



# Analyzing and revivifying function signature inference using deep learning

Yan Lin<sup>1</sup> · Trisha Singhal<sup>2</sup> · Debin Gao<sup>3</sup> · David Lo<sup>3</sup>

Accepted: 30 January 2024 / Published online: 8 May 2024

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

## Abstract

Function signature plays an important role in binary analysis and security enhancement, with typical examples in bug finding and control-flow integrity enforcement. However, recovery of function signatures by static binary analysis is challenging since crucial information vital for such recovery is stripped off during compilation. Although function signature recovery using deep learning (DL) is proposed in an effort to handle such challenges, the reported accuracy is low for binaries compiled with optimizations. In this paper, we first perform a systematic study to quantify the extent to which compiler optimizations (negatively) impact the accuracy of existing DL techniques based on Recurrent Neural Network (RNN) for function signature recovery. Our experiments show that the state-of-the-art DL technique has its accuracy dropped from 98.7% to 87.7% when training and testing optimized binaries. We further investigate the type of instructions that existing RNN model deems most important in inferring function signatures with the help of saliency map. The results show that existing RNN model mistakenly considers non-argument-accessing instructions to infer the number of arguments, especially when dealing with optimized binaries. Finally, we identify specific weaknesses in such existing approaches and propose an enhanced DL approach named ReSIL to incorporate compiler-optimization-specific domain knowledge into the learning process. Our experimental results show that ReSIL significantly improves the accuracy and F1 score in inferring function signatures, e.g., with accuracy in inferring the number of arguments for callees compiled with optimization flag O1 from 84.83% to 92.68%. Meanwhile, ReSIL correctly considers the argument-accessing instructions as the most important ones to perform the inferencing. We also demonstrate security implications of ReSIL in Control-Flow Integrity enforcement in stopping potential Counterfeit Object-Oriented Programming (COOP) attacks.

**Keywords** Function signature · recurrent neural network · compiler optimization · control-flow integrity

---

Communicated by: Tegawendé F. Bissyandé

✉ Yan Lin  
yanlin@jnu.edu.cn

Extended author information available on the last page of the article

## 1 Introduction

Function signatures play a critical role in many security applications. For example, enforcing control-flow integrity (CFI) (Prakash et al. 2015; Van Der Veen et al. 2016; Muntean et al. 2018; Lin et al. 2019) and detecting code clones Hu et al. (2017) are typically based on function signatures, a fuzzer can strategically mutate test cases (Jain et al. 2018; He et al. 2019) for better vulnerability detection when function signatures are known. However, it is challenging to recover function signatures from stripped binaries, though, since there is no debug information available and the binary only contains low-level information such as instructions and register usage. Compilers typically do not preserve much language-level information, e.g., types, in generating the binary executable. To make things worse, compiler optimizations further complicate the recovery of function signatures Lin and Gao (2021).

Many existing techniques to recover function signatures for closed-source applications (Van Der Veen et al. 2016; Muntean et al. 2018; Balakrishnan and Reps 2007; Lee et al. 2011) are limited to custom and manually created rules based on calling conventions of the toolchain that generated the binary. For example, TypeArmor Van Der Veen et al. (2016) and  $\tau$ CFI Muntean et al. (2018) are based on the calling convention that the first six integer arguments are properly set and retrieved at callee and caller sites. A callee, defined as a function invoked or called by another function (the caller), plays a critical role in reconstructing the call graph. Consequently, CFI can be enforced based on the constructed call graph.

However, compiler optimizations may violate such calling convention as demonstrated in the recent study Lin and Gao (2021), where, e.g., a callee or caller may skip retrieving or setting certain arguments, respectively, resulting in the miscalculation of the number of arguments. For example, the accuracy in identifying the number of arguments at the callee site drops to 83% for applications compiled by clang with optimization flag O2. Also note that new optimization strategies are constantly being proposed and added to our mainstream compilers including gcc and clang. Moreover, identifying idioms common in binary code and designing analysis procedures, both principled and heuristic-based, have been an area that is reliant on human expertise, often engaging years of specialized binary analysts. These analysis engines need to be continuously updated as compilers evolve or newer architectures are targeted.

DIVINE Balakrishnan and Reps (2007) and TIE Lee et al. (2011) reduce the binary type inference problem into a constraint solving problem. However, they are too heavy-weight to be used in practice. For example, large-scale programs may result in too many constraints to solve. "DIVINE Balakrishnan and Reps (2007) spends 2 hours while analyzing programs of the order of 55, 000 assembly instructions" ElWazeer et al. (2013) because (i) there are many more instructions at the low-level code than high-level code; and (ii) there are several possible constraints for low-level instructions, such as add and sub Xu et al. (2018).

To avoid the reliance on a potentially brittle set of manually created rules and keep track of the latest optimization strategies to make it more efficient to infer function signature, an alternative line of research is proposed to train machines to learn features from binary code directly, without specifying compiler idioms and instruction semantics explicitly, e.g., EKLAVYA Chua et al. (2017). It uses word embedding and a three-layer Recurrent Neural Network (RNN) to learn the number and types of arguments from disassembled binary code. Meanwhile, no domain-specific knowledge is assumed and everything is inferred from the training data, and once a classifier is trained, it can learn the function signatures efficiently. However, the reported results in EKLAVYA show that the accuracy in inferring the number of arguments at callee and caller sites for *optimized binaries* (O1/O2/O3) is only around

80%, compared to 97% for unoptimized ones (OO). Moreover, the accuracy in inferring even a coarse-grained type (considers different types of integers, e.g., bool, short, int, and long as one type) for the third argument is only about 70% for optimized callees. Such low accuracy has a significant impact on corresponding applications, e.g., in enforcement of CFI.

To have a better understanding on how EKLAVYA infers the number of arguments and dig up the underlying reasons of inferior performance of RNN approaches in recovering function signatures from optimized binaries, in this paper, we make use of saliency maps Simonyan et al. (2013) to study which parts of an input the deep-learning network considers important in a prediction at the corresponding callees and callers. Our results show that most of the mistakes are not due to the learning capabilities of the model, but rather the absence of evidence that function arguments are accessed or prepared. Such absence of evidence in the representation of training and testing sets results in EKLAVYA considering non-argument-accessing instructions as the most important ones in determining the number of arguments — the critical cause of low accuracy for optimized binaries. For example, Compiler optimizations could potentially make two callees (callers) with different signatures look very similar in terms of their raw instructions because, e.g., the accessing and preparing of specific arguments are omitted, or that the same instructions are used to operate on different types of arguments. Having two functions with different ground-truth labels but seemingly identical function bodies confuses a supervised deep learning model in its training. The natural question then is how we could design an enhanced representation of the training and testing sets that incorporate all vital function signature evidence even when optimization is enabled, so that we can revivify the RNN capabilities in function signature recovery.

We address the challenge of inferring function signatures by incorporating compiler-optimization-specific domain knowledge into the representation of samples to improve the quality of the dataset. For example, our proposed system ReSIL selectively injects additional instructions into the function body of optimized training samples to reinstantiate evidence of the access or preparation of certain function arguments. Note that such *additional instructions injected* are solely based on the analysis of the optimized binary (e.g., child function of the callee and parent function of the caller) without relying on other information sources like the compilation process. ReSIL supports ELF binaries on Linux x86-64 and is able to recover function signatures with higher accuracy and F1 score compared to EKLAVYA . We also show that ReSIL correctly considers argument-accessing instructions as the most important ones in predicting the number of arguments.

We further use CFI enforcement as an example of applications of ReSIL to demonstrate its security implication. Due to the higher accuracy in recovering function signatures which further limits the control transfer targets allowed (only callees with matching function signatures) in an optimized binary, we show that stealthy Counterfeit Object-Oriented Programming (COOP) attacks could be defeated.

The contributions of the paper are as follows:

- We identify an extensive set of intricacies that confuse existing RNN approaches in inferring function signatures, and show that the root causes are not about learning capability but representation of binary samples.
- We use saliency map to study how existing RNN approaches mistakenly consider “noisy” instructions as most important in function signature inference.
- We propose ReSIL, a system that incorporates compiler-optimization-specific domain knowledge to improve the accuracy in inferring function signatures.

- We perform a thorough evaluation on ReSIL and show that ReSIL can achieve better accuracy and F1 score in inferring function signatures compared to EKLAVYA . We also demonstrate the security implication of using ReSIL for CFI enforcement.

## 2 Background and related work

In this section, we discuss related work on function signature recovery and machine learning approaches for analyzing binaries.

### 2.1 Function signature recovery

Function signature recovery is an important step in binary analysis. Variable liveness analysis and heuristics based on calling conventions and idioms are usually used to recover function signatures. ElWazeer et al. (2013) apply liveness analysis to recover arguments, variables, and their types for x86 executables. TIE Lee et al. (2011) infers variable types in binaries through formulating the usage of different data types. Caballero et al. (2009) make use of dynamic liveness analysis to recover function arguments for execution traces.

TypeArmor Van Der Veen et al. (2016) and  $\tau$ CFI Muntean et al. (2018) make use of liveness analysis and heuristics to recover the number of arguments and widths of the argument-storing registers at callee and indirect caller sites by inspecting the state for the six integer argument registers. Both of them can be used to enforce fine-grained CFI by matching the observed number of arguments and arguments widths at indirect caller and callee sites. Zeng et al. (2018) propose to perform type inference based on debug information generated by the compiler. Yan et al. Lin and Gao (2021) demonstrate how compiler optimization impacts function signature recovery and propose heuristic methods to recover function signature more precisely. Besides the difference of being a heuristic-based approach vs. a machine-learning-based approach as we do in this paper, Yan et al. Lin and Gao (2021) suffers from the requirement of having to constantly update the corresponding heuristics by domain experts whenever new compiler optimization strategies are incorporated into our modern compilers.

### 2.2 Machine learning for binary analysis

Machine learning methods have been adopted for different binary analysis tasks. For example, function (boundary) identification (Bao et al. 2014; Shin et al. 2015; Wang et al. 2017) is the preliminary of many advanced binary analysis, including our proposed system in this paper, ReSIL, EKLAVYA Chua et al. (2017), and Nimbus Qian et al. (2022). In addition, machine learning has been increasingly applied to type inference (Hellendoorn et al. 2018; He et al. 2018; Maier et al. 2019; Chen et al. 2020; Pei et al. 2021), compiler provenance recovery (Rosenblum et al. 2011; Otsubo et al. 2020; Pizzolotto and Inoue 2020; Tian et al. 2021; Ji et al. 2021), function similarity identification (Xu et al. 2017; Duan et al. 2020), and decompiling (Katz et al. 2018, 2019; Fu et al. 2019; Liang et al. 2021). Here, we discuss some of them briefly. Rosenblum et al. (2011) make use of linear Support Vector Machines (SVMs) to infer the compiler family, versions, optimization options, and source languages. Pizzolotto and Inoue Pizzolotto and Inoue (2020) recognize both the compiler and the presence of optimizations using a Long-Short Term Memory network and a Convolutional Neural Network. TypeMiner Maier et al. (2019) extracts Data Object Traces obtained through data dependency analysis and uses n-gram to capture the characteristics of each data object trace and

predict the data type. STATEFORMER Pei et al. (2021) first approximates execution effects of assembly instructions in both forward and backward directions and obtain a pretrained model, then infers argument and variable types with the pretrained model using transfer learning. CATI Chen et al. (2020) uses Convolutional Neural Network (CNN) to infer variable types by taking the instruction context into account. DEBIN He et al. (2018) recovers debugging information in stripped binaries including symbol names, types, and locations on three architectures (x86, x64, and ARM) using Extremely Randomized Trees classification and Conditional Random Field model. DEEPBINDIFF Duan et al. (2020) learns basic block embeddings via unsupervised DL and then use these embeddings to efficiently and accurately calculate the similarities among basic blocks. Katz et al. (2018) train an RNN model to convert binary code into C-like code directly and improve the syntax and semantic accuracy through post-processing. Our approach, ReSIL, belongs to this category of work, leveraging powerful machine learning techniques with a focus on improving the accuracy in analyzing *optimized* binaries specifically.

Since EKLAVYA is closely related to the contributions we are making in this paper, we discuss it in slightly more detail. Fig. 1 shows an overview of EKLAVYA .

EKLAVYA first uncovers the syntactic information of each instruction using word embedding. Specifically, all instructions are represented in a 256-dimensional vectors. Labels denoting the number of arguments and types (the ground truth) serve as input to a Recurrent Neural Network (RNN) with Gated Recurrent Unit (GRU). EKLAVYA has four tasks:

- **Task 1:** Inferring the number of arguments for each function based on instructions from the *caller*;
- **Task 2:** Inferring the number of arguments for each function based on instructions from the *callee*;
- **Task 3:** Recovering the type of arguments based on instructions from the *caller*;
- **Task 4:** Recovering the type of arguments based on instructions from the *callee*;

The classes of argument types are defined as  $\tau ::= int|char|float|void *|enum|union|struct$  with different types of integers (e.g., 32-bit integers and 64-bit integers) merged into one single type *int*. All the instructions (with a limit of 500) preceding a direct `call`

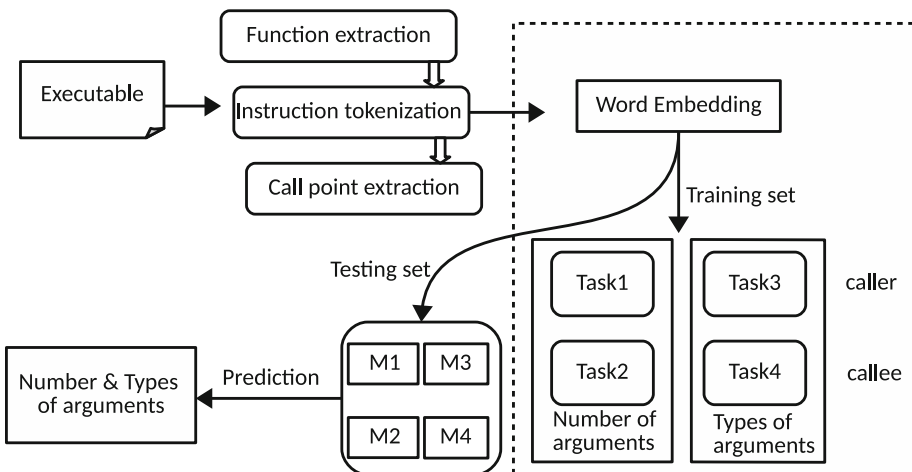


Fig. 1 Overview of EKLAVYA

instruction are used in tasks 1 and 3, whereas only instructions in the callee itself are used in tasks 2 and 4.

A key observation is that EKLAVYA uses instructions in a function body as input to RNN without considering whether they actually access the argument registers (potentially introducing noise) or whether they provide a complete representation of all arguments (potentially missing key inputs). In this paper, we first provide a detailed analysis on the extent to which these drawbacks would misguide the machine learning model when processing optimized binaries (our first and second contributions in this paper), and then propose applying compiler-optimization-specific domain knowledge in the representations of the input binary to improve model accuracy (our third contribution).

### 3 Methodology

Characterizing the impact of compiler optimizations on function signature inference is a complex problem. Modern compilers employ a vast number of optimizations, many of which are synergistic and require careful phase ordering to be effective. Individual optimizations are also quite diverse in their features; they have varying goals (e.g., reducing code size vs. reducing execution time), make different trade-offs (e.g., increase code size to reduce execution time), and have different scopes (e.g., intra- vs. inter-procedural). As a result, compilers bundle optimizations into levels (i.e., O0 - O3) tuned for different objectives such as fast compilation or fast execution.

In this section, we present the systematic methodology we developed to characterize the impacts of compiler optimizations on function signature inference.

#### 3.1 Method selection

Numerous deep learning approaches have been proposed for binary analysis, with some specifically focused on accurately recovering the types of arguments (variables) at callee sites, such as STATEFORMER Pei et al. (2021) and TypeMiner Maier et al. (2019). The primary objective of this paper is to investigate the impact of compiler optimization on function signature inference, encompassing both the number and type of arguments. To achieve this, we select the open-source approach EKLAVYA for an investigation into how compiler optimization affects its performance in recovering arguments at both callee and caller sites.

Another notable work called Nimbus Qian et al. (2022) also makes use of RNN with GRUs to infer function signature. However, its main purpose is to reduce the time in training and testing procedures and it achieves about 1% higher prediction accuracy over all function signature recovery tasks compared to EKLAVYA. The low level RNN architecture of it is quite similar to EKLAVYA except that it uses one multi-task learning (MTL) structure to infer the number and types of arguments simultaneously. Therefore, We stress that it should have the similar issues discussed in this paper and don't perform the detailed comparison since it is not open source.

