz), four NVIDIA GPUs (TITAN X (Pascal)) and 256GB RAM, and is installed with LLVM 10.0.0, GCC 9.2.1, libIpt 2.0.0 (commit 892e12c5), a docker image of PANDA (tag: 0729f0d4), IDA Pro 6.8, and Python 3.6. The Python 3.6 is equipped with gensim 3.8.1, TensorFlow 1.14.0 and Keras 2.3.1.

1) Datasets: We first collect binaries from various sources including the Juliet Test Suite for C/C++ v1.3, the GNU Binutils (v2.31), the GNU Coreutils (v8.31), the Firefox browser (v72.0a1) and the Linux kernel (v5.3.11).

We extract only direct callsite-callee pairs from the first three sources. Code from them are compiled with two different compilers (GCC and Clang) and four different optimization levels (from O0 to O3), and the binaries could have different numbers of callsites and callees. Afterward, we extracted functions from the binaries and perform the slicing algorithm.

We extract indirect callsite-callee pairs from the Firefox browser (including corresponding shared libraries) and the Linux kernel. The Firefox binaries are compiled with default options, of which different parts are compiled with different optimization levels. When compiling the Linux kernel, we additionally disable KASLR. After dynamic testing, we extract functions from the binaries and perform the slicing algorithm in the same way.

In summary, we collected 64,470 unique functions and 24,630 unique legitimate indirect callsite-callee pairs. As for direct calls, we collected 825,724 legitimate pairs in total. Different compilers and optimization levels result in different numbers of pairs, details are shown in Table V.

Data augmentation. To diversify the training data, we also augment the original dataset. According to [96], training with noise is equivalent to Tikhonov regularization. Analogous to augmenting image datasets, we can expand our ground-truths by adding "noise" instructions before embedding. We consider three kinds of "noise" instructions: the nop instruction, the align instruction, and instruction gadgets that do nothing meaningful, e.g., push r12; pop r12. We restrict the proportion of "noise" to 10% and triple the ground-truth dataset.

To ensure that the ground-truth dataset is balanced, 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 unmatched pairs which are actually positive pairs that were not covered by execution, we additionally check the source-level type of the unmatched pairs with the help of debug information. To better test the generalization ability of iCALLEE, we extract pairs from different binaries for training and testing. 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.
2) Hyperparameters Setup: We set the batch size to 512, and train the Siamese Neural 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. We adopt Batch Normalization [97] and Dropout [98] to help the network converge, and the dropout rate is set to 0.2.

3) Evaluation Metrics: We choose the commonly used 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 ideal classifier should have high Precision and high Recall at the same time. However, the two metrics can conflict with each other during training. Thus, we take the harmonic mean of them (i.e., F1) to further evaluate the classifier.

B. Efficiency of Indirect Callee Recognition

1) Effect of Slicing: To evaluate the effect of slicing, we first fixate other parts of iCALLEE (Loose symbolization, FCN feature extraction layers). Based on the ground-truth dataset, we compare two situations: full context and sliced context. As shown in Table III, the model trained with full context suffers from severe over-fitting: F1 drops 22.9% from training to testing. With roughly the same number of pairs, performing slicing increases the F1 by 22% approximately during testing, compared to the full context. Therefore, processing binaries with slicing could greatly help the Siamese neural network comprehend the context.

2) Effect of Symbolization: Similarly, we fixate the Siamese neural network (FCN feature extraction layers) of iCALLEE, and compare different symbolization policies. We set the hyper-parameter N of loose symbolization to 10 and train the Siamese network on the ground-truth dataset.

As shown in Table IV, strict symbolization has worse performance than loose symbolization. Results confirm that, the strict symbolization discards too much data-flow information, as discussed in Section IV. Additionally, the performance of strict symbolization degrades steeply (10.1% F1) from training to testing, which means that strict symbolization causes overfitting. In other words, strict symbolization leads to poor generalization performance. Therefore, embedding with loose symbolization could better preserve data-flow information.

3) Effect of the Siamese Neural Network: Based on the previous analysis, we fix the symbolization policy to the Loose symbolization. Then, we train the Siamese neural network with different datasets and try different feature extraction layers.

