ReSIL: Revivifying Function Signature Inference using Deep Learning with Domain-Specific Knowledge

Yan Lin
Singapore Management University
yanlin0816@gmail.com

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

David Lo
Singapore Management University
davidlo@smu.edu.sg

ABSTRACT

Function signature recovery is important for binary analysis and security enhancement, such as bug finding and control-flow integrity enforcement. However, binary executables typically have crucial information vital for function signature recovery stripped off during compilation. To make things worse, recent studies show that many compiler optimization strategies further complicate the recovery of function signatures with intended violations to function calling conventions.

In this paper, we first perform a systematic study to quantify the extent to which compiler optimizations (negatively) impact the accuracy of existing deep learning techniques for function signature recovery. Our experiments show that a state-of-the-art deep learning technique has its accuracy dropped from 98.7% to 87.7% when training and testing optimized binaries. We further identify specific weaknesses in existing approaches and propose an enhanced deep learning approach named ReSIL (Revivifying Function Signature Inference using Deep Learning) 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 84.8% to 92.67%. We also demonstrate security implications of ReSIL in Control-Flow Integrity enforcement in stopping potential Counterfeit Object-Oriented Programming (COOP) attacks.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Function Signature, Recurrent Neural Network, Compiler Optimization

ACM Reference Format:
Yan Lin, Debin Gao, and David Lo. 2022. ReSIL: Revivifying Function Signature Inference using Deep Learning with Domain-Specific Knowledge. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3508398.3511502

CODASPY '22, April 24–27, 2022, Baltimore, MD, USA.

ACM ISBN 978-1-4503-9220-4/22/04…$15.00
https://doi.org/10.1145/3508398.3511502

1 INTRODUCTION

Function signature recovery is a crucial step for many security applications including code hardening [19, 21, 24, 31, 34, 35], bug finding [26, 30], decompilation [2, 10], and clone detection [13]. A stripped binary, however, only contains low-level information such as instructions and register usage because compilers do not preserve much language-level information, e.g., types, in generating the binary executable. To make things worse, a recent study [20] shows that many compiler optimization strategies further complicate the recovery of function signatures with intended violations to function calling conventions.

Many existing techniques to recover function signature [21, 31] are often limited to custom and manually created rules based on calling convention of the toolchain that generated the binary. For example, TypeArmor [31] and τCFI [21] are based on the calling convention that every function argument is properly set and retrieved at callee and caller sites. However, compiler optimizations may violate such calling convention as demonstrated in the recent study [20], where, e.g., a callee or caller may skip retrieving or setting certain arguments, respectively, resulting in miscalculation of the number of arguments at callees and callers. Also note that new compiler optimization strategies are constantly being proposed and added to our mainstream compilers including gcc[1] and clang[2].

To avoid the reliance on a potentially brittle set of manually created rules and keeping track of the latest optimization strategies, recent years have witnessed an increased interest in leveraging deep learning models trained on large and freely available code repositories, e.g., EKLAVYA [6]. It uses a three layers 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. 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 (O0)). 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 negative impact on corresponding applications, e.g., in enforcement of Control-Flow Integrity (CFI).

To dig up the underlying reasons of inferior performance of deep learning approaches in recovering function signatures from optimized binaries, we not only perform training and testing of a set of deep learning models but also carefully analyze the instructions at the corresponding callees and callers where the models make mistakes in their classification. We find that most of the mistakes

[1] https://ftp.gnu.org/gnu/gcc/gcc-10.2.0/
[2] https://releases.llvm.org/download.html#10.0.1

are not due to the learning capabilities of the models, but rather the absence of evidence that function arguments are accessed or prepared. For example, two callee functions with different numbers of arguments could present exactly the same sets of instructions to the deep learning model due to compiler optimizations. Such absence of evidence in the representation of training and testing sets is the critical cause of low accuracy. The natural question then is how we could design an enhanced representation of the training and testing sets that incorporate all vital function signature evidences even when optimization is enforced, so that we can revivify the deep learning capabilities in function signature recovery.

To this end, our contribution in this paper is two-fold. First, we identify root causes of inferior performance of existing deep learning approaches in recovering function signatures from optimized binaries. Second, we propose ReSIL, with compiler-optimization-specific domain knowledge injected into the representation of training and testing sets, to improve the capability of deep learning on optimized binaries.

Difficulties in recovering function signatures for optimized binaries. 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 could be omitted, or that the same instructions could operate on different types of arguments. Having two functions with different ground-truth labels but seemingly identical function bodies would therefore confuse a supervised deep learning model in its training.

Incorporation of domain-specific knowledge in deep learning. 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 soley 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.

Application of ReSIL and security implications. We 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.

Our paper makes the following key contributions.

- We identify an extensive set of intricacies that confuse current deep learning approaches in inferring function signatures, and show that the root causes are not about learning capability but representation of binary samples.
- 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 briefly discuss related work on function signature recovery, machine learning approaches in general for analyzing binaries, and more specifically EKLAVYA since it is closely related to contributions we make in this paper.