#### 3.2 Benchmark selection and variant generation

We selected the same dataset employed in the EKLAVYA paper, comprising binutils, coreutils, findutils, sg3utils, util-linux, inetutils, diffutils, and usutils. We compiled these

programs using the latest compiler versions, resulting in the creation of 2,584 distinct variants (binaries) for our benchmark tests. This choice to use the most recent compiler versions aligns with the continuous improvement of optimization strategies. It's worth noting that this approach differs from replicating the experiments detailed in the EKLAVYA paper.

Our primary focus revolves around function signature inference within the x86-64 Linux environment. To generate the binaries, we employed two widely used compilers: gcc-10 and clang-10, each with varying optimization levels, namely O0, O1, O2, and O3. While optimization level definitions vary between compilers, they are consistent for GCC and Clang. Code produced at level O0 is almost entirely unoptimized. Specifying level O1 enables entry-level optimizations that reduce code size and execution time with minimal incremental compile time. At level O2, optimizations that improve execution speed without increasing code size are enabled. This increases compile time, but further reduces the binary's size and execution time over level O1. Specifying level O3 enables almost all optimizations and generally produces the fastest binaries, but they are generally larger and require more compile time versus level O2.

### 3.3 Variant analysis

To determine how each optimization level strategy impacts function signature inference, we conduct a comprehensive analysis, contrasting function signature in various variants against the groundtruth collected by baseline. More Specifically, for each variant, we recover the number and widths of arguments at the callee and caller sites using EKLAVYA and compare them with the groundtruth obtained by parsing the DWARF debug information Committee et al. (2010).

Furthermore, we delve into the source code of compilers, with particular emphasis on the mechanisms governing argument passage from callers to callees under distinct optimization flags such as -O0, -O1, -O2, and -O3. In our pursuit of a deeper understanding, we also consult the Intel instruction manual INTEL (2018) to gain insights into how various instructions might influence function signatures.

At the same time, we scrutinize instances in which EKLAVYA incorrectly infers the number and types of arguments at either callee and caller sites with the help of saliency maps. Through the aggregation of these observations, we identify recurring patterns and common features among the affected callees and callers. Finally, we consolidate our findings by summarizing the complexities encountered in function signature inference by EKLAVYA. For a comprehensive account of these complexities, please refer to Section 4.3.

### 3.4 Threats to validity

As with any empirical study, our methodology is subject to certain threats to validity. We acknowledge the following potential biases:

**Representativeness of the Selected Approach:** The findings of this paper are solely based on EKLAVYA and may not be fully representative of all RNN approaches. Our focus was on open-source approaches, and while they are widely used, our findings may not generalize to the broader population of other DL approaches. Nonetheless, our findings provide insights into function signature recovery, and we argue that they can be applied to all DL approaches that do not take into account inter-procedure control flow transfers in function signature inference.

**Table 1** Percentage of functions with specific number of arguments

Opt	% of functions with specific number of arguments									
	0	1	2	3	4	5	6	7	8	9
O0	7.36	32.50	31.41	18.11	7.21	3.29	0.08	0.01	0.02	0.01
O1	9.58	30.26	30.46	17.27	6.72	3.27	1.37	0.58	0.37	0.12
O2	8.93	27.43	31.49	18.02	7.43	3.72	1.61	0.72	0.48	0.19
O3	7.76	21.50	32.97	20.12	9.38	4.53	2.20	0.86	0.45	0.24

**Diversity of the Dataset:** It is important to acknowledge that our dataset may not cover all types of callees and callers. The determination of an appropriate dataset depends on various factors, such as the desired level of statistical significance, effect size, and variability within the population. While a larger dataset may provide more precise estimates and increase confidence in the findings, it may not always be feasible due to constraints such as time and budget. In our study, we believe that the Linux utility dataset, which consists of 2,584 different variants, provides valuable insights into common issues of EKLAVYA .

## 4 Why deep learning techniques fall short of optimized binaries

In this section, we take a deep dive into the reasons why existing RNN techniques fall short of recovering function signatures from optimized binaries. We first present our experiments with a state-of-the-art RNN technique EKLAVYA and its overall accuracy and F1 scores in processing unoptimized and optimized binaries (Section 4.1).

Due to the difficulty in explainability of machine learning models in general, our next task (Section 4.2) is to make use of saliency map to understand whether EKLAVYA manages to learn from argument-accessing-relevant instructions. We also use saliency map to analyze optimized binary samples with which EKLAVYA makes mistakes in its recovery of function signatures in order to shed light on possible reasons of its inferior performance (Section 4.3).

### 4.1 Accuracy in inferring function signatures

We evaluate the accuracy of inferring the number and types of arguments for a state-of-the-art RNN approach, named EKLAVYA Chua et al. (2017). The experiments are performed on a server machine with two 32-core AMD Ryzen ThreadRipper 3GHz CPUs, 128GB of RAM, and four GeForce RTX 2080 Ti GPUs with 12GB of memory.

We use the same sanitizing method in EKLAVYA to remove duplicated functions in the benchmark. The resulting dataset contains 51,907 distinct functions, and 104,046 direct callers. Table 1 shows the percentage of functions with specific number of arguments in different optimization levels. We find that most functions have fewer than 3 arguments (e.g., more than 80% of functions have fewer than 3 arguments as shown in Table 1). Therefore, we only report recovery results in type inference for the first 3 arguments, most of which are pointers, 32-bit integers, and 64-bit integers. Note that this strategy is also used by EKLAVYA and Nimbus.

We use 5-fold cross-validation to perform training and testing. The average results are shown in Tables 2 and 3 with the definitions of accuracy and F1 score given in Definition 1. Note that here we report the F1 score for fine-grained type inference, so that we can compare



**Table 2** Accuracy of EKLAVYA

(a) Inferring number of arguments

Inst	Opt	Clang			Gcc		
		CI	T	Acc%	CI	T	Acc%
Caller	O0	3,157	3,363	93.87	4,182	4,293	97.41
	O1	2,936	3,359	87.41	3,252	3,591	90.56
	O2	1,039	1,140	91.14	1,735	1,957	88.66
	O3	155	164	94.51	571	614	93.00
Callee	O0	2,000	2,022	98.91	2,175	2,209	98.46
	O1	1,772	2,089	84.83	1,349	1,542	87.48
	O2	725	797	90.97	1,008	1,136	88.73
	O3	150	159	94.34	390	425	91.76

(b) Inferring types of arguments

Inst	Arg	Opt	Clang			Gcc		
			CI	T	Acc%	CI	T	Acc%
Caller	1st	O0	3,645	3,766	96.78	4,061	4,173	97.31
		O1	3,004	3,133	95.88	3,239	3,362	96.34
		O2	1,015	1,049	96.75	1,717	1,795	95.65
		O3	160	163	98.16	518	530	97.74
	2nd	O0	2,065	3,274	90.81	2,598	2,828	91.87
		O1	1,647	1,895	86.91	2,087	2,331	89.53
		O2	631	673	93.76	1,068	1,175	90.89
		O3	123	127	96.85	338	358	94.41
	3rd	O0	1,142	1,291	88.46	1,466	1,608	91.17
		O1	806	918	87.80	1,298	1,422	91.28
		O2	347	382	90.84	693	760	91.18
		O3	89	91	97.80	223	237	94.09
Callee	1st	O0	1,799	1,871	96.15	1,963	2,049	95.80
		O1	1,809	1,927	93.88	1,360	1,420	95.77
		O2	716	735	97.41	1,012	1,047	96.66
		O3	147	150	98.00	381	390	97.69
	2nd	O0	1,079	1,215	88.81	1,170	1,327	88.17
		O1	1,087	1,252	86.82	851	954	89.20
		O2	492	528	93.18	648	710	91.27
		O3	115	121	95.04	278	293	94.88
	3rd	O0	489	579	84.42	528	633	83.41
		O1	514	612	83.99	409	473	86.47
		O2	240	265	90.57	320	361	88.64
		O3	59	66	89.39	143	156	91.67

CI: Number of callers/callees correctly inferred

T: Number of callers/callees in the testing set

**Table 3** F1 score of EKLAVYA

(a) Inferring the number of arguments

Inst	CPL	Opt	Number of Arguments										
			0	1	2	3	4	5	6	7	8	MF	
Caller	Clang	O0	0.87	0.96	0.96	0.94	0.92	0.87	0.87	0.93	0.87	0.94	
		O1	0.77	0.90	0.90	0.89	0.87	0.83	0.83	0.79	0.85	0.88	
		O2	0.81	0.92	0.91	0.92	0.93	0.85	0.86	0.86	0.92	0.91	
		O3	0.85	0.94	0.93	0.89	0.93	0.97	0.99	0.89	0.80	0.94	
	Gcc	O0	0.92	0.98	0.98	0.98	0.97	0.96	0.97	0.97	0.95	0.98	
		O1	0.79	0.92	0.90	0.92	0.90	0.89	0.89	0.92	0.92	0.90	
		O2	0.73	0.91	0.87	0.91	0.85	0.85	0.85	0.84	0.93	0.89	
		O3	0.78	0.94	0.91	0.94	0.89	0.90	0.85	0.85	0.78	0.92	
	Callee	Clang	O0	0.98	0.99	0.99	0.99	0.98	0.97	0.00	0.00	0.00	0.99
			O1	0.79	0.87	0.87	0.86	0.82	0.83	0.65	0.39	0.40	0.85
			O2	0.89	0.92	0.93	0.90	0.90	0.89	0.82	0.75	0.74	0.91
			O3	0.93	0.96	0.98	0.95	0.94	0.90	0.82	0.47	0.78	0.95
Gcc		O0	0.99	0.99	0.99	0.98	0.97	0.96	0.00	0.50	0.00	0.98	
		O1	0.80	0.89	0.90	0.89	0.87	0.85	0.67	0.48	0.43	0.88	
		O2	0.85	0.91	0.90	0.89	0.85	0.85	0.73	0.69	0.55	0.89	
		O3	0.90	0.93	0.93	0.93	0.88	0.89	0.75	0.56	0.43	0.92	

(b) Inferring types of arguments

Inst	CPL	Opt	Type of Arguments								MF		
			int8	int16	int32	int64	float	pointer	enum	struct		union	
Caller	Clang	O0	0.74	0.38	0.90	0.74	0.00	0.94	0.54	0.53	0.98	0.90	
		O1	0.63	0.17	0.88	0.71	0.00	0.92	0.65	0.37	0.97	0.88	
		O2	0.60	0.22	0.90	0.78	-	0.94	0.69	0.50	-	0.91	
		O3	0.75	-	0.96	0.74	-	0.98	0.80	-	-	0.96	
	Gcc	O0	0.77	0.31	0.93	0.76	0.00	0.94	0.75	0.50	0.96	0.90	
		O1	0.62	0.10	0.89	0.78	-	0.93	0.78	0.59	-	0.89	
		O2	0.66	0.00	0.91	0.80	-	0.93	0.71	0.90	-	0.90	
		O3	0.86	-	0.91	0.83	-	0.95	0.80	-	-	0.93	
	Callee	Clang	O0	0.93	0.33	0.97	0.55	0.17	0.93	0.25	0.36	0.88	0.91
			O1	0.39	0.25	0.87	0.57	0.22	0.92	0.34	0.19	0.22	0.89
			O2	0.63	0.00	0.91	0.74	1.00	0.95	0.47	0.89	-	0.94
			O3	0.95	-	0.90	0.81	-	0.97	0.93	-	-	0.95
Gcc		O0	0.91	0.10	0.96	0.55	0.39	0.93	0.29	0.69	0.88	0.91	
		O1	0.48	0.25	0.89	0.64	0.50	0.94	0.42	0.44	0.00	0.91	
		O2	0.61	0.33	0.91	0.70	1.00	0.94	0.59	0.00	0.00	0.93	
		O3	0.76	-	0.96	0.85	1.00	0.97	0.73	-	-	0.96	

it with our approach. We only analyze the type inference result for the second argument as it has a wider variety of different types. Note that MF (the last column of Table 3) measures the F1-score of the aggregated contributions of all classes; see its definition in Definition 3.

$$Acc = \sum_{i=1}^n P_i \times R_{C_i} \quad F1_i = 2 \times \frac{P_{C_i} \times R_{C_i}}{P_{C_i} + R_{C_i}} \tag{1}$$

where  $n$  is the number of labels in the testing set and  $P_i$  is the fraction of samples belonging to label  $i$  in the testing set.  $P_{C_i}$  and  $R_{C_i}$  are the Precision and Recall for class  $i$ , and they are defined as:

$$P_{C_i} = \frac{TP_i}{TP_i + FP_i} \quad R_{C_i} = \frac{TP_i}{TP_i + FN_i} \tag{2}$$

where  $TP_i$ ,  $FP_i$  and  $FN_i$  are the true positive prediction, false positive prediction, and false negative prediction of class  $i$  respectively.

$$MF = 2 \times \frac{mp \times mr}{mp + mr} \tag{3}$$

$$mp = \frac{\sum_{i=1}^n TP_i}{\sum_{i=1}^n TP_i + \sum_{i=1}^n FP_i} \quad mr = \frac{\sum_{i=1}^n TP_i}{\sum_{i=1}^n TP_i + \sum_{i=1}^n FN_i} \tag{4}$$

As shown in Table 2a, when optimization is enabled (O1/O2/O3), the accuracy in inferring the number of arguments drops compared to that for unoptimized callees and callers. Intuitively, this is mainly because that optimizations eliminate unnecessary argument reading (reading from argument registers) or preparing (writing to argument registers) instructions. Probably a bit counter-intuitive, another observation is that binaries compiled with O1 typically have lower accuracy than those compiled with O2 and O3. We will discuss in more details the reasons behind this in Section 4.3.

Table 2b shows more interesting results. First, the accuracy in inferring the types for the second and third arguments is lower than that for the first argument. When counting the type distribution, we realize that more than 95% of the first arguments are 32-bit integers and pointers, which are relatively easy to tell apart. On the other hand, a lot more of the second and third arguments are 64-bit integers, which are more difficult to recognize as the compiler typically uses similar instructions to access 64-bit integers and pointers regardless of optimization settings. We will discuss this in more detail in Section 4.3.

As shown in Table 3a, the F1 score for callers with zero arguments is relatively low. It shows that existing DL techniques have difficulties in distinguishing callers which actually do not have any arguments from callers which do not prepare any argument due to compiler optimization. Meanwhile, It also shows that the F1 score for callers compiled with optimization is generally lower compared to non-optimized callers. The F1 score for callees in Table 3a delivers a similar message that callees compiled with optimization have a lower F1 score in inferring the number of arguments compared with unoptimized callees. In addition, the F1 score in inferring callees with more than five arguments suffers a significant drop.

Table 3b shows that the F1 score for 64-bit integers (int64) is low but that for pointers is higher, which implies that EKLAVYA does not have a good performance in distinguishing between 64-bit integers and pointers. The F1 score for arguments whose sizes are less than 32-bit is quite small. We note that this is mainly because of lack of samples.

## 4.2 Analysis with saliency map

The previous section shows that EKLAVYA has a good performance for binaries compiled with O0 but suffers in inferring the number of arguments for optimized binaries. In this subsection, we further generate saliency map to dig up the reasons behind that — more specifically, does the RNN model manage to learn from *argument-accessing instructions* for its inferencing.

Given the instruction sequence of a certain callee (caller) function and a RNN model's (potentially incorrect) classification, a logical question to ask is: "which parts of the instruction sequence are most influential for the classification?" Does the DL model consider argument-accessing instructions most important in its inferencing (which makes sense), or does it rely more on "noisy" instructions (which could lead to inaccurate classification)? To answer this question, we seek to visualize the amount of influence each instruction contributes to the prediction. We find saliency map is a good candidate to help us answer this question since it provides per-time-step explanations that align well with the sequential nature of RNNs and offers a contextually relevant understanding of the model's decision-making process. Moreover, by examining the saliency map of misclassified samples, we can gain insights into the model's failure modes. Another possible interpretable approach is called Gradient-weighted Class Activation Mapping (Grad-CAM) Selvaraju et al. (2017), which is a technique primarily designed for interpreting Convolutional Neural Network (CNN) and is more suitable for visualizing the important regions in images that contribute to a specific class prediction. Although it can be adapted for RNN, it provides a more global explanation that may not be as relevant in the context of sequential data.