### Table III: Influence of slicing.

| Context | Train | Test |
|---------|-------|------|
|         | Precision | Recall | F1   | Precision | Recall | F1   |
| Full    | 94.7% | 94.8% | 94.7% | 90.0% | 73.1% | 71.8% |
| Sliced  | 97.4% | 97.5% | 97.4% | 93.5% | 93.8% | 93.7% |

### Table IV: Influence of symbolization.

| Symbolization | Train | Test |
|---------------|-------|------|
|               | Precision | Recall | F1   | Precision | Recall | F1   |
| Strict        | 95.9% | 96.1% | 96.0% | 85.1% | 86.7% | 85.9% |
| Loose         | 97.4% | 97.5% | 97.4% | 93.5% | 93.8% | 93.7% |

Robustness to different datasets. We train the neural network with the ground-truth dataset. Note that, binaries in this dataset are built with different compilers and different optimization levels. We therefore evaluate the robustness of the network given different datasets.

In addition to the ground-truth indirect-call dataset, we train several Siamese networks with multiple groups of direct call datasets, which are compiled with different compilers and different optimization levels. As shown in Table V, network trained with different datasets achieves an average F1 of 95.3%, and ROC curves are shown in Appendix B. The network performs better on O0 datasets but still gets an F1 of 95% on O1–O3 datasets. Even in the ground-truth dataset with much fewer pairs, the network still achieves about 93.7% F1. Therefore, the Siamese network can comprehend data from different sources, e.g., with mixed compilers and optimizations, and is applicable to both direct and indirect calls.

### Feature Extraction Layers.

Additionally, we test the performance of Siamese networks with different feature extraction layers. The FCN we test has three hidden layers with 512 neurons. The LSTM model has 512 neurons. The 1dCNN has one convolutional layer with 512 filters and the filter_length is set to 3. The TextCNN is adopted from [99]. We use ReLU as the activation function for these models.

As shown in Table VI, Siamese networks with FCN layers have the best performance, achieving an F1 of 93.7%. TextCNN layers perform slightly worse than FCN layers, with an F1 of 92.9%. 1dCNN layers perform best on the training set but overfits the most, leading to relatively poor performance on the test set. The F1 drops 6.1% 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 [100], even if LSTM tried to ease gradient problems by introducing gates [83].

### Classifier Network.

The final classifier is an FCN consisting of three layers with 512, 512, and 1 neurons respectively. The sigmoid function is used as the final activation function. Overall, we choose FCN layers as feature extraction layers based on the ground-truth dataset. The FCN layers are chosen as the final feature extraction layers.
and this FCN as the final classifier, and use the ground-truth dataset to train the final model.

4) Comparison with state-of-the-art solutions: Unfortunately, $\tau$CFI [26] and the refinement of TypeArmor [28] fail to discuss their efficiency in recognizing indirect callees and have not open-sourced yet, so we only compare iCALLEE with BPA [29] and TypeArmor [27]. Since BPA is not open-sourced, we adopt the results from their paper: based on a dynamically collected dataset [29], BPA and TypeArmor have precision rates of 35.1% and 57.6%, recall rates of 99.9% and 100% and thus F1-measures of 51.8% and 73.1% respectively.

As for real-world binary analysis tools such as IDA Pro, Angr, GHIDRA, etc., they identify indirect-call targets by constant propagation. Although constant propagation can avoid false positives, it can only resolve very few targets and has high false negatives. For indirect callsites of subject binaries in Table XII, constant propagation can only recognize 2, 4, 0 and 1 target(s) respectively.

As shown in Table III, IV, VI, iCALLEE has an $F1$-Measure of 93.7%, recall of 93.8%, and precision of 93.5%, therefore is much more effective than state-of-the-art solutions.

5) Time Efficiency: As a static analysis, iCallee has little time consumption. Based on the final model, we measure the time consumption of keys parts of iCALLEE. It takes about 103s to train the doc2vec model and 2.407s to train the Siamese network. After training, 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.

According to the number of callsites and address-taken functions in Table XII, iCALLEE takes 4~10 seconds in total to analyze a binary with iCALLEE and 6~14 seconds with TypeArmor. $\tau$CFI did not discuss efficiency in their paper and has not open-sourced either, so we suppose that it has an approximately equal time consumption to TypeArmor, because they are both implemented on DynInst [101]. However, as a pointer analysis, BPA needs more than 100 seconds to analyze small programs such as lighttpd, and more than 2,700 seconds to analyze large programs like nginx.

In summary, we could draw the following conclusion:

 Conclusion 1: iCALLEE is more efficient and effective at recognizing indirect callees than state-of-the-art solutions such as BPA, TypeArmor as well as binary analysis tools.