Specifically, Saliency maps are visual representations that highlight the most important regions or features in an input data sample that significantly influence the model's decision. Intuitively, the important part of an input is one for which a minimal change results in a different prediction. This is commonly obtained by computing the gradient of the network's output with respect to the input. In our work, We chose the approach described by Simonyan et al. Simonyan et al. (2013) to obtain the gradient by back-propagation. Given a sequence  $X_0$  of length  $|X_0|$ , and class  $c \in C$ , a RNN model provides a score function  $S_c(X_0)$ . We rank the instructions of  $X_0$  based on their influence on the score  $S_c(X_0)$ .  $S_c(X)$  is a highly non-linear function of  $X$  with deep neural nets. It is hard to directly see the influence of each instruction of  $X$  on  $S_c$ . Mathematically, around the point  $X_0$ ,  $S_c(X)$  can be approximated by a linear function by computing the first-order Taylor expansion:

$$S_c(X) \approx w^T X + b = \sum_{i=1}^{|X|} w_i x_i + b \quad (5)$$

where  $w$  is the derivative of  $S_c$  with respect to the sequence variable  $X$  at the point  $X_0$ :

$$w = \left. \frac{\partial S_c}{\partial x} \right|_{X_0} = \text{saliency map} \quad (6)$$

This derivative is simply one step of backpropagation in the RNN model, and is therefore easy to compute. It results in a Jacobian matrix and each element in a Jacobian matrix tells us how each dimension of the instruction vector will affect the output of a specific class. In this case, we just want to know how much effect a particular dimension has over the entire output, therefore we sum up the partial derivatives for all elements of the output with respect to the particular input dimension. The result is a 256-dimension vector which tells us the magnitude of change each dimension has over the input. In order for us to visualize our

saliency map, we choose to calculate the L2-norm of the gradient vector of each instruction in the function. To keep the value between 0 to 1, we divide each L2-norm with the largest one in the function. We call this value as the *saliency score*.

### 4.2.1 Categorization of instructions

In x86-64, argument registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` are used to pass the first six integer arguments, with the seventh and subsequent arguments passed onto the stack. Therefore, these argument registers and the corresponding instructions accessing them are deemed the most important in function signature inferencing. To see if these instructions actually have higher saliency scores, we first classify all instructions in callees and callers into the following six categories:

- **RG** : instructions reading the ground-truth integer argument registers.
- **RO** : instructions reading the non-ground-truth integer argument registers.
- **WG** : instructions writing to the ground-truth integer argument registers.
- **WO** : instructions writing to the non-ground-truth integer argument registers.
- **AS** : instructions accessing the stack pointer (register `%rsp`, `%rbp`) which are not in the above categories.
- **OT** : instructions that are not in the above categories (e.g., those that never access any integer argument registers and control-flow transfer instructions).

For example, the callee shown in Fig. 2a has three arguments and the instruction at Line 6 reads the ground-truth integer argument register `%rdx`; therefore it belongs to **RG**. Similarly, the caller in Fig. 2b requires three arguments and the instruction at Line 7 belongs to **WG** since it writes to the ground-truth argument register. A good RNN model should learn to consider **RG** (**WG**) as the most important instructions and instructions access the other argument registers (**RO**, **WO**) as the secondary important instructions. Arguments which are pushed onto the stack will be in category **AS**. Floating-point arguments that a callee uses and a caller prepares can be in the category **OT**. Since the percentage of callees which have floating-point arguments is small, we don't put such instructions into a new category.

### 4.2.2 Distribution of all instructions

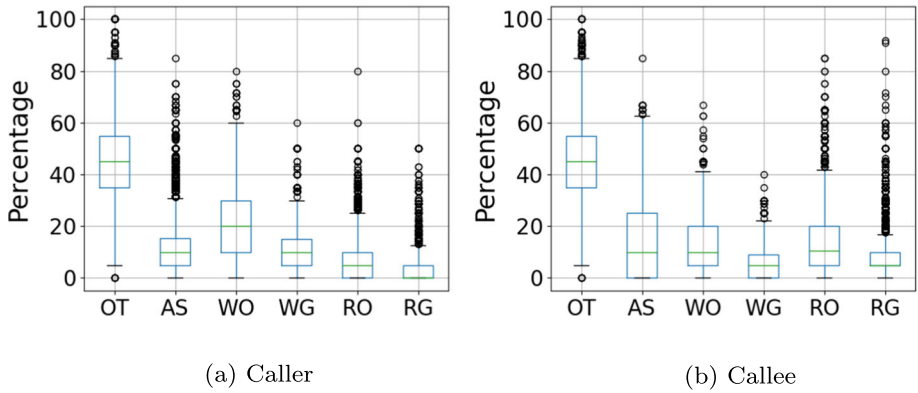
We first present the distribution of various instruction types for caller and callee instruction sequences which are the input to RNN, respectively, in Fig. 3. We can find the inputs to the RNN consist of a lot of **OT**, and the percentage of **WG** and **RG** is very small. This confirms

1	<code>push %rbp</code>	<b>AS</b>	<code>mov 0x90(%rax),%rax</code>	<b>OT</b>
2	<code>mov %rsp,%rbp</code>	<b>AS</b>	<code>mov %rax,-0x30(%rbp)</code>	<b>AS</b>
3	<code>sub \$0x90,%rsp</code>	<b>AS</b>	<code>cmpq \$0x0,-0x30(%rbp)</code>	<b>AS</b>
4	<code>mov %rdi,-0x10(%rbp)</code>	<b>RO</b>	<code>je 41c401</code>	<b>OT</b>
5	<code>mov %rsi,-0x18(%rbp)</code>	<b>RO</b>	<code>mov -0x40(%rbp),%rdi</code>	<b>WO</b>
6	<code>mov %rdx,-0x20(%rbp)</code>	<b>RG</b>	<code>mov -0x30(%rbp),%rsi</code>	<b>WO</b>
7	<code>mov -0x20(%rbp),%rax</code>	<b>AS</b>	<code>mov -0x18(%rbp),%rdx</code>	<b>WG</b>
8	<code>mov 0x28(%rax),%rax</code>	<b>OT</b>	<code>callq 4276b0</code>	

(a) Callee

(b) Caller

**Fig. 2** Example for different kinds of instructions



**Fig. 3** Instruction type distribution for the caller and callee instruction sequences

that function signature inferencing is a non-trivial problem, with DL proposed to identify the small amount of **WG** and **RG** instructions to perform accurate inferencing. The question is whether EKLAVYA manages to pick up **RG** or **WG** as the most important instructions.

### 4.2.3 Distribution of the most important instruction

We next focus on the specific types for instructions which are considered as the most important (with the highest saliency score) by EKLAVYA in inferring the number of arguments, and the results are shown in Table 4. Note that we exclude callees and callers with (ground-truth)

**Table 4** Instruction types for instructions that EKLAVYA considers most important

Inst	CPL	Opt	Instruction Types					
			<i>OT</i>	<i>AS</i>	<i>WO</i>	<i>WG</i>	<i>RO</i>	<i>RG</i>
Caller	Clang	O0	1341	172	469	966	136	21
		O1	1569	138	588	724	81	37
		O2	529	57	213	226	30	11
		O3	100	11	19	20	7	2
	Gcc	O0	1871	302	1033	679	134	28
		O1	1479	139	832	804	79	49
		O2	944	122	386	310	79	21
		O3	316	24	112	111	17	10
Callee	Clang	O0	323	228	100	74	90	965
		O1	678	70	150	58	317	706
		O2	262	21	40	19	143	245
		O3	52	6	7	4	41	64
	Gcc	O0	243	173	68	28	175	1314
		O1	409	70	90	49	266	617
		O2	320	22	90	20	195	466
		O3	109	23	32	8	90	153

The value shows the number of callers (callees) in which different types of instructions are considered as most important

0 argument since the RNN model would have no choice but to consider non-argument-accessing instructions as the most important ones. Results show that when optimization is enabled, the most important instruction to infer the number of arguments is *OT* for many callers and callees. When optimization is disabled, *OT* is still the most important instruction for callers even though the accuracy value is high as shown in Table 2a. That is, there are a significant number of cases which confuse the training model to use non-argument-accessing instructions to infer the number of arguments. We identify four categories of sub-cases and introduce them in the next section.

### 4.2.4 Zooming into samples misclassified

To get a better understanding of how and why EKLAVYA misclassifies test samples, we further zoom into the instruction types for callers and callees whose number of arguments are incorrectly classified by EKLAVYA ; see Table 5. As highlighted in the table, *OT* accounts for the biggest share, meaning that *OT* contributes to the most misclassification cases. That said, there are a significant number of cases where *WG* and *RG* instructions are correctly picked up as the most important, while EKLAVYA still fails to infer the number of arguments correctly. We therefore pick up some specific cases to have a deeper understanding.

**Examples of callees** Fig. 4 shows the saliency map for some examples of misclassified callees. As shown in Fig. 4a, callee `bfd_arch_default_fill` has three arguments but the instruction sequence only has an instruction that reads the first argument register (`%rdi`). EKLAVYA considers the call instruction at Line 6 as the most important one (which isn't really important for inferring the number of arguments). After checking the source

**Table 5** Instruction types of the most important instructions for callers and callees whose number of arguments are incorrectly recovered

Inst	CPL	Opt	Instruction Types					
			<i>OT</i>	<i>AS</i>	<i>WO</i>	<i>WG</i>	<i>RO</i>	<i>RG</i>
Caller	Clang	O0	50	15	37	61	5	-
		O1	146	19	68	97	20	5
		O2	41	6	12	23	5	1
		O3	4	-	1	1	2	-
	Gcc	O0	30	12	18	26	6	-
		O1	80	8	74	95	19	8
		O2	68	15	55	42	18	3
Callee	Clang	O0	2	7	3	-	7	9
		O1	96	16	23	6	95	75
		O2	20	2	2	2	29	12
		O3	1	-	-	-	6	2
	Gcc	O0	3	11	1	-	13	18
		O1	48	9	10	2	59	55
		O2	35	1	10	1	29	28
		O3	5	3	1	1	14	9

The value shows the number of callers (callees) in which different types of instructions are considered as most important

Instruction	Score	Instruction	Score	
1				
2	push %r14	0.1889	movzbl (%rdi),%eax	0.8519
3	push %rbx	0.0462	sub \$0x21,%eax	0.4929
4	push %rax	0.0000	cmp \$0xc,%al	0.2427
5	mov %rdi,%r14	0.3930	ja 409a58	0.2858
6	callq 406ff0	1.0000	movzbl %al,%eax	0.1725
7	mov %rax,%rbx	0.1823	nopl 0x0(%rax)	0.0836
8	test %rax,%rax	0.0873	xor %eax,%eax	0.0371
9	je 4042c1	0.2606	retq	0.0519
10	mov %rbx,%rdi	0.1245	nopl	0.1351
11	xor %esi,%esi	0.1010	xor %eax,%eax	0.0371
12	mov %r14,%rdx	0.7068	cmpb \$0x0,0x1(%rdi)	0.9825
13	callq 401f40	0.6221	jne 409a5a	0.6126
14	mov %rbx,%rax	0.6564	mov %esi,%eax	1.0000
15	add \$0x8,%rsp	0.0757	...	
16	pop %rbx	0.0206	retq	0.0449
17	pop %r14	0.1052	nopl	0.1145
18	retq	0.1498	cmpb \$0x0,0x1(%rdi)	0.9825

(a) bfd\_arch\_default\_fil compiled with clang-O1; ground truth with 3 arguments while misclassified as having 1 argument

(b) looks\_like\_expression compiled with gcc-O2; ground truth with 2 arguments while misclassified as having 1 argument

Instruction	Score	Instruction	Score	
1				
2	-	push %rbp	0.0000	
3	push %rax	0.0095	mov %rsp,%rbp	0.0107
4	mov 0x8(%rdi),%rax	0.3017	sub \$0x10,%rsp	0.0482
5	movzwl 0x20(%rax),%eax	1.0000	mov %rdi,-0x8(%rbp)	0.2453
6	xor %ecx,%ecx	0.1609	callq 4016c0	1.0000
7	cmp \$0x100,%eax	0.6657	movl \$0x5f,(%rax)	0.7103
8	setb %cl	0.4868	mov \$0xffffffff,%eax	0.4777
9	callq 404510	0.4972	add \$0x10,%rsp	0.1195
10	pop %rax	0.0757	pop %rbp	0.0588
11	ret	0.0000	retq	0.0036

(c) fh\_baseaddr\_query compiled with clang-O1; ground truth with 3 arguments while misclassified as having 2 argument

(d) getcon compiled with clang-O0; ground truth with 1 argument while misclassified as having 2 arguments

**Fig. 4** Saliency map example for callees whose number of arguments are incorrectly classified by EKLAVYA

code of the callee, we realize that the callee never uses the second and third argument, and therefore compiler optimization leaves only the instruction that reads the first argument in its function body. Similarly, callee fh\_baseaddr\_query in Fig. 4c has three arguments but the instruction sequence does not have any evidence about it since the reading of the second and third arguments is in callee at address 0x404510. Therefore, EKLAVYA considers the instruction at Line 5 as most important and the number of arguments is incorrectly predicted.

It is obvious that when the instruction sequence does not have **RG**, EKLAVYA would prefer to consider non-argument-accessing instructions as the most important ones. However, the mistakes also impact classification results for callees which actually have **RG** instructions, like the one in Fig. 4d. We can see that callee getcon actually has one argument but EKLAVYA incorrectly identifying instruction at Line 6 as the most important one even though the **RG** instruction at Line 5, in fact, provides the strongest hint.

**Examples of callers** Fig. 5 shows the saliency map of some examples of misclassified callers. As shown in Fig. 5a, the caller at Line 8 requires 1 argument but EKLAVYA determines that



	Instruction	Score	Instruction	Score
1	push %rbp	0.0000	push %r12	0.0000
2	push %rbx	0.0050	push %rbp	0.0652
3	push %rax	0.0380	push %rbx	0.0480
4	mov %rdi,%rbx	0.0520	mov %edi,%r12d	0.5254
5	mov 0x4c(%rdi),%ebp	0.7053	mov %rsi,%rbx	0.3048
6	movl \$0x1,0x4c(%rdi)	1.0000	mov %edx,%ebp	1.0000
7	callq 49dcf0}		callq 40a4a6	

(a) d\_expression compiled with clang-O1; ground truth with 1 argument while misclassified as having 0 argument

(b) command compiled with clang-O1; ground truth with 0 arguments while misclassified as having 3 arguments

	Instruction	Score	Instruction	Score
1	sub \$0x68,%rsp	0.0153	push %rbp	0.0092
2	mov %edx,%r11d	0.0340	push %r14	0.0000
3	mov %rdi,%r9	0.0340	push %rbx	0.0137
4	mov %rcx,%r8	0.0332	mov %esi,%r14d	0.0439
5	lea 0x8(%rsp),%rdx	0.0286	mov %edi,%ebp	0.0438
6	xor %eax,%eax	0.0000	mov \$0x48,%edi	0.0912
7	mov \$0xb,%ecx	0.0972	callq 406ff0	0.1035
8	mov %r9,(%rsp)	0.1986	test %rax,%rax	0.0096
9	mov %rdx,%rdi	0.0793	je 488546	0.1301
10	mov %rsi,%rdx	0.4254	mov %rax,%rbx	0.1100
11	mov %rsp,%rsi	0.1677	mov \$0x4891f0,%esi	0.7447
12	rep stos	0.3558	mov %rax,%rdi	0.0292
13	mov %r11d,%ecx	1.0000	mov \$0x38,%edx	0.6892
14	mov %r9,%rdi	0.8014	mov \$0x413b,%ecx	1.0000
15	callq 405410		callq 406510	

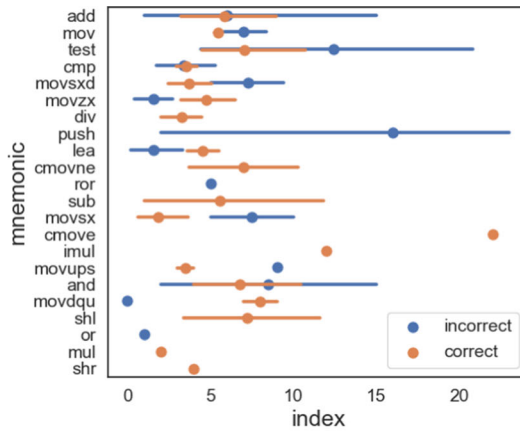
(c) argp\_help compiled with gcc-O2; ground truth with 5 arguments while misclassified as having 3 arguments

(d) sec\_merge\_init compiled with clang-O1; ground truth with 4 arguments while misclassified as having 3 arguments

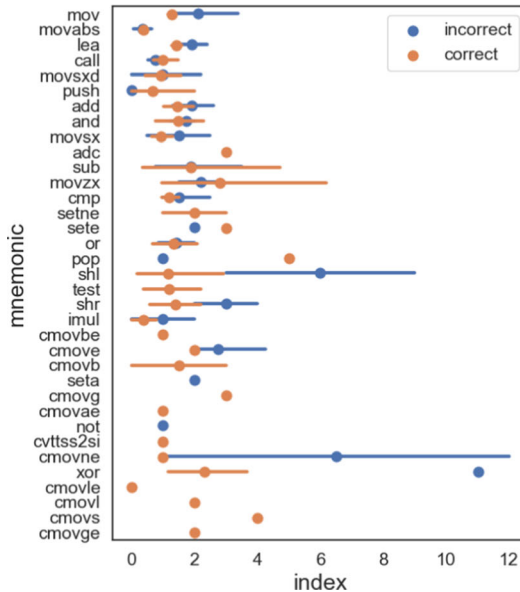
**Fig. 5** Saliency map example for callers whose number of arguments are incorrectly predicted. *Callers that need to predict the number of arguments are highlighted with gray color*

it doesn't require any argument since the instruction sequence does not have **WG**. It, instead, considers the instruction at Line 7 as the most important. Caller in Fig. 5b does not require any argument but EKLAVYA determines that it requires one. Caller in Fig. 5c requires 5 arguments but the instruction that writes to the fifth argument register (%r8) at Line 5 has a very small saliency score, leading EKLAVYA to determine that it has 3 arguments rather than 5. Caller at Line 16 of Fig. 5d requires 4 arguments with the **WG** instruction at Line 15 having the largest saliency score. However, EKLAVYA still wrongly classifies it as having only 3 arguments.

**Examples of misclassification with correctly identified RG and WG** More interestingly, even when **RG** and **WG** are considered as the most important instructions by EKLAVYA as shown in Fig. 4b and Fig. 5d, the classification results are still incorrect for some callees and callers. We perform detailed analysis for these cases and find that those **RG** and **WG** instructions usually have a larger distance from the entry of the function and to the caller, respectively, compared to cases which are correctly predicted. Fig. 6 draws the distance (in terms of the number of instructions from the entry of the function or to the caller) of the instruction with the highest saliency score. We can see that the misclassification cases typically have a larger distance compared to cases of correct classification. In addition, all the argument registers can also be used to store temporary values, and therefore the RNN



(a) Callee. *index* is the distance from the entry of the callee.



(b) Caller. *index* is the distance to the caller minus one.

**Fig. 6** Instruction types and distance measurement for the most important instructions when the number of arguments is incorrectly recovered

model may incorrectly consider **WG** as very important at the caller site, when the registers are in fact being used to store temporary values.

**Top 5 important instructions** Finally, we extend our analysis to five instructions with the biggest saliency scores instead of only the top one, in an effort to see to what extent EKLAVYA is completely lost in picking up **WG** and **RG** instructions (not even among the top 5). Table 6 shows the number of functions (among the misclassification cases) which manage

**Table 6** Instruction types of the top 5 important instructions for callers and callees whose number of arguments are incorrectly recovered

CPL	Opt	Caller		Callee	
		without <b>WG</b>	with <b>WG</b>	without <b>RG</b>	with <b>RG</b>
Clang	O0	32	136	9	19
	O1	147	193	178	141
	O2	38	50	39	28
	O3	4	4	6	3
Gcc	O0	31	61	12	34
	O1	89	195	81	102
	O2	76	125	47	57
	O3	21	15	14	19

The value shows the number of callers (callees) with at least one **WG** or **RG** instruction in its top 5 most important instructions

to identify at least one **WG** or **RG** instructions among the top 5 most important instructions and otherwise. We can see that EKLAVYA fails to identify **RG** and **WG** instructions as the top 5 most important instructions for a large number of callees and callers, which shows up more significantly with optimized binaries.

### 4.2.5 Summary of the saliency map analysis

Our detailed analysis using saliency map shows that the existing RNN approaches fail to identify argument-accessing instructions as important ones in inferring the function signatures, and that is the reason for many misclassification cases. The problem shows up even more significantly with optimized binaries. The next question is, is this failure in identifying the most important instructions due to the poor learning capability of our machine learning model, or is this failure due to other reasons, e.g., noisy training data.

## 4.3 Four key scenarios that contribute to the lower accuracy and bad saliency map

The overall statistics presented in Section 4.1 and Section 4.2 show that existing RNN approaches fall short on analyzing optimized binaries. However, the failure in attributing important instructions to **WG** and **RG** could be due to the poor learning capability of the machine learning, noisy inputs in the training data, or even non-existence of **WG** and **RG** instructions. In this section, we dig deeper into the reasons behind that observation by analyzing the instructions in functions to which EKLAVYA makes mistakes in inferring the number and type of function arguments. Specifically, we summarize the common features those callees and callers have and compile our findings into the following four key scenarios. For example, for callees whose top 5 important instructions do not have instruction type **RG**, we perform detailed analysis to collect the features they have.

### 4.3.1 Missing argument-reading instructions

This refers to cases where argument-reading instructions are missing in optimized binaries due to, e.g., dead code elimination, dead argument elimination, constant propagation, and other optimization strategies. This scenario could be the result of the following two sub-cases.

**Arguments are accessed in helper functions (denoted as *Helper* )** There are cases where the access of an argument is in helper functions when optimization is enabled. Here, a helper function is a function that performs part of the computation and is called by the callee being analyzed. Fig. 7a and b show such an example in which all the arguments of function `lua_toboolean` are accessed in helper function `index2adr`. If the input to the RNN engine only consists of the function body of `lua_toboolean` (which is the case for existing deep learning approaches like EKLAVYA ), the training sample would confuse the RNN model since the sample label indicates multiple arguments whereas the function body (helper excluded) does not have the corresponding argument reading instructions. As a result, the model would likely pick up other irrelevant information, e.g., *OT* , as important instructions.

**Arguments are not used (denoted as *Unread* )** There are also cases with unused arguments in callees, usually due to fixed prototypes of the functions (e.g., virtual functions). However, the label given to the RNN engine in existing approaches always has these unused arguments counted, which would confuse the training process. As shown in Fig. 8, the first

```

1 LUA_API int lua_toboolean (lua_State *L, int idx) {
2     const TValue *o = index2adr(L, idx);
3     return !l_isfalse(o);
4 }
5 push    %rax
6 callq   4021f0 <index2adr>
7 mov     %rax,%rcx
8 mov     0x8(%rax),%eax
9 test    %eax,%eax
10 je     402674 <lua_toboolean+0x24>
11 cmp     $0x1,%eax
12 jne    40266f <lua_toboolean+0x1f>
13 xor     %eax,%eax
14 cmpl   $0x0,(%rcx)
15 setne  %al
16 pop     %rcx
17 retq

```

(a) Arguments being accessed in a helper function

```

1 index2adr:
2 4021f0:    85 f6          test    %esi,%esi
3 402206:    48 3b 4f 10   cmp    %rcx,(%rdi+0x10)

```

(b) Argument reading instructions in the helper function

```

1 0000000000402650 <lua_toboolean>:
2 push    %rax
3 d6 f6
4 48 d6 4f 10
5 callq   4021f0 <index2adr>

```

(c) ReSIL inserting instructions at callee

**Fig. 7** Arguments are accessed in helper function

```

1 GLOBAL(void) jpeg_free_large (j_common_ptr cinfo, void FAR * object, size_t
  ↳ sizeofobject) {
2     free(object);
3 }
4 mov     %rsi,%rdi
5 jmpq   400950 <free@plt>
6 caller site:
7 mov     0x70(%r14,%r15,8),%rsi
8 mov     %r12,%rdi
9 mov     %rbp,%rdx
10 callq  41b6b0 <jpeg_free_large>
    
```

**Fig. 8** Not reading argument registers

and third arguments of `jpeg_free_large` are not used, but the label used in the training set indicates that it has three arguments.

### 4.3.2 Missing argument-preparing instructions (denoted as *Wrapper* )

If a caller is in a wrapper function, an optimized compiler may decide not to reset the argument registers but simply “pass them through” from the caller of the wrapper function. Here, a wrapper function is a subroutine whose main purpose is to call another subroutine. As shown in Fig. 9a, function `ar_emul_append` is a wrapper function and existing RNN approaches only use the four instructions from Line 2 to Line 5, none of which accesses an argument register, to infer the number of arguments of the caller at Line 6, while the ground-truth label indicates five arguments. It would, again, confuse the training process.

We can see that in both cases of missing argument-reading and missing argument-preparing instructions, the label of the training sample does not tally with instructions in the function body for optimized binaries, which affects the training quality of RNN models and in turn the accuracy of function signature recovery. To rectify this problem, our key idea would be to make the label and representation of function body agree; see how ReSIL achieves this in Section 5.

Figure 10 shows the distribution of the most important instruction for *Helper* , *Unread* , and *Wrapper* . We can find that for *Helper* , EKLAVYA uses non-argument-accessing instructions (*OT* ) to infer the number of arguments. *RO* and *OT* have the largest score for most callees as

<pre> 1 ar_emul_append: 2 mov     0x2f8e19(%rip),%rax 3 test   %rax,%rax 4 je     342bf 5 sub    \$0x8,%rsp 6 call   352b0 7 - 8 - 9 -     </pre>	<pre> 1 ar_emul_append: 2 48 0f 25 c7    op %rdi 3 48 0f 25 c6    op %rsi 4 48 0f 25 c2    op %rdx 5 0f 25 c1      op %ecx 6 41 0f 25 c0    op %r8d 7 mov     0x2f8e19(%rip),%rax 8 ..... 9 call   352b0     </pre>
---	---

(a) Caller missing argument-preparing instructions

(b) ReSIL inserting instructions at caller

**Fig. 9** Missing argument-preparing instructions

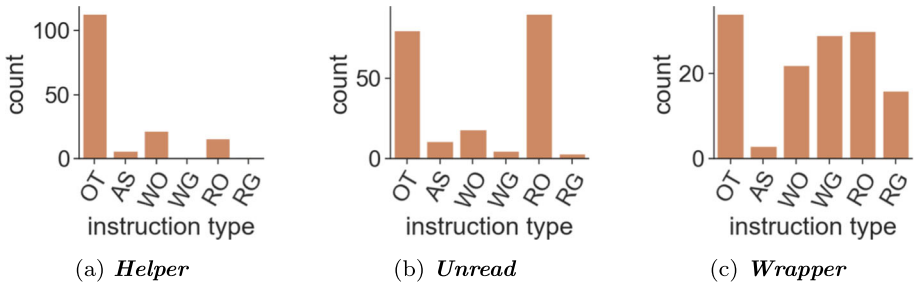


Fig. 10 Most important instruction types for *Helper*, *Unread*, and *Wrapper*

shown in Fig. 10b. Since some *Wrapper* cases will prepare the ground-truth argument register but do not set other argument registers, there are significant *WG* cases in Fig. 10c. We can also find that for case *Wrapper*, instructions that read the argument register (*RO*, *RG*) are also considered as most important in EKLAVYA for many callers. These cases would further confuse the RNN models to wrongly consider *OT* as the most important instructions for callees (callers).

### 4.3.3 Indistinguishable cases

The next key reason to lower accuracy when dealing with optimized binaries refers to cases where even human experts would not be able to classify correctly with all information presented.

**Argument registers used for other purposes (denoted as *Temp*)** As described in the Intel Manual INTEL (2018), all argument registers could also be used as scratch registers to store temporary values. This serves as noise to the RNN engine, sometimes to the extent that it is impossible to distinguish the intended usage of a register. There is hardly any evidence for distinguishing the two cases even for human experts, not to mention a machine learning engine; see an example in Fig. 11.

We had considered handling such cases by accommodating information from the callee. However, some argument-reading instructions at the callees could have been eliminated due to optimization (as discussed above) while similar optimization is absent at the caller site, which introduces mismatches and uncertainties. In addition, we are not able to get actual target for an indirect caller from a binary. Therefore, it remains infeasible to distinguish them with reasonable accuracy.

```

1 mov %eax,%esi
2 test %r15,%r15
3 je 51e1ad
4 lea 0xe0(%rsp),%rdi
5 callq (func_one(&cc,c))
    
```

(a) %esi used to pass argument

```

1 mov %ebp,%esi
2 test %rax,%rax
3 je 43ae6f
4 mov %ebp,%edi
5 callq obj_attrs_order(i)
    
```

(b) %esi used to store temporary

Fig. 11 Indistinguishable argument register usage

```

1 mov %rdi, %rbx
2 mov %rsi, %rdi (pointer)
3 mov %rdx, %rsi
4 call 505a00
    
```

(a) tr: append\_char\_class

```

1 mov %rdi, %rbp
2 mov %rsi, %rdi (size_t)
3 sub %rsp, 0x8
4 call 402640
    
```

(b) tr: xmemdup

**Fig. 12** Example of Indistinguishable types

**Indistinguishable argument types (denoted as *Indis-type* )**

There simply isn’t enough evidence for a machine learning engine to tell some types apart as the same (or similar in general) instruction can be used in an optimized binary to access different types of arguments. This problem also makes it hard for a DL engine to differentiate various integer types (e.g., int and long), which could provide significant benefits to security applications like CFI. We further discuss this in Section 6.4. It is more difficult to distinguish these cases for optimized binaries since compiler optimizations remove many redundant instructions that would provide information to help distinguish them. Fig. 12 shows such an example.

**4.3.4 Irrelevant instructions (denoted as *Irrelevance* ).**

The last contributor to lower accuracy in processing optimized binaries and bad saliency map information for unoptimized ones refer to noise in the training and testing samples. Current DL approaches take the entire function body as input, where many instructions are not related to the identification of the number (and types) of arguments. Such noise may affect the performance of machine learning as shown by Sharma et al. (2015). As shown in Fig. 7a, many instructions are not relevant to argument reading, such as instructions at Line 9 and Line 10. As shown in Table 4, *OT* are considered as the most important instruction in inferring the number of arguments for a lot of callees and callers even though *RG (WG )* and *RO (WO )* can be found in the instruction sequence, such as the callees and callers compiled with optimization flag O0.

**4.3.5 Statistics of the various complication scenarios**

Table 7 shows the number of occurrences of *Helper* , *Unread* , *Wrapper* , and *Temp* in our dataset. They are calculated by making use of static binary analysis based on TypeArmor and then the recovered signature is compared with the ground truth. It is clear that the complications of *Helper* , *Unread* , *Wrapper* only occur in optimized binaries, which partially

**Table 7** Number of intricacy cases

Opt	# <i>Helper</i>	# <i>Unread</i>	# <i>Wrapper</i>	# <i>Temp</i>
O0	0	0	0	6,803
O1	1,182	1,189	819	3,218
O2	349	568	82	1,630
O3	78	151	6	266
Total	1,609	1,908	907	11,917

explains why existing deep learning approaches have lower accuracy in analyzing them. Interestingly, the number of these cases in binaries compiled with O1 is larger than those compiled with O2 and O3. Our further analysis shows that O1 binaries simply have a larger number of functions compared to O2 and O3 binaries. In other words, the numbers reported in Table 7 do not necessarily imply the likelihood of occurrences in different optimization levels. Another interesting observation is that unoptimized binaries are more likely to use argument registers to store temporary values (case *Temp*).

We do not report the number of cases for *Indis-type* and *Irrelevance* as it is difficult to define what kind of instructions are considered similar and that almost every callee (caller) would have irrelevant instructions.

## 5 ReSIL: revivifying deep learning on optimized binaries

In the previous section, we detail four complication scenarios that contribute to the lower accuracy of DL when processing optimized binaries. In this section, we present our solutions to these four scenarios and propose ReSIL to incorporate compiler-optimization-specific domain knowledge into the representation of samples to make DL models regain its learning capability.

### 5.1 Missing argument-related instructions

As discussed in Section 4.3, the problem here is that the function body (with missing argument-reading or argument-preparing instructions) does not match with ground-truth labels, causing difficulties in the learning process. Intuitively, we need to correct such mismatches so that DL can regain its learning capability and high accuracy. Such correction in ReSIL takes different forms depending on the nature of the mismatches.

**Arguments are read in helper functions (*Helper*)** A simple solution is to include all instructions in the helper function, but that will also include many irrelevant instructions which would at the same time hurt the learning process. Instead, we make use of inter-procedural analysis to find all (potential) argument-reading instructions (process of which is identical to that detailed in existing work Lin and Gao (2021)), and “summarize” them with our newly introduced instruction set before inserting the summarized instructions ahead of the call to the helper function. Such summarizing instructions serve as part of the input samples to the machine learning model. Note that the summary instructions inserted preserve the operand information with only the opcode being replaced, to signal to the machine learning engine that the corresponding argument related instructions are present in a helper function.

Our newly introduced instruction set includes the following opcode that is not defined in the Intel Manual INTEL (2018):

- `0xd6` is used to summarize any single-byte argument-reading instructions.
- `0x0f 0x25` is used to summarize any two-byte argument-reading instructions whose operand is an integer.
- `0x0f 0x27` is used to summarize any two-byte argument-reading instructions whose operand is floating-point.
- `0x0f 0x38 0x51` is used to summarize any three-byte argument-reading instructions whose single operand is an integer.
- `0x0f 0x38 0x53` is used to summarize any three-byte argument-reading instructions whose single operand is floating-point.