C. Applications of iCALLEE

1) Promoting binary similarity detection: With the trained Siamese neural network, we utilize iCALLEE 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 inter-procedural CFG (ICFG), 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 [102] to embed basic blocks. With generated embeddings, DeepBinDiff utilizes a $k$-hop greedy matching algorithm to match basic block pairs. In principle, if two indirect callsites in two binaries have similar callees, the two basic blocks they are in should be likely to match. Therefore, we can utilize iCALLEE to complement ICFGs, i.e., add indirect-call edges, and thereby improve the performance of DeepBinDiff.

Our experiments are performed on the same binaries used by DeepBinDiff, i.e., printenv, md5sum, splitl, 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. We compare iCALLEE with DeepBinDiff in performing diffing between binaries across different versions and optimization levels. To further verify the effectiveness of iCALLEE, given indirect callsites, we also add edges to random callees. 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.

Cross-optimization-level diffing. Table VII shows the F1-scores of cross-optimization-level diffing. We fix the Coreutils’ version to 7.6 and perform 6 experiments (O3 vs O2, O3 vs O1, O3 vs O0, O2 vs O1, O2 vs O0, O1 vs O0). Column +Rand and +iCALLEE indicate the F1-scores after adding random edges and iCALLEE edges to DeepBinDiff respectively. As shown, comparing with DeepBinDiff, adding random edges lead to a 1.9% F1-score decrease on average, while adding iCALLEE edges increases the F1-score by 13.7% on average. Statistics of DeepBinDiff, the +Rand and +iCALLEE setting are shown in Table IX, Table X and Table XI respectively. Specifically, adding random edges decreases all settings’ F1-scores, whereas adding edges given by iCALLEE increases all settings’ F1-scores, showing the effectiveness of iCALLEE in recognizing indirect callees.

Cross-version diffing. Table VIII 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). As shown, comparing with

| Optimization Levels | DeepBinDiff | +Rand | +iCALLEE |
|---------------------|-------------|-------|----------|
| 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 VII: Cross-optimization-level Binary Diffing Results.

| Versions          | DeepBinDiff | +Rand | +iCALLEE |
|-------------------|-------------|-------|----------|
| 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%    |

TABLE VIII: Cross-version Binary Diffing Results.
DeepBinDiff, adding random edges lead to a 2.3% F1-score decrease on average, while adding +CalLEE edges increases the F1-score by 4.6% on average. Statistics of DeepBinDiff, the +Rand and +CalLEE setting are shown in Table XIII, Table XIV and Table XV respectively. Consistent with the cross-optimization-level diffing results, adding random edges decreases all settings’ F1-scores and adding +CalLEE edges behaves in contrast.

Additionally, the evaluation shows that, comparing 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 in consistent with conclusions of DeepBinDiff and BINKIT [103]. Therefore we can obtain larger promotion in cross-optimization-level diffing by complementing the ICFG.

In summary, CalLEE can improve the performance of the state-of-the-art binary diffing solution DeepBinDiff by a large margin, especially in the cross-optimization-level diffing task. It indicates that CalLEE can effectively recognize indirect callees to promote inter-procedural binary analysis.

2) Promoting binary program hardening: With the trained Siamese neural network, we apply CalLEE to a fundamental task in program hardening: reducing indirect callees and refining CFI. We compare our model with state-of-the-art solutions: BPA, τCFI, TypeArmor and its refinement [28], a representative source-level type analysis IFCC [20], as well as real-world binary analysis tools such as IDA Pro, etc.

The IFCC is straight-forward because its easy to obtain type information with source code, yet the TypeArmor and τCFI need to recover callsite and callee signatures including numbers of arguments and the usage of return value at first. We choose the same binaries used by TypeArmor and τCFI, i.e., ProFTPd v1.3.3, nginx v0.8.54, lighttpd v1.4.28, and vsftpd v1.1.0, compile them with the same compiler Clang at the same optimization level O2 as TypeArmor does, and adopt the same metric used by TypeArmor and τCFI, which is the median number of callees for each callsite.