The key idea of introducing our new opcodes here to summarize the argument-reading instructions is two-fold. First, once these instructions with our new opcode are inserted into the function body, we correct the mismatches between function body and ground-truth labels. Note that we do not need to tell the DL engine what these newly introduced opcodes mean, as the DL engine is supposed to be able to learn their meanings given sufficient number of training samples. Second, such insertion of only summary instructions avoids “over-correcting” with other noisy instructions.

For the example in Fig. 7a and b, we identify the two argument-reading instructions in function `index2adr` and insert two summary instructions with our newly defined opcode; see Line 3 and Line 4 in Fig. 7c<sup>1</sup>. We perform the same correction to both training and testing samples.

We emphasize here that our identification of argument-reading instructions in the helper function does not need to be 100% accurate, e.g., an instruction reading `%rcx` not for argument reading but for accessing temporary storage could also be summarized and inserted into the function body. We leave the DL model to choose which extra instruction inserted is more important in inferring the number of arguments as our additional treatment here is not to replace the DL engine but to present it the corrected samples.

**Arguments are not used (*Unread*)** In this case, ReSIL takes a simpler approach to correct the label so that the deep learning classification would eventually output the number of arguments used by a function rather than the number of arguments the function (to be more precise — its source code) has. Therefore, in the training set, we label function `jpeg_free_large` in Fig. 8 as using two arguments.

One may question the extent to which such a change in the expected output of the machine learning engine would impact its usefulness in specific application scenarios. For example, in the case of CFI, it will lead to the use of a CFI policy that the number of arguments passed at the caller site should be equal *or larger than* that at the targeted callee site. However, we argue that such impact would be minimal as existing fine-grained CFI enforcement always apply exactly the same policy due to the conservative inferencing at callees and callers (Van Der Veen et al. 2016; Muntean et al. 2018).

ReSIL obtained the corrected ground-truth label by examining the presence or absence of attribute `__attribute__((unused))` or by performing static binary analysis Lin and Gao (2021) to find out the number of argument registers that are actually used. Such analysis is only performed for the training samples. The ground-truth label of the testing samples is obtained by performing static binary analysis Lin and Gao (2021).

### Arguments are not set in wrapper functions (*Wrapper*)

For the same reason outlined above, here we summarize the argument-preparing instructions in caller of the wrapper function (identified in the same way as in related work Lin and Gao (2021)) with our newly introduced opcodes, and then insert them into the wrapper function. Fig. 9b shows the result of our insertion (Line 2 to Line 6) to the example shown in Fig. 9. Note that we cater for cases where argument-preparing instructions appear in both the wrapper function and its caller. We perform this analysis and correction for both training and testing samples.

<sup>1</sup> The opcode for the instruction at Line 4 is 0x3b with 0x48 being the prefix used to indicate use of a 64-bit register.

## 5.2 Indistinguishable cases

Since these are indistinguishable cases (number of arguments and argument types) where not enough evidence is present in the function body for the DL engine to perform classification, ReSIL simply outputs the top five inferencing results rather than only the top one as in the existing approach EKLAVYA. For the example in Fig. 12b, the top five outputs for the type of the second argument are  $\langle \text{pointer}, \text{int64}, \text{int32}, \text{float}, \text{int8} \rangle$  with probabilities  $\langle 0.876, 0.124, 3.18\text{e-}05, 2.23\text{e-}05, 2.44\text{e-}07 \rangle$ . We can see that it has a non-negligible probability being a 64-bit integer.

## 5.3 Irrelevant instructions

We consider an instruction relevant if it accesses any argument registers or stack addresses. Meanwhile, any branch instructions are also considered relevant. For example, the irrelevant instructions in function `lua_toboollean` are shown in Fig. 7a with light gray color, which are not included as input to the DL engine (for both training and testing samples) in ReSIL.

## 5.4 Classification output of ReSIL

With our corrections to the DL engine discussed above, the output of ReSIL for a target function  $a$  includes:

- **Number of arguments.** At the callee site  $a$ , ReSIL outputs the number of arguments used by  $a$  while at the caller site, ReSIL outputs the number of arguments that are passed to  $a$ . Note that the ground-truth of the two outputs could be different. Also note that ReSIL outputs the top five most likely values.
- **Types of arguments.** Each argument of  $a$  is defined as:  $\tau ::= \text{int8}|\text{int16}|\text{int32}|\text{int64}|\text{pointer}|\text{struct}|\text{float}$ . Different from the *struct* type in EKLAVYA, we only use *struct* to represent an argument on the stack whose aligned size is bigger than 16 bytes. We also don't have types *enum* and *union* used in EKLAVYA since ReSIL always outputs the corresponding container type.

## 6 Evaluation

In this section, we first evaluate the performance of ReSIL in inferring the number and types of arguments. Then compare the saliency map obtained from ReSIL with the one in EKLAVYA to confirm the impact of our targeted changes in ReSIL. We further investigate the impact of instruction embedding on function signature inference, including the dimension of embedding vectors and the type of embedding. The implementation of the neural network remains the same as in EKLAVYA, while the data processing routine is written in Python with 1,850 lines of code which extracts the binary code for each function, inserts our special instructions for callees and callers, and corrects labels for callees that do not use some of their arguments.

We base our ground truth (the number and types of arguments) on information collected by an LLVM Lattner and Adve (2004) pass and on DWARF v4 debugging information Committee et al. (2010) which is the default setting for `gcc` and `clang`. We use LLVM to collect source-level information, including the number and types of arguments for each callee

and caller as well as their source line numbers. We then compile the test applications with DWARF information and link the source-level line numbers with binary-level addresses using the DWARF line number table. Different from EKLAVYA which obtains the ground truth by parsing the DWARF debug information, the ground truth we obtained is of finer-grain. For example, if one function has an argument whose type is a structure and its size is less than 16-bytes, it will be passed by two consecutive integer argument registers. In this case, ReSIL uses the ground truth by LLVM as the function has two arguments rather than one. Note that the ground truth is collected at the compiler intermediate level after optimization. That is, the ground truth we collected focus on how many arguments one function will use rather than the number of arguments it has. Inferred result is not 100% accurate, and usually we need human effort to help see whether the result is correct. In this case, human can only see how many arguments are used from the binary.

We use the same dataset described in Section 4.1 to perform the evaluation, in which a five-fold cross-validation is used to perform training and testing. Specifically, we randomly split each complication case in Table 7 into five folds (one used for testing and the remaining for training). For other callees (callers) which are not in the complication cases, we randomly split each utility package into five folds. Note that the training set contains all binaries compiled with multiple optimization levels from both compilers. The test results are reported on different categories of optimizations for different compilers. In order to ensure the generalizability of our approach, we also use a new training and testing set to see the effectiveness of our approach.

ReSIL and EKLAVYA share the same architecture, which is based on RNN. RNN is very effective for sequential data, as it mines time-series information and semantic information. In order to deal with the exploding and vanishing gradients during training Bengio et al. (1994), one could use an LSTM network or use an RNN model with gated recurrent units (GRUs). We use GRUs since it has the control to save or discard previous information and may train faster due to fewer parameters. Meanwhile, GRU further simplifies the gate structure, merges the forget gate and the input gate into an update gate.

## 6.1 Performance evaluation

### 6.1.1 Accuracy

Our goal is to evaluate the accuracy of inferences for the four tasks by using the approach we proposed in Section 5 to correct the complication scenarios we identified. Results are shown in Tables 8, 9 and 10, which are the average accuracy for the five folds under different compilers with different optimization levels. Note that we present the accuracy of ReSIL with its top five output, top one output, and top one output for coarse-grained type inference for comparison with EKLAVYA on argument types. Confidence scores are computed by calculating the geometric means using softmax probability with matrix scaling calibration Guo et al. (2017).

Table 8 shows that ReSIL (with its top one output) generally outperforms EKLAVYA with its most significant improvement in inferring the number of arguments at callees when optimization is enabled, especially for callees compiled by O1. This agrees with the statistics of the complication scenarios in Table 7 from which we can see that complications happen mostly in callees compiled with O1. The confidence score of ReSIL top one output is comparable with EKLAVYA .

The accuracy in identifying the type of an argument at the caller site is comparable with the result of EKLAVYA ; see Table 9. If we only use ReSIL's top one output as the inferred label,

**Table 8** Accuracy in inferring the number of arguments

Inst	Opt	App	Clang				Gcc			
			CI	T	Acc%	CS%	CI	T	Acc%	CS%
Caller	O0	R5	3,217	3,363	95.66	96.88	4,197	4,293	97.76	95.94
		E	3,157		93.87	99.00	4,182		97.41	98.99
		R1	3,169		94.23	99.57	4,168		97.09	99.69
	O1	R5	3,142	3,359	93.54	96.62	3,317	3,591	92.37	93.49
		E	2,936		87.41	98.80	3,252		90.56	98.39
		R1	3,058		91.04	99.19	3,202		89.17	98.69
	O2	R5	1,080	1,140	94.47	97.94	1,779	1,957	90.90	93.89
		E	1,039		91.14	99.01	1,735		88.66	98.55
		R1	1,051		92.19	99.56	1,734		88.61	98.70
	O3	R5	160	164	97.56	98.42	567	614	92.35	94.41
		E	155		94.51	99.28	571		93.00	98.72
		R1	158		96.34	99.66	555		90.40	99.09
Callee	O0	R5	2,006	2,022	99.21	99.55	2,181	2,209	98.73	98.78
		E	2,000		98.91	99.30	2,175		98.46	99.37
		R1	2,002		99.01	99.46	2,167		98.10	99.54
	O1	R5	1,936	2,089	92.68	95.12	1,444	1,542	93.64	95.78
		E	1,772		84.83	98.86	1,349		87.48	98.90
		R1	1,894		90.67	99.13	1,415		91.76	99.21
	O2	R5	756	797	94.86	97.01	1,063	1,136	93.57	95.86
		E	725		90.97	99.28	1,008		88.73	99.08
		R1	746		93.60	99.30	1,044		90.00	99.14
	O3	R5	152	159	95.60	96.32	406	425	95.53	96.57
		E	150		94.34	99.70	390		91.76	99.27
		R1	150		94.34	99.63	399		93.88	99.08

Numbers in the gray background show that ReSIL improves the accuracy in inferring the number of arguments even only using the top 1 output as the inference number

R5: ReSIL with top 5 outputs

E: EKLAVYA

R1: ReSIL with top 1 output

CI: Number of callers/callees correctly inferred

T: Number of callers/callees

CS: Confidence score

we can see that the accuracy of ReSIL in identifying the argument type in a finer-grained manner is a bit lower especially for binaries compiled by gcc. This is because there are many misidentifications among different types of integers. For binaries compiled by clang, the main misidentification comes from the indistinguishability between 64-bit integers and pointers.

We stress that this comparison between ReSIL and EKLAVYA isn't fair as ReSIL performs a much finer-grained inferencing of argument types. To establish a fairer comparison, we re-group the finer-grained types identified by ReSIL to be as close to that in EKLAVYA as possible, and present the results in the last three columns of Tables 9 and 10. We can see that now ReSIL has a comparable accuracy with EKLAVYA when inferring a coarse-grained type. It also shows that the insertion of our summary instructions helps argument type recovery as

**Table 9** Accuracy in inferring the type of argument at callers

CPL	Arg	Opt	ReSIL			EKLA VYA			R1			RC1			T
			CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS%	
Clang	1st	O0	3,649	96.89	98.43	3,645	96.79	99.60	3,612	95.91	99.69	3,625	96.26	99.65	3,766
		O1	3,027	96.56	98.79	3,004	95.88	99.62	3,003	95.79	99.75	3,023	96.42	99.72	3,133
		O2	1,033	98.38	98.86	1,015	96.76	99.81	1,021	97.23	99.86	1,025	97.52	99.84	1,049
	2nd	O3	160	98.16	99.24	160	98.16	99.82	160	98.16	99.91	161	98.77	99.87	163
		O0	2,108	92.70	95.79	2,065	90.81	99.19	2,041	89.75	99.23	2,059	90.64	99.16	2,274
		O1	1,722	90.68	96.00	1,647	86.91	99.29	1,658	87.31	99.23	1,688	88.89	99.13	1,895
	3rd	O2	637	94.51	96.61	631	93.76	99.50	616	91.39	99.56	618	91.69	99.50	673
		O3	123	96.85	97.33	123	96.85	99.64	121	95.28	99.69	121	95.28	99.61	127
		O0	1,177	91.17	96.34	1,142	88.46	99.10	1,151	89.16	99.32	1,169	89.16	99.19	1,291
Gcc	1st	O1	806	87.80	97.12	806	87.80	99.24	784	85.12	99.53	813	88.27	99.44	918
		O2	350	91.38	97.69	347	90.83	99.57	340	88.77	99.54	346	90.34	99.44	382
		O3	87	95.60	99.59	89	97.80	99.43	87	95.60	99.75	87	95.60	99.75	91
	2nd	O0	4,073	97.60	98.49	4,061	97.32	99.63	4,040	96.81	99.67	4,060	97.29	99.62	4,173
		O1	3,263	97.03	97.87	3,239	96.34	99.49	3,222	95.81	99.56	3,249	96.61	99.47	3,362
		O2	1,735	96.60	97.74	1,717	95.65	99.52	1,713	95.38	99.56	1,723	95.94	99.50	1,795
	3rd	O3	518	97.74	98.49	518	97.74	99.53	516	97.36	99.62	519	97.92	99.55	530
		O0	2,640	93.35	94.42	2,598	91.87	99.06	2,566	90.74	99.01	2,612	92.36	98.86	2,828
		O1	2,158	92.58	93.43	2,087	89.53	98.91	2,077	89.10	98.88	2,133	91.51	98.60	2,331
3rd	O2	1,081	91.84	93.74	1,068	90.89	99.11	1,038	88.19	98.80	1,070	91.08	98.58	1,175	
	O3	331	92.46	95.86	338	94.41	99.36	325	90.78	98.98	332	92.74	98.81	358	
	O0	1,458	90.67	94.39	1,466	91.17	98.93	1,402	88.18	99.10	1,475	91.73	98.85	1,608	
3rd	O1	1,294	91.00	95.22	1,298	91.28	98.83	1,220	89.25	99.04	1,304	91.70	98.80	1,422	
	O2	714	93.71	94.97	693	91.18	99.11	688	89.58	99.06	719	94.36	98.88	760	
	O3	224	94.51	96.65	223	94.09	99.40	216	92.31	99.57	227	95.78	99.46	237	

CI: Number of callers/callees correctly inferred

T: Number of callers/callees in the testing set

CS: Confidence score

R1: ReSIL with top1 output

RC1: ReSIL with top1 output and coarse-grained type

**Table 10** Accuracy in inferring the type of argument at callees

CPL	Arg	Opt	ReSIL			EKLA VYA			R1			RC1			T
			CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS%	
Clang	1st	O0	1,821	97.33	98.54	1,799	96.15	99.68	1,808	96.63	99.66	1,809	96.69	99.65	1,871
		O1	1,842	95.59	97.33	1,809	93.88	99.59	1,820	94.45	99.53	1,839	95.43	99.39	1,927
		O2	719	97.82	98.47	716	97.41	99.76	714	97.14	99.84	718	97.69	99.81	735
	2nd	O3	148	98.67	99.54	147	98.00	99.84	148	98.67	99.87	148	98.67	99.87	150
		O0	1,114	91.69	95.33	1,079	88.81	99.16	1,090	89.71	99.06	1,092	89.88	99.02	1,215
		O1	1,122	89.62	93.56	1,087	86.82	99.19	1,087	86.82	99.00	1,109	88.58	98.90	1,252
	3rd	O2	498	94.32	96.12	492	93.18	99.41	489	92.61	99.34	493	93.37	99.22	528
		O3	117	96.69	96.56	115	95.04	99.69	115	95.04	99.50	116	95.87	99.50	121
		O0	514	88.77	92.40	489	84.46	98.87	494	85.32	98.65	495	85.49	98.60	579
Gcc	1st	O1	530	86.60	93.07	514	83.99	98.96	512	83.66	98.71	524	85.62	98.50	612
		O2	243	91.70	95.54	240	90.57	99.17	238	89.81	99.36	241	90.94	99.22	265
		O3	60	90.91	94.70	59	89.39	99.49	58	87.88	99.40	60	90.91	99.12	66
	2nd	O0	1,988	97.02	97.93	1,963	95.80	99.65	1,969	96.10	99.58	1,974	96.34	99.57	2,049
		O1	1,376	96.90	97.90	1,360	95.77	99.68	1,362	95.92	99.68	1,372	96.62	99.53	1,420
		O2	1,023	97.71	97.86	1,012	96.66	99.67	1,013	96.75	99.56	1,015	96.94	99.53	1,047
	3rd	O3	381	97.69	98.52	381	97.69	99.74	379	97.18	99.66	380	97.44	99.61	390
		O0	1,213	91.41	94.49	1,170	88.17	99.08	1,182	89.12	89.37	1,186	89.07	99.10	1,327
		O1	879	92.14	95.06	851	89.20	99.12	858	89.94	99.16	867	90.88	99.05	954
3rd	O2	660	92.96	95.16	648	91.27	99.12	646	90.99	98.89	654	92.11	98.77	710	
	O3	279	95.22	96.02	278	94.88	99.62	274	93.52	99.53	275	93.86	99.45	293	
	O0	552	87.20	92.93	528	83.41	98.92	532	84.04	98.62	536	84.68	98.58	633	
3rd	O1	419	88.58	93.02	409	86.47	98.44	405	85.62	99.03	412	87.10	98.92	473	
	O2	327	90.58	93.84	320	88.64	99.23	317	87.81	98.67	322	89.20	98.49	361	
3rd	O3	146	93.59	95.61	143	91.67	99.30	143	91.67	99.04	144	92.31	99.05	156	

CI: Number of callers/callees correctly inferred  
 T: Number of callers/callees in the testing set  
 CS: Confidence score  
 R1: ReSIL with top 1 output  
 RC1: ReSIL with top 1 output and coarse-grained type

they give evidence to the RNN model on the number of bits of the argument that are being accessed.