Table XII shows the comparison results. Column Total indicates the number of all the functions in a binary, and columns Callsites and AT indicate the number of indirect callsites and address-taken functions respectively. BPA and the refinement [28] of TypeArmor have not open-sourced their solutions, so we adopt the results in their papers: reducing 42% and 34.5% more targets on average than TypeArmor respectively. Moreover, 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 allows much fewer callees for callsites than τCFI on most binaries, and can reduce 72% indirect call targets than TypeArmor on average, which is better than BPA and the refinement solution. While as expected, IFCC still outperforms CalLEE, since it is a source-level solution which could utilize function type information to recognize callees.

| Binary | Total Callsites | TypeArmor (median) | τCFI (median) | CalLEE (median) | IFCC (median) |
|--------|----------------|-------------------|--------------|----------------|---------------|
| ProFTPD | 1,162 | 1,157 | 85 | 1,060 | 83 | 31 | 5 |
| nginx  | 1,110 | 1,052 | 218 | 254 | 528 | 99 | 25 |
| lighttpd | 358 | 356 | 54 | 47 | 51 | 9 | 6 |
| vsftpd | 455 | 449 | 4 | 12 | 10 | 5 | 3 |
| Average | 731.25 | 753.5 | 90.25 | 172.25 | 244.75 | 48.5 | 8.75 |

11
### TABLE XIII: Cross-version Binary Differing Results of DeepBinDiff.

| 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 XIV: Cross-version Binary Differing Results with Random Edges Added to DeepBinDiff.

| 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 | 97.9% | 97.3% | 92.6% | 97.6% | 93.8% | 92.0% | 92.3% | 95.8% | 94.8% | 94.6% | 94.6% |
| Average | 78.5% | 82.5% | 83.5% | 89.1% | 84.2% | 88.2% | 82.4% | 79.3% | 80.4% | 81.7% | 83.0% |

### TABLE XV: Cross-version Binary Differing Results with Indirect Call Edges Added to DeepBinDiff.

| 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% | 84.7% | 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% | 98.0% | 99.0% | 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% |

**False Negatives.** On the other hand, binary level solutions may have false negatives in recognizing indirect callees. To check false negatives, we take the results of IFCC as pseudo-ground-truths and compare tCALLEE with it. We find that, on average, tCALLEE only misses 2 callees in nginx, and 2 callees in lighttpd, which is consistent with Recall demonstrated in Table V.

We further inspect several false negatives by manually checking source code and assembly, and find that some of them are indeed false negatives. For example, in nginx, the function ngx_http_core_generic_phase performs an indirect function call ph->handler(r). The type of expected functions is ngx_int_t function (ngx_http_request_t *r). Thus, ngx_http_static_handler and ngx_http_index_handler which have matching types are candidate targets reported by IFCC. But tCALLEE wrongly predicts the latter as a non-matching callee (i.e., with a difference score 0.61).

Therefore, when enforcing CFI based on tCALLEE, one needs to eliminate these false negatives by profiling, i.e. running target program with test suites and collect legitimate indirect callees to avoid the false negatives. Actually, BPA establishes their ground-truths by profiling, and even TypeArmor can have false negatives as well [29].

**Outperforming IFCC.** We noticed that, in some cases, IFCC would have false positives even more than tCALLEE. In other words, tCALLEE provides more precise results than IFCC in some cases. For instance, in the binary lighttpd, function array_free performs an indirect function call a->data[1]->fn->free(a->data[1]). Functions fn->free and fn->reset have the same signature, and thus are reported as candidates by IFCC. But tCALLEE disallows the latter. Literally, human experts can verify that fn->reset is not a correct candidate, so tCALLEE surpasses IFCC in this case.

**Functions with variable-length arguments.** Type-based solutions, whether at binary-level or source-level, cannot well support variable-length argument functions. However, tCALLEE does not have this problem. tCALLEE 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.

**Comparison with binary analysis tools.** As mentioned in VI-B4, for indirect callsites of subject binaries in Table XII, binary analysis tools such as IDA Pro, Angr, GHIDRA, etc. can only recognize 2, 4, 0 and 1 target(s) respectively through constant propagation. Subsequently, they can barely reduce indirect callees for program hardening. Apparently, tCALLEE as well as BPA and type-based solutions all outperform current binary analysis tools by a large margin.

In summary, we could draw the following conclusion:

**Conclusion 2:** tCALLEE can promote inter-procedural binary analysis tasks such as binary similarity detection and binary program hardening.

**D. Interpretability of tCALLEE**

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

1) Embedding Model: We use T-SNE [104] algorithm to project high dimensional vectors to the two-dimensional space, to verify 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. For example, token ‘rb’ and ‘jnb’ are both instructions related to conditional jump, so they are clustered together in Figure 3. Therefore,
word vectors trained by the doc2vec model can well capture semantic features of tokens in assembly instructions.