Regarding recovery of argument types from the callee site, we can see that ReSIL has its more significant improvement in identifying types of the second and third arguments. Such improvement comes from the benefit of using the top five outputs. As discussed in Section 4.1, most of the first arguments are 32-bit integers and pointers which are immune to the complication scenarios we identified in Section 4.3.3; therefore, our accuracy in identifying them is much higher.

### 6.1.2 F1 scores

We also use F1 score to measure the performance of ReSIL for each class; see Table 11 and for the F1 scores improvement in different tasks.

Table 11a shows that ReSIL can effectively infer the number of arguments when callers have zero arguments. This is mainly due to the insertion of our summary instructions in wrapper functions that do not prepare arguments due to compiler optimization. ReSIL also performs better than EKLAVYA especially for callees compiled with optimizations and callees with more than six arguments. Moreover, we can see that ReSIL has a higher F1 score for callees which have zero arguments.

Table 11b shows that ReSIL improves the performance in distinguishing between 64-bit integers and pointers. For example, we can see that ReSIL improves the F1 score in inferring 64-bit integers for callers and callees compiled by Clang with optimization level O3 by 13.8% and 9.4% respectively.

### 6.1.3 Distribution of testing set

Table 12 shows the distribution of the testing dataset and in particular, the number of various cases that our testing dataset contains. For example, when evaluating the function signature inference for callees compiled by clang with O0 optimization flag, the testing set has 151 callees with 0 arguments, 656 callees with 1 argument, and 636 callees with 2 arguments.

We can find that there are few callees which have more than 6 arguments and there are few cases where the type of the argument is 16-bit integers, floating-point, or struct. This explains why the F1-score improvement in Table 11 for these cases is not meaningful.

### 6.1.4 Saliency map

To see if ReSIL manages to correct the complexity scenarios we identified earlier, here we perform a more in-depth saliency map analysis comparing the extent to which ReSIL and EKLAVYA make best use of **RG** and **WG** instructions. More specifically, we first perform the saliency map analysis on both models and extract the saliency scores of all **RG** and **WG** instructions within a callee or caller. Among these **RG** and **WG** instructions, we pick the one that has the highest saliency score within the callee or caller, and then present the box and count distribution plot in Fig. 13. Note that the saliency score is always normalized to 1.0 among all instructions within the callee or caller.

We can see from both the box plot and the count distribution that ReSIL values the **RG** and **WG** instructions more when inferring the number of arguments, which is exactly the problem we identified in EKLAVYA (see Section 4.2). Result is more apparent from

**Table 11** Improvement of F1 score

(a) Inferring the number of arguments

Inst	CPL	Opt	Number of Arguments									MF
			0	1	2	3	4	5	6	7	8	
Caller	Clang	O0	4.5	1.3	0.9	1.3	1.2	3.4	0.9	2.8	5.5	1.6
		O1	10.9	3.9	3.8	4.6	4.9	7.1	5.8	9.4	5.8	4.9
		O2	7.5	3.7	3.9	4.7	2.8	7.4	6.5	10.7	3.8	4.4
		O3	3.0	4.2	4.6	6.0	5.9	2.4	0.7	5.5	20.0	4.0
	Gcc	O0	5.2	0.3	-0.2	-0.3	-0.1	1.6	0.1	-0.2	-1.1	0.4
		O1	8.6	2.4	2.1	1.4	1.0	1.5	2.2	1.8	1.5	2.3
		O2	10.4	2.2	1.6	1.5	3.7	4.2	0.7	3.9	-0.5	2.5
		O3	2.9	0.0	0.5	0.8	3.4	1.9	3.0	7.3	22.2	0.9
Callee	Clang	O0	1.2	0.3	0.2	0.4	0.1	0.0	51.7	0.0	0.0	0.3
		O1	15.0	7.1	6.5	6.8	10.1	7.9	7.4	19.9	20.0	7.9
		O2	7.1	4.5	3.4	4.1	1.4	-0.7	5.7	-0.3	3.7	3.8
		O3	3.3	1.0	-0.2	0.9	2.9	1.4	9.5	19.4	-11.1	1.4
	Gcc	O0	0.6	0.4	0.4	0.0	-0.4	0.5	22.2	-50.0	0.0	0.3
		O1	15.6	6.3	5.1	5.3	4.9	3.3	4.5	10.8	6.1	6.1
		O2	10.0	4.7	4.3	4.3	5.1	5.3	0.9	3.5	10.7	4.9
		O3	7.1	3.8	3.0	2.4	5.4	3.5	4.5	18.5	14.0	3.7

(b) Inferring types of arguments

Inst	CPL	Opt	Type of Arguments							MF
			int16	int32	int64	float	pointer	struct		
Caller	Clang	O0	4.8	7.5	3.7	3.9	0.00	0.9	-	1.9
		O1	8.2	-16.7	3.6	6.1	0.00	1.7	-	2.8
		O2	0.7	0.00	4.8	8.3	-	2.1	-	3.5
		O3	25.0	-	0.9	13.8	-	0.8	-	1.6
	Gcc	O0	4.7	8.7	2.7	5.6	0.00	1.5	29.7	2.6
		O1	8.3	30.5	3.6	4.4	-	1.8	40.7	2.9
		O2	5.6	0.0	1.1	3.6	-	1.4	3.3	1.9
		O3	-5.2	-	1.5	3.3	-	0.8	-	1.2
Callee	Clang	O0	4.4	16.7	2.4	7.7	56.7	1.3	10.2	0.4
		O1	15.4	-8.3	4.8	5.9	33.3	1.5	3.3	0.7
		O2	7.9	100.0	3.5	5.1	0.0	1.1	-88.9	0.2
		O3	-25.0	-	4.4	9.4	-	1.8	-	1.8
	Gcc	O0	0.42	-10.0	2.7	6.4	51.1	1.4	-69.1	0.1
		O1	16.0	0.0	5.4	4.2	-50.0	1.4	-11.1	0.9
		O2	6.2	16.7	2.9	6.5	-16.7	1.2	33.3	0.4
		O3	3.1	-	0.7	-1.8	0.0	-0.2	-	-0.8

The number is calculated by  $(F1_{ReSIL} - F1_{EKLAVYA})\%$



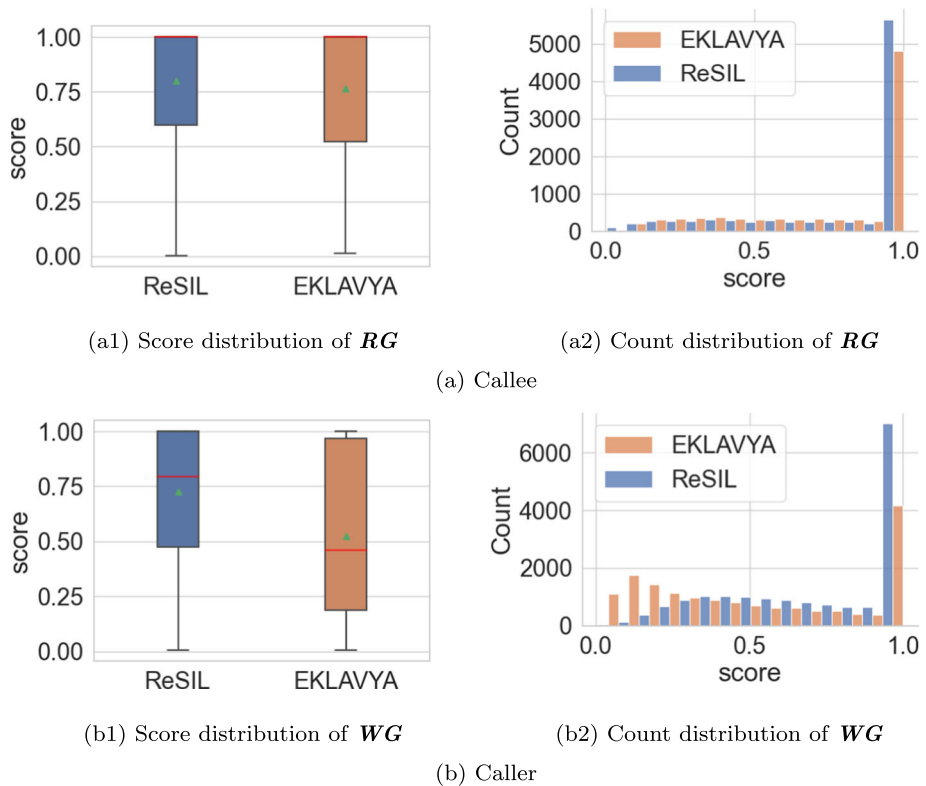
**Table 12** Distribution of testing dataset

(a) Number of arguments

Inst	CPL	Opt	Number of Arguments								
			0	1	2	3	4	5	6	7	8
Caller	Clang	O0	258	1,064	984	496	299	126	66	31	19
		O1	221	928	929	692	304	130	82	36	15
		O2	74	305	230	272	147	38	40	16	8
		O3	5	53	21	21	22	9	22	2	2
	Gcc	O0	246	1,267	1,188	948	318	154	89	39	24
		O1	209	986	998	799	314	139	74	36	20
		O2	95	704	371	486	140	91	36	15	12
		O3	24	213	145	143	42	18	12	3	1
Callee	Clang	O0	151	656	636	366	144	67	1	0	1
		O1	145	639	646	379	150	70	30	15	7
		O2	41	200	278	150	67	31	16	7	4
		O3	9	29	55	30	19	9	5	1	1
	Gcc	O0	161	722	693	402	156	73	1	1	1
		O1	102	449	500	289	110	53	20	8	6
		O2	78	325	363	216	81	42	15	8	5
		O3	25	97	142	92	35	18	8	4	2

(b) Types of arguments

Inst	CPL	Opt	Type of Arguments						
			int8	int16	int32	int64	float	pointer	struct
Caller	Clang	O0	20	3	289	334	2	1,609	-
		O1	18	3	326	301	1	1,229	-
		O2	3	2	86	115	-	467	-
		O3	3	-	21	7	-	96	-
	Gcc	O0	43	4	621	420	2	1,690	2
		O1	16	4	515	384	-	1,369	2
		O2	12	2	242	245	-	662	1
		O3	5	-	59	50	-	239	-
Callee	Clang	O0	17	2	222	141	2	818	3
		O1	17	2	229	145	2	844	3
		O2	4	1	88	55	1	376	1
		O3	2	-	15	13	-	91	-
	Gcc	O0	14	6	195	167	2	748	5
		O1	10	3	167	105	1	659	2
		O2	8	2	120	82	1	491	2
		O3	3	-	55	33	1	201	-

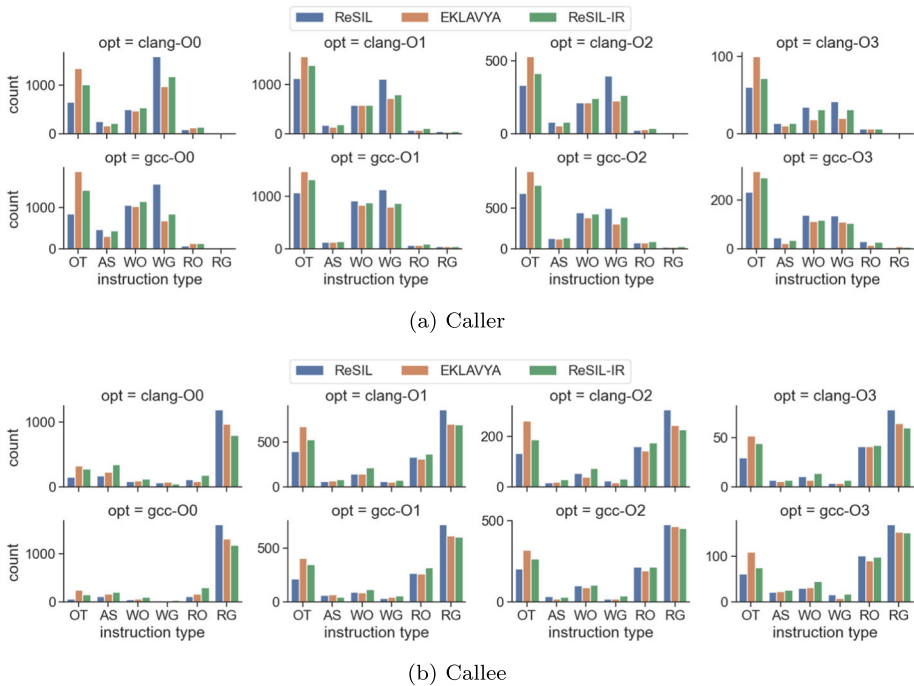


**Fig. 13** Distribution of  $RG$  and  $WG$  with largest score at callee and caller sites

the count distribution — ReSIL has almost two times of the cases of EKLAVYA with the corresponding saliency score equal to 1.0.

We also compare the types of the most important instruction in ReSIL with the ones in EKLAVYA ; see Fig. 14. We can find that ReSIL considers  $RG$  and  $WG$  as the most important instructions for most callees and callers, as opposed to EKLAVYA considering  $OT$  as the most important ones. The interesting part is that even though the accuracy of ReSIL has not improved a lot in inferring the number of arguments at the caller site as shown in Table 8, it makes the RNN focus more on  $WG$  and less on  $OT$  compared to EKLAVYA as shown in Fig. 14a.

One may argue that this could be due to the policy to remove the irrelevant instructions (see Section 5.3) rather than the policy to insert summarized instructions and to correct the labels for *Unread* (see Section 5.1). To check the relative effectiveness of these two policies, we show the most important instructions at the callee and caller sites when only irrelevant instructions are removed (without inserting summarized instructions); see the bar for ReSIL-IR in Fig. 14. We can find that even though some irrelevant instructions are removed, the RNN model has not improved significantly in terms of identifying important instructions. An interesting observation is that when only removing irrelevant instructions,  $RO$  will be considered as the most important instructions for most callees. It also demonstrates that our policy for *Helper* and *Unread* further makes the RNN model focus on the inserted summary instructions and  $RG$ .



**Fig. 14** Types of the most important instructions in ReSIL. *ReSIL-IR* refers to the model which only removes irrelevant instructions

### 6.1.5 Handling of complication scenarios

We also evaluate the effectiveness of ReSIL in dealing with different complication scenarios. Specifically, about 90% of callees with *Unread* that are incorrectly inferred by EKLAVYA are now correctly inferred by ReSIL. For callees with *Helper* which are incorrectly inferred by EKLAVYA, ReSIL can correctly identify the number of arguments for around 50% of them. Note that EKLAVYA is able to correctly infer the number of arguments for the majority of the callers with *Temp*, but it would misidentify registers that are used to store temporary values rather than passing arguments. Such inaccuracy is mitigated by ReSIL with top five outputs. Meanwhile, nearly all callers with *Wrapper* that are incorrectly inferred by EKLAVYA are correctly recovered by ReSIL.

The result seems to suggest that ReSIL does not perform well with *Helper*. To get a better idea if the relatively small improvement comes from the ineffectiveness of ReSIL or the nondeterminism introduced in the RNN training process, we evaluate ReSIL and EKLAVYA by repeating the 5-fold cross validation process ten times. We consider the signature recovered (in)correctly if all these ten runs give (in)correct results. The result can be found Table 13.

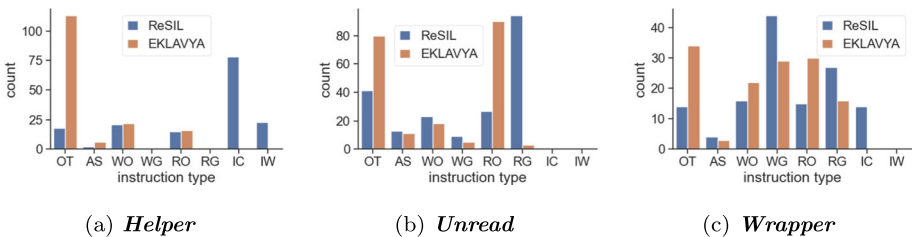
From the analysis result, we find that ReSIL can correctly infer the number of arguments for all callees with *Helper* and *Unread* in the testing set. Meanwhile, ReSIL provides a limited improvement in inferring the number of arguments for other callees since the cases of *Helper* and *Unread* will confuse the RNN model to incorrectly infer the signature for them. We also find that ReSIL can correctly identify all *Wrapper* cases which are wrongly inferred by EKLAVYA. Moreover, ReSIL can also help identify function signature for callers which do not have any arguments.

**Table 13** Number of complication scenarios incorrectly recovered

	Opt	Fold 1			Fold 2			Fold 3			Fold 4			Fold 5		
		E	R5	R1	E	R5	R1	E	R5	R1	E	R5	R1	E	R5	R1
<b>Unread</b>	O0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O1	129	17	19	141	16	19	123	10	16	124	20	30	132	14	19
	O2	33	3	3	37	3	4	35	7	9	31	4	7	37	4	4
	O3	7	1	1	6	1	2	3	0	0	10	0	0	7	1	1
<b>Helper</b>	O0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O1	64	38	49	72	30	34	67	32	39	78	35	45	54	35	46
	O2	14	5	6	18	11	11	14	10	10	19	13	16	13	10	11
	O3	3	1	1	1	2	2	1	0	0	0	1	1	1	0	0
<b>Temp</b>	O0	4	3	3	6	6	9	8	1	5	4	2	5	1	2	7
	O1	4	0	0	1	1	1	8	5	5	6	3	6	6	4	1
	O2	2	0	0	2	0	1	1	0	0	3	1	1	3	0	2
	O3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
<b>Wrapper</b>	O0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O1	14	1	1	9	0	0	17	0	0	9	1	2	4	1	1
	O2	1	0	0	1	0	0	2	0	0	0	0	0	0	0	0
	O3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

E: EKLAVYA  
 R5: ReSIL with top 5 output  
 R1: ReSIL with top 1 output

The distribution of instruction types for cases **Helper**, **Unread** and **Wrapper** can be found in Fig. 15. We can find that the inserted summary instructions (*ic*, *iw*) are considered as the most important instruction for case **Helper**. For case **Wrapper** which actually does not have **WG**, ReSIL considers *ic* as the most important instructions. When the RNN model outputs the number of arguments used by a function rather than the number of arguments the function has, instructions which read the ground-truth argument register (**RG**) are more important as shown in Fig. 15b.



**Fig. 15** Most important instruction types for **Helper**, **Unread**, and **Wrapper** in ReSIL. IC refers to the inserted summary instructions with correct prediction result; IW refers the inserted instructions with incorrect prediction result

### 6.1.6 Time consumption

We also compare the time consumption between EKLAVYA and ReSIL in the 5-fold cross-validation training and testing stages, the result is shown in Table 14. Training represents the average time in seconds that the model takes to train one epoch. Testing represents the testing time for one function in milliseconds.

We also compared the time consumption of EKLAVYA and ReSIL with Nimbus at the last row of Table 14. Since Nimbus is not open source, we just cited the number from the Nimbus paper. The result shows that Nimbus is more efficient due to the fact that MTL merges the task-specific layers into the shared layer, thus avoiding duplicate computations. We can also see that ReSIL takes less time for training and testing since it removes some irrelevant instructions as mentioned in Section 4.3.4.

## 6.2 Impact of instruction embedding

In this section, we further investigate the impact of instruction embedding on the effectiveness of function signature inference, including the dimension of the instruction embedding vector and the type of instruction embedding approach.

### 6.2.1 Embedding dimensions

We modify the embedding dimension to further explore its impact on accuracy and the time required for both training and testing stages. We experiment with four common instruction embedding dimensions: 64, 128, 256, and 512, which are also widely used in other deep learning-based approaches. These four models are evaluated for inferring the number of arguments at callee sites, and the results are presented in Fig. 16.

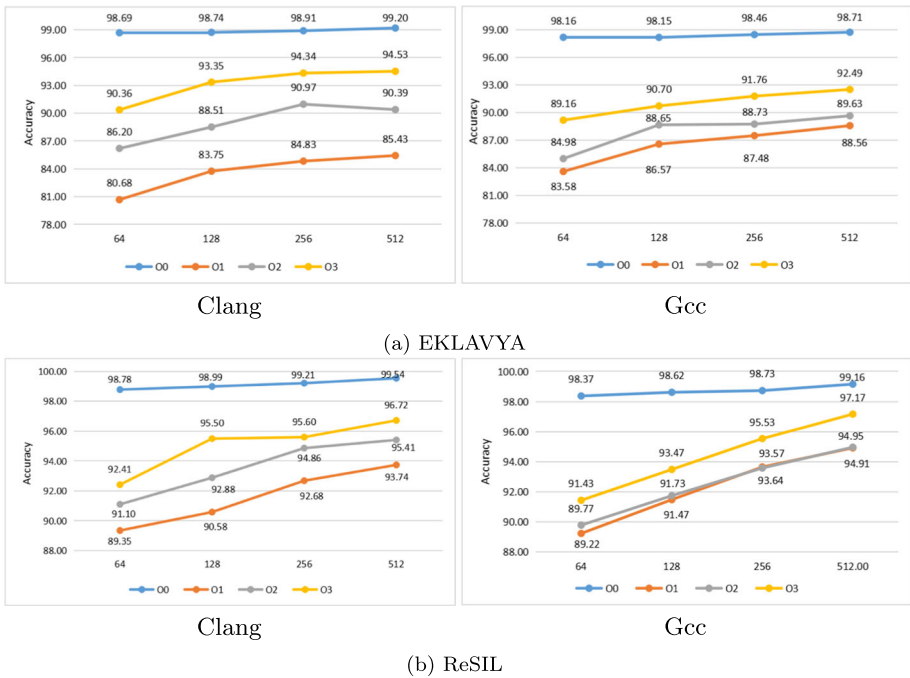
From the results, we can observe that there is a clear trend that the performance becomes better when increasing the embedding size. The largest embedding size has the best performance both in EKLAVYA and ReSIL. However, considering efficiency, we recommend having a suitable embedding dimension configuration. Here, we use 256 as the embedding dimension. This choice is based on the observation that the accuracy in inferring the number of arguments doesn't significantly increase, but training and testing times considerably increase when the embedding dimension is set to 512, as shown in Fig. 17.

### 6.2.2 Type of instruction embedding

In this section, we further test the performance of different types of embeddings in inferring the number of arguments at callee sites using EKLAVYA and ReSIL as the downstream application. Specifically, we replaced the original Word2Vec embedding in EKLAVYA and

**Table 14** Comparison on training and testing time

Approach	Training (s)	Testing (ms)
EKLAVYA	217.41	22.83
ReSIL	210.47	20.40
Nimbus	94.84	2.20

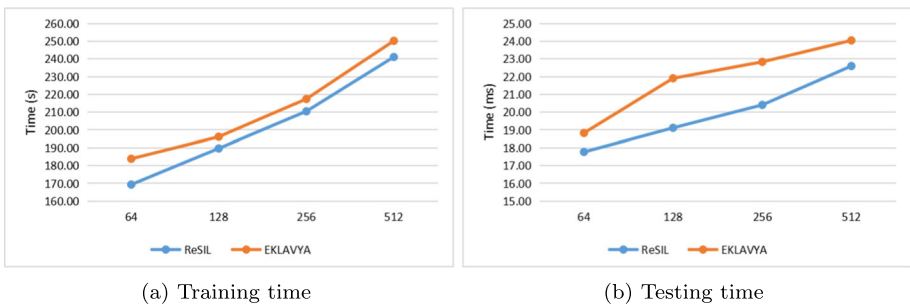


**Fig. 16** Accuracy in inferring the number of arguments at callee sites under different embedding dimensions

ReSIL with the BERT embedding, as used by Li et al. (2021). We compare the accuracy of inferring the number of arguments at callee sites in Fig. 18.

We can observe that using instruction embedding generated by BERT can improve the accuracy of inferring the number of arguments, but the accuracy for binaries compiled with optimization is still not good for EKLAVYA. For example, the accuracy for binaries compiled by clang -O1 is only around 86.89%. Meanwhile, we can find ReSIL can also work for instruction embedding generated by BERT and further improves the accuracy.

We also compare the distribution of box and count scores for the RG instructions with the highest scores when using BERT instruction embedding for both EKLAVYA and ReSIL, as



**Fig. 17** Training and testing times in inferring the number of arguments at callee sites under different embedding dimensions

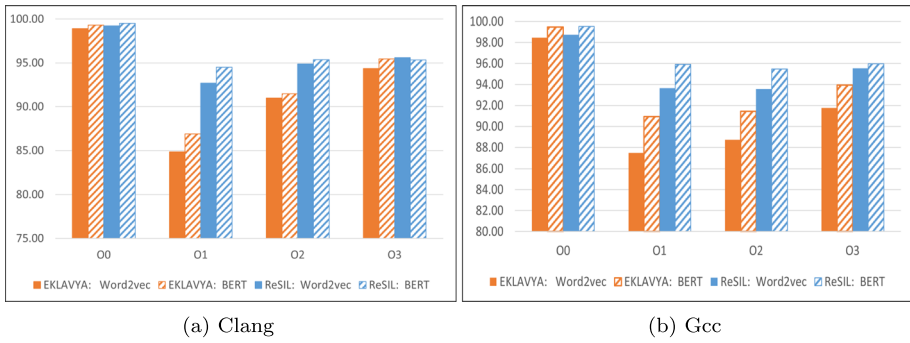


Fig. 18 Accuracy in inferring the number of arguments at callee sites under different types of instruction embedding

shown in Fig. 19. We observe that ReSIL assigns a higher value to the RG instructions when inferring the number of arguments with BERT instruction embedding.

### 6.3 Generalizability of ReSIL

To thoroughly evaluate the generalizability of the trained model across various inputs, we employed entirely distinct and more diverse training and testing sets, compiled them using different compiler versions, and documented the dataset details in Table 15. The dataset includes programs of diverse functionality and complexity, written in C, and containing hand-written assembly and intentional security vulnerabilities. It also carries a significant amount of complications described in Section 4.3.

In direct comparison to the original dataset utilized by EKLAVYA, this new dataset stands out for its heightened diversity. The inclusion of hand-written assembly code and a broader range of intentional security vulnerabilities introduces additional dimensions of complexity. This diversity is pivotal for evaluating the generalizability and robustness of ReSIL.

Specifically, the new dataset comprises 73,765 distinct functions, and we also use 5-fold cross-validation to perform training and testing on those functions. Table 16 presents the accuracy of recovering the number of arguments at callee sites. As shown, even with this new dataset, ReSIL consistently outperforms EKLAVYA, when using only the top 1 output for inference. This further validates the robustness of our approach.

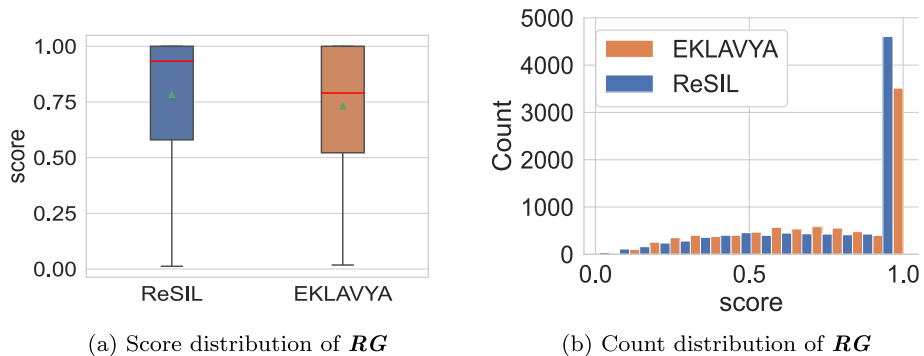


Fig. 19 Distribution of RG with the largest score at callee sites when BERT instruction embedding is used

**Table 15** New applications used in training and testing

Type	Name	Programs/Binaries
Servers	Vsftpd-3.0.3 Uftpd-2.15 Sqlite3-3.30.1 Zaver-0.1	4/32
Clients	Cflow-1.4 Gzip-1.3.5 Bcrypt-1.1 FastLZ Http-parser MiniUnz-1.01b Iperf-3.11 Mdp-1.0.15 Nuklear- Vcut Vorbis-tools-1.4.0	28/224
Benchmark	SPECCPU-2006 LLVM test-suite CGC Challenge Binaries	112/896
Total		144/1,152

## 6.4 Security applications of ReSIL

In this subsection, we look into a specific application of ReSIL and evaluate the extent to which ReSIL could be used to improve the effectiveness and security of CFI enforcement.

After we recover the function signature at both the indirect caller and callee sites, we can enforce CFI by allowing only control transfers from indirect callers to callees with matching signature. Specifically, we discuss how ReSIL can be used to defend against practical Counterfeit Object-oriented Programming (COOP) Schuster et al. (2015). It is a security attack on object-oriented systems, where attackers exploit vulnerabilities to manipulate objects, potentially causing unintended behavior or data corruption. We stress that our purpose in this subsection is to provide examples of security applications in which ReSIL plays a critical part. We leave a more systematic coverage of all security applications as our future work.

### 6.4.1 Effectiveness against COOP

By exploiting a memory corruption vulnerability, COOP diverts execution flows to a chain of existing virtual function calls (so-called vfgadgets) via an initial vfgadget. The original COOP paper Schuster et al. (2015) proposes two main types of initial vfgadgets, the main-loop gadget (ML-G) and the recursive gadget (REC-G). Such gadgets are responsible for dispatching the vfgadget chain using virtual function calls. We use the published exploits for Firefox Schuster et al. (2015) to show how ReSIL could stop such an exploit.

The details about the gadgets can be found in Fig. 20. Function `nsMultiplexInputStream::Close` is used as the ML-G gadget, while when we

**Table 16** Accuracy in inferring the number of arguments at the callee site with the new dataset

Approach	Clang				Gcc			
	O0	O1	O2	O3	O0	O1	O2	O3
R5	99.44	92.19	92.35	94.17	98.49	92.34	92.11	91.22
E	98.53	81.95	85.78	88.99	96.50	84.49	84.44	83.42
R1	99.25	89.30	89.61	91.88	98.00	89.64	89.34	88.29

R5: ReSIL with top 5 outputs

E: EKLAVYA

R1: ReSIL with top 1 output



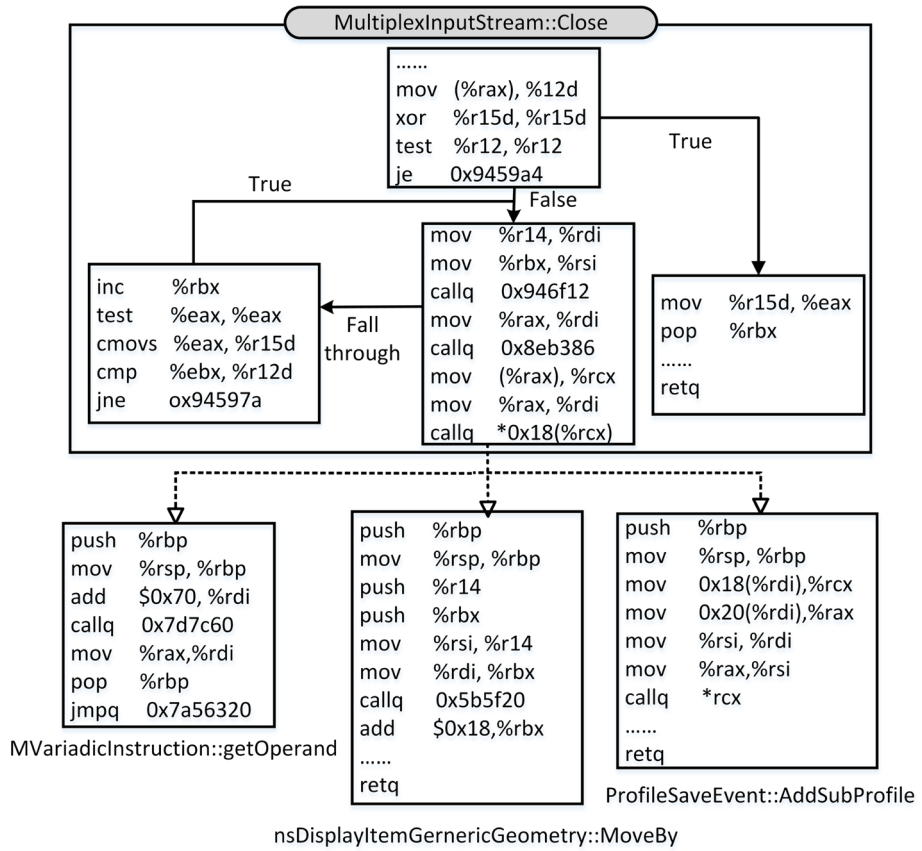


Fig. 20 Gadgets used in COOP's 64-bit Firefox exploit

use ReSIL to recover the number of arguments for the indirect callers in it, the result suggests that it has one argument, which is the top one output of ReSIL. However, the inferred number of arguments for other three functions is two using ReSIL with the top one output. Therefore, the target of this indirect caller cannot be any of the three functions, suggesting that ReSIL successfully stops the Firefox COOP exploit.