2) **Siamese network**: We utilize the saliency map to interpret the network, to deduce the sensitivity of output regarding input vectors. Since iCALLEE only has one output from the Siamese network, we can focus on important tokens or instructions which are critical to the matching result.

First, we compute partial derivatives for input pairs. Given a callsite or callee slice (after vectorization) \( x \in \mathbb{R}^{l \times d} \). \( l \) is the length of the slice, and \( d \) is the dimension of a token’s vector. \( f(x) \) is the output of the Siamese network. The partial derivatives is given by:

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

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:

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

We visualize the results of an example pair which is predicted as “match” in Figure 4 with Ncrf++ [105]. The token with darker color means the larger \( S(x)[i] \). In other words, it means a greater contribution to model decision, according to the definition of saliency map. As we can see from Figure 4, in the slices of a callsite and callee, the two most important tokens are all related to argument register \( \text{rdi} \) and return value register \( \text{eax} \). It demonstrates that the network indeed can capture important features of instruction sequences.

In summary, we could draw the following conclusion:

**Conclusion 3**: The embedding model reasonably represents tokens in a high-dimensional space, and the Siamese neural network pays more attention to tokens and instructions related to argument and return value registers.

### VII. Limitations

**Indirect jumps**. Currently, iCALLEE only handles indirect calls 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 [21]. For the latter, they are almost the same as indirect calls. Our solution could be extended to support them in the same way, i.e., slicing, preprocessing, embedding and matching with a Siamese network.

**Applicability to programs with other calling conventions or in other architectures**. Other calling conventions differ from the calling convention of the System V AMD64 ABI. For example, for 32-bit programs using the x86 cdecl calling convention, function arguments are passed via the stack and do not involve registers. Therefore, to apply iCALLEE to 32-bit x86 programs, one can adjust the current policies of slicing and symbolization. In the same way one can apply iCALLEE to programs in other architectures. 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 Control-flow integrity (CFI)**. Most CFI solutions need to restrict indirect callees to mitigate the control-flow hijacking attack. As mentioned in VI-C2, to apply iCALLEE to CFI, additional efforts are needed to reach a 100% recall. Except for binary profiling, one can ease the false-negative problem by increasing the matching threshold, while introducing more false positives.

### VIII. Conclusion

In this paper, we present iCALLEE, a deep-learning-based approach that effectively recognizes indirect callees at the binary level. iCALLEE relies on an inter-procedural slicing algorithm to obtain fine-grained contexts of callsites and callees, and trains an assembly-centric doc2vec model to embed such contexts into feature vectors, and trains a Siamese neural network to match callsites with callees. Evaluation results show that, iCALLEE can recognize indirect callees with a high precision and recall, and can recover call graphs with a high precision and promote applications relying on inter-procedural analysis, e.g., binary code similarity detection and binary program hardening. By interpreting the embedding model and the Siamese neural network, we demonstrate that iCALLEE learns knowledge similar to human experts, and thus can apprehend the assembly language to some extent. Therefore, we believe that deep-learning approaches are promising for the indirect callee recognition program analysis task.
A. Call-site-callee pair collection

User-mode binaries. For user-mode binaries, we have several ways to achieve the goal. 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) [93] to get inputs that can cover as much code as possible. Then run the program with these inputs, and use Intel PT [84] to record execution traces. Finally, with the libipt [106] decoder library, we extract indirect call instructions from the trace, take their next instructions as targets and make pairs.

The Linux kernel. To monitor the kernel, we need a hypervisor. Likewise, we turn off Kernel Address Space Layout Randomization (KASLR) when compiling the kernel. Afterward, the kernel is emulated in an open-source record and replay platform PANDA [85], which is built upon the QEMU [107] 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 5, 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).

B. ROC Evaluation

To evaluate the robustness of the Siamese network on different datasets, we additionally visualize the relationship between \( TP \) and \( FP \) using the Receiver Operating Characteristic curve (ROC curve), and calculate the Area Under Curve (AUC). The diagonal dotted line indicates the performance of a random classifier with a 0.5 AUC. An ROC curve convex to the top left corner and an AUC close to 1 means that the classifier is close to optimal.

As shown in 6, network trained with multiple groups of direct call datasets, which are compiled with two different compilers (GCC and Clang), and four different optimization levels (O0, O1, O2, and O3), achieves an average AUC of 0.964, showing that the network is not sensitive to different compilers and optimization levels.

Fig. 6: ROC evaluation results based on the 8 datasets of direct-call pairs.