When we use EKLAVYA to infer the number of arguments for function `MVariadicInstruction::getOperand`, the result is that it only has one argument as the access of the second argument is in the helper function at address `0x7d7c60`. The inter-procedural analysis performed by ReSIL can identify the instruction that accesses the second argument and the summary instruction inserted by ReSIL enables the RNN model to correctly recover the number of arguments for it.

### 6.4.2 Other applications of ReSIL

In addition to applying ReSIL to help CFI enforcement, ReSIL can also be used to help binary code reuse, fuzzing, function reuse detection, and malware similarity analysis. Reusing binary

code is useful especially in scenarios where the source code is not available. Extraction of a functional code block (e.g., function) allows a programmer to add that functionality directly to other applications, and function signature is required so that we know how to invoke this code block Caballero et al. (2009). In addition, as shown by Jain et al. (2018), the inferred function signature can be used to improve fuzzing and trigger bugs much earlier than existing solutions. Function signature (number of arguments and argument types) is also an important feature to help perform malware similarity and function reuse analysis (Nouh et al. 2017; Kim et al. 2020).

## 7 Discussion

### 7.1 Adopting ReSIL to other platforms

Now, the focus of the paper is on function signature recovery of x86-64 Linux binaries. However, the same technique can also be applied to other platforms, such as Windows and ARM64 architectures. These new platforms may encounter new issues since the calling conventions, argument passing mechanisms, and register allocation strategies are different. For example, the calling convention of Windows with non-overlapping register-indices for both floating-point and integer parameters within a function may further complicate the function signature inference.

### 7.2 Limitations of ReSIL

In this section, we discuss a number of limitations of ReSIL.

#### 7.2.1 Limitations of the GRU-Based Approach

The use of GRU in ReSIL might introduce certain limitations, such as the handling of long-range dependencies, scalability to larger datasets, and capturing complex contextual relationships within scenarios. Adopting a transformer-based architecture in ReSIL could potentially address some of the limitations associated with the current GRU-based approach. For example, transformers can be scaled effectively to handle larger datasets and more complex scenarios, allowing for the exploration of a broader range of possibilities and transformers' attention mechanism naturally handles long-range dependencies, which can be crucial when considering scenarios with extended time horizons or complex causal chains. While transformers offer several advantages, they also come with their own challenges, including increased computational requirements and potential overfitting on small datasets.

#### 7.2.2 Limitations in inferring function signatures

By analyzing the results, we notice a number of limitations of ReSIL in inferring function signatures.

- **Callees (Callers) with more than six arguments.** We find it more difficult for ReSIL to correctly infer the number of arguments for a callee (caller) with more than 6 arguments. This is because the higher-order arguments are passed onto the stack and typically

accessed in the later part of a callee. At the caller, the compiler always uses register `%rsp` plus some displacements to pass them. When optimization is enabled, local variables are also accessed using `%rsp` plus some displacements. This causes ReSIL to misclassify some argument-preparing instructions as storing local variables.

- **Callees (Callers) whose arguments are accessed (prepared) by instructions of large sizes.** It is more difficult for ReSIL to correctly identify arguments accessed (prepared) with instructions more than five bytes long. For example, instruction `mov %rdi, 0x25e088(%rip)` has seven bytes, and ReSIL cannot correctly infer that there is a reading operation on the first argument register `%rdi` and this causes the number of arguments to be underestimated.
- **Callees with complex functionalities.** We find it easier for ReSIL to correctly identify arguments which are accessed in the first two to three basic blocks of callees. This is also the limitation of RNN which does not have good performance for long sentences.
- **Callers whose argument-preparing instructions precede `ret`, `call`, and `direct jump instructions`.** It appears that ReSIL does not consider argument-preparing instructions that precede a `ret`, `call`, and `jump` instruction in recovering the number of arguments.

In addition to these limitations, we also find cases where EKLAVYA infers the function signature correctly while ReSIL cannot, but the number of these cases is quite small (fewer than 10 in our dataset). We show one example of such cases in Fig. 21. Here, callee `stringer` has one argument and the access of the argument (`%dil`) is after the call instruction at Line 5. ReSIL infers that this callee has zero arguments while EKLAVYA outputs one. This is because ReSIL finds that there are no summarized instructions inserted before the call instruction at Line 5, and it (incorrectly) outputs that this callee has zero arguments.

## 8 Artifact availability

- (1) **Data.** Our corpus of binary variants and their associated function signature output is available at: [https://1drv.ms/f/s!Aj9CYr\\_j\\_6FAoM5x2fHA9J0whrpyYw?e=34pI8z](https://1drv.ms/f/s!Aj9CYr_j_6FAoM5x2fHA9J0whrpyYw?e=34pI8z)
- (2) **Source Code.** The source code for instruction embedding, model training and testing is available at: [https://1drv.ms/f/s!Aj9CYr\\_j\\_6FAoM51VbxaL5xQ8I61FQ?e=CDkhW3](https://1drv.ms/f/s!Aj9CYr_j_6FAoM51VbxaL5xQ8I61FQ?e=CDkhW3)
- (3) **Trained Model.** The trained model for different tasks is available at: [https://1drv.ms/f/s!Aj9CYr\\_j\\_6FAoNBpzELCBZsjh7tBTw?e=iZYpol](https://1drv.ms/f/s!Aj9CYr_j_6FAoNBpzELCBZsjh7tBTw?e=iZYpol)

```

1 00000000041e810 <stringer>:
2 #18 instructions
3 41e84e: je      41e8b8 <stringer+0xa8>
4 #7 instructions
5 41e86b: call   402070
6 #17 instructions
7 41e8b8: mov   $0x2c,%eax
8 #3 instructions
9 41e8ce: test  %dil,0x1

```

**Fig. 21** Number of arguments inferred correctly by EKLAVYA but wrongly by ReSIL

## 9 Conclusion

In this paper, we study the underlying reasons why state-of-the-art deep learning approaches suffer lower accuracy in recovering function signatures from optimized binaries, and propose ReSIL which incorporates compiler-optimization-specific domain knowledge into the samples. Experimental results show that ReSIL effectively improves the accuracy and F1 score in identifying function signatures. Our evaluation also shows that ReSIL effectively improves security of CFI enforcement.

**Acknowledgements** This work of Yan Lin was supported by the Fundamental Research Funds for the Central Universities (Grant No. 21623342) and the National Natural Science Foundation of China (Grant No. 62302193). This work of Debin Gao was supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Software Systems - Core Technologies Grant (TSSCTG) award no. NSOE-TSS2019-02.

**Data Availability Statements** The authors confirm that the data supporting the findings of this study are available within the article. Raw data that support the findings of this study are available from the corresponding author, upon reasonable request.

## Declarations

**Conflicts of Interests** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper

## References

- Balakrishnan G, Reps T (2007) Divine: Discovering variables in executables. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, pp 1–28
- Bao T, Burket J, Woo M, Turner R, Brumley D (2014) *byteweight*: Learning to recognize functions in binary code. In: Proceedings of the 23rd USENIX Security Symposium, pp 845–860
- Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. *IEEE Trans Neural Netw* 5(2):157–166
- Caballero J, Johnson NM, McCamant S, Song D (2009) Binary code extraction and interface identification for security applications. California Univ Berkeley Dept of Electrical Engineering and Computer Science, Tech. rep
- Chen L, He Z, Mao B (2020) Cati: Context-assisted type inference from stripped binaries. In: Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, pp 88–98
- Chua ZL, Shen S, Saxena P, Liang Z (2017) Neural nets can learn function type signatures from binaries. In: Proceedings of the 26th USENIX Security Symposium, pp 99–116
- Committee DDIF et al (2010) Dwarf debugging information format, version 4. Free Standards Group
- Duan Y, Li X, Wang J, Yin H (2020) Deepbindiff: Learning program-wide code representations for binary diffing. In: Network and Distributed System Security Symposium
- ElWazeer K, Anand K, Kotha A, Smithson M, Barua R (2013) Scalable variable and data type detection in a binary rewriter. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, pp 51–60
- Fu C, Chen H, Liu H, Chen X, Tian Y, Koushanfar F, Zhao J (2019) Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems* 32
- Guo C, Pleiss G, Sun Y, Weinberger KQ (2017) On calibration of modern neural networks. In: International Conference on Machine Learning, PMLR, pp 1321–1330
- He J, Balunović M, Ambroladze N, Tsankov P, Vechev M (2019) Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp 531–548

- He J, Ivanov P, Tsankov P, Raychev V, Vechev M (2018) Debin: Predicting debug information in stripped binaries. In: Proceedings of the 25th ACM Conference on Computer and Communications Security, ACM, pp 1667–1680
- Hellendoorn VJ, Bird C, Barr ET, Allamanis M (2018) Deep learning type inference. In: Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 152–162
- Hu Y, Zhang Y, Li J, Gu D (2017) Binary code clone detection across architectures and compiling configurations. In: Proceedings of the 25th International Conference on Program Comprehension, IEEE, pp 88–98
- INTEL I (2018) Intel® 64 and ia-32 architectures software developer's manual
- Jain V, Rawat S, Giuffrida C, Bos H (2018) Tiff: using input type inference to improve fuzzing. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACM, pp 505–517
- Ji Y, Cui L, Huang HH (2021) Vestige: Identifying binary code provenance for vulnerability detection. In: International Conference on Applied Cryptography and Network Security, Springer, pp 287–310
- Katz DS, Ruchti J, Schulte E (2018) Using recurrent neural networks for decompilation. 2018 IEEE 25th International Conference on Software Analysis. Evolution and Reengineering (SANER), IEEE, pp 346–356
- Katz O, Olshaker Y, Goldberg Y, Yahav E (2019) Towards neural decompilation. arXiv preprint [arXiv:1905.08325](https://arxiv.org/abs/1905.08325)
- Kim D, Kim E, Cha SK, Son S, Kim Y (2020) Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. [arXiv:2011.10749](https://arxiv.org/abs/2011.10749)
- Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd international symposium on Code generation and optimization, IEEE
- Lee J, Avgerinos T, Brumley D (2011) Tie: Principled reverse engineering of types in binary programs. In: Proceedings of the 18th Network and Distributed System Security Symposium
- Liang R, Cao Y, Hu P, Chen K (2021) Neutron: an attention-based neural decompiler. *Cybersecurity* 4(1):1–13
- Lin Y, Cheng X, Gao D (2019) Control-flow carrying code. In: Proceedings of the 14th ACM Asia Conference on Computer and Communications Security, ACM, pp 3–14
- Lin Y, Gao D (2021) When function signature recovery meets compiler optimization. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy, IEEE
- Li X, Qu Y, Yin H (2021) Palmtree: Learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp 3236–3251
- Maier A, Gascon H, Wressnegger C, Rieck K (2019) Typeminer: Recovering types in binary programs using machine learning. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp 288–308
- Muntean P, Fischer M, Tan G, Lin Z, Grossklags J, Eckert C (2018)  $\tau$ cfi: Type-assisted control flow integrity for x86-64 binaries. In: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, pp 423–444
- Nouh L, Rahimian A, Mouheb D, Debbabi M, Hanna A (2017) Binsign: fingerprinting binary functions to support automated analysis of code executables. In: IFIP International Conference on ICT Systems Security and Privacy Protection, Springer, pp 341–355
- Otsubo Y, Otsuka A, Mimura M, Sakaki T, Ukegawa H (2020) o-glassesx: compiler provenance recovery with attention mechanism from a short code fragment. In: Proceedings of the 3rd Workshop on Binary Analysis Research
- Pei K, Guan J, Broughton M, Chen Z, Yao S, Williams-King D, Ummadisetty V, Yang J, Ray B, Jana S (2021) Stateformer: fine-grained type recovery from binaries using generative state modeling. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 690–702
- Pizzolotto D, Inoue K (2020) Identifying compiler and optimization options from binary code using deep learning approaches. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, IEEE, pp 232–242
- Prakash A, Hu X, Yin H (2015) vfguard: Strict protection for virtual function calls in cots c++ binaries. In: Proceedings of the 22nd Network and Distributed System Security Symposium
- Qian Y, Chen L, Wang Y, Mao B (2022) Nimbus: Toward speed up function signature recovery via input resizing and multi-task learning. 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), IEEE, pp 454–463
- Rosenblum N, Miller BP, Zhu X (2011) Recovering the toolchain provenance of binary code. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ACM, pp 100–110

- Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi AR, Holz T (2015) Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: Proceedings of the 36th IEEE Symposium on Security and Privacy, IEEE, pp 745–762
- Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D (2017) Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE international conference on computer vision, pp 618–626
- Sharma A, Tian Y, Lo D (2015) Nirmal: Automatic identification of software relevant tweets leveraging language model. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering, IEEE, pp 449–458
- Shin ECR, Song D, Moazzezi R (2015) Recognizing functions in binaries with neural networks. In: Proceedings of the 24th USENIX Security Symposium, pp 611–626
- Simonyan K, Vedaldi A, Zisserman A (2013) Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint [arXiv:1312.6034](https://arxiv.org/abs/1312.6034)
- Tian Z, Huang Y, Xie B, Chen Y, Chen L, Wu D (2021) Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access* 9:49160–49175
- Van Der Veen V, Göktaş E, Contag M, Pawoloski A, Chen X, Rawat S, Bos H, Holz T, Athanasopoulos E, Giuffrida C (2016) A tough call: Mitigating advanced code-reuse attacks at the binary level. In: Proceedings of the 37th IEEE Symposium on Security and Privacy, IEEE, pp 934–953
- Wang S, Wang P, Wu D (2017) Semantics-aware machine learning for function recognition in binary code. In: Proceedings of the 33rd International Conference on Software Maintenance and Evolution, IEEE, pp 388–398
- Xu Z, Wen C, Qin S (2018) Type learning for binaries and its applications. *IEEE Transactions on Reliability* 68(3):893–912
- Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp 363–376
- Zeng D, Tan G (2018) From debugging-information based binary-level type inference to cfg generation. In: Proceedings of the 8th ACM Conference on Data and Application Security and Privacy, ACM, pp 366–376

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.



**Yan Lin** received her Ph.D. degree in computer science from Singapore Management University. She is currently a pre-tenure Associate Professor at the College of Cyber Security, Jinan University. Her current research focuses on system security, software security, and mobile security.



**Trisha Singhal** received the B.Tech degree in Computer Science and Engineering from Dr. A.P.J. Abdul Kalam University, India in 2019. She is working as a Data Scientist in Optum, Inc. (UnitedHealth Group). Her research interests include Natural Language Processing, Computer Vision, and Multimodal Learning.




**Debin Gao** is currently an Associate Professor from School of Computing and Information Systems, Singapore Management University. Having obtained his Ph.D degree from Carnegie Mellon University in 2006, Debin focuses his research on software and systems security. In recent years, Debin also actively participated in research of mobile security, cloud security, and human factors in security. Debin received the best paper award from NDSS in 2013.



**David Lo** is an Professor of Computer Science in the School of Computing and Information Systems, Singapore Management University (SMU). He received his Ph.D. in Computer Science from the National University of Singapore. His research interests include software analytics, software maintenance, empirical software engineering, and cyber security.

## Authors and Affiliations

Yan Lin<sup>1</sup>  · Trisha Singhal<sup>2</sup> · Debin Gao<sup>3</sup> · David Lo<sup>3</sup>

Trisha Singhal  
singhaltrisha25@gmail.com

Debin Gao  
dbgao@smu.edu.sg

David Lo  
davidlo@smu.edu.sg

<sup>1</sup> Jinan University, Guangzhou, China

<sup>2</sup> Optum, Inc, Gurugram, India

<sup>3</sup> Singapore Management University, Singapore, Singapore