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. EIWazeer et al. [9] apply liveness analysis to recover arguments, variables, and their types for x86 executables. TIE [18] infers variable types in binaries through formulating the usage of different data types. Caballero et al. [3] make use of dynamic liveness analysis to recover function arguments for execution traces. TypeArmor [31] and τCFI [21] 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. [33] propose to perform type inference based on debugging information generated by the compiler. Yan et al. [20] 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. [20] suffers from the requirement of having to constantly update the corresponding heuristics manually 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 [1, 29, 32] is the preliminary of many advanced binary analysis, including our proposed system in this paper, ReSIL, and EKLAVYA [6]. Laika [8] uses Bayesian unsupervised learning to detect data structures given a memory image of the program. Caliskan et al. [4] use random forest to infer programmer identity. Rosenblum et al. [25] make use of linear Support Vector Machines (SVMs) to infer the compiler family, versions, optimization options, and source languages. DEBIN [12] 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.

Pizzolotto and Inoue [23] recognize both the compiler and the presence of optimizations using a Long-Short Term Memory network and a Convolutional Neural Network. CATI [5] uses Convolutional Neural Network (CNN) to infer variable types by taking the instruction context into account. Our approach, ReSIL, belongs to this category of leveraging powerful machine learning techniques, with a focus on improving the accuracy in analyzing *optimized* binaries specifically.

2.3 EKLAVYA

EKLAVYA [6] is a representative and state-of-the-art project that applies machine learning techniques to recover function signatures, and is closely related to contributions we are making in this paper. We discuss it in slightly more detail. All instructions are represented in a 256-dimensional vectors and labels denoting the number (types) of arguments serving as input to a RNN. It 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` instruction are used in tasks 1 and 3, whereas only instructions in the callee itself are used in tasks 2 and 4.

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

3 WHY DEEP LEARNING TECHNIQUES FALL SHORT OF OPTIMIZED BINARIES

In this section, we take a deep dive into the reasons why existing deep learning techniques fall short of recovering function signatures from optimized binaries. We first present our experiments with a state-of-the-art deep learning technique EKLAVYA and its overall accuracy and F1 scores in processing unoptimized and optimized binaries (Section 3.1). Due to the difficulty in explainability of machine learning models in general, our next task is 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 3.2).

3.1 Accuracy in Inferring Function Signatures

We evaluate the accuracy of inferring the number and types of arguments for a state-of-the-art deep learning approach, named EKLAVYA [6]. 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 choose the same dataset used in EKLAVYA, which includes *binutils*, *coreutils*, *findutils*, *sg3utils*, *utillinux*, *inetutils*, *diffutils*, and *usbutils*, and compile them with the latest compiler version. We intentionally use the latest compiler versions as optimization strategies keep being improved. Note that this also means that we are not replicating the experiments in the EKLAVYA paper. We focus on function signature inference in x86-64 Linux, and the binaries are generated by using two commonly used compilers, gcc-10 and clang-10, with different optimization levels ranging from O0 to O3. We use the same sanitization method in EKLAVYA to remove duplicated functions. The resulting dataset contains 2,584 different binaries, 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 in these real-world applications. Therefore, we only report recovery results for the first 3 arguments, most of which are pointers, 32-bit integers, and 64-bit integers. The ground truth is collected by parsing the DWARF debug information [7], which has the source-level information about each argument.

Table 1: Number of arguments of functions in dataset

Opt	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

We use 5-fold cross-validation to perform training and testing. The average results are shown in Table 2 and Table 3 with the definitions of accuracy given in Definition 1.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

where TP, TN, FP, and FN are the true positive, true negative, false positive, and false negative predictions, respectively. True positive and true negative predictions are the cases correctly classified by the model.

As shown in Table 2, when optimization is enabled (O1/O2/O3), the accuracy in inferring the number of arguments drops compared to that for unoptimized callees and callers. 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 3.2.

Table 3 shows more interesting results. First, the accuracy in inferring the types for the second and third arguments are lower than that for the first argument. When counting the type distribution, we realize that more than 95% of the first arguments are

Table 2: Accuracy of EKLAVYA in inferring number of arguments.

Task	Opt	Clang			Gcc		
		CI	T	Acc%	CI	T	Acc%
Task 1	*O0*	*4,946*	*5,157*	*95.91*	*4,232*	*4,332*	*97.69*
	O1	3,903	4,276	91.28	3,248	3,593	90.40
	O2	1,463	1,568	93.30	1,706	1,926	88.58
	O3	223	234	95.30	526	568	92.61
Task 2	*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

Task 1: Inferring the number of arguments from callers.
Task 2: Inferring the number of arguments from callees.
CI: Number of callers/callees correctly inferred.
T: Number of callers/callees in the testing set.

Table 3: Accuracy of EKLAVYA in inferring types of arguments

Task	Arg	Opt	Clang			Gcc		
			CI	T	Acc%	CI	T	Acc%
Task 3	1st	*O0*	*4,236*	*4,370*	*96.93*	*3,977*	*4,086*	*97.33*
		O1	3,868	3,997	96.77	3,257	3,379	96.39
		O2	1,440	1,470	97.96	1,740	1,817	95.76
		O3	212	217	97.70	537	547	98.17
	2nd	*O0*	*2,981*	*3,260*	*91.44*	*2,567*	*2,804*	*91.55*
		O1	2,727	2,955	92.28	2,114	2,333	90.61
		O2	1,115	1,166	95.63	1,052	1,155	91.08
		O3	151	156	96.79	334	355	94.08
	3rd	*O0*	*1,729*	*1,869*	*92.51*	*1,447*	*1,590*	*91.01*
		O1	1,615	1,743	92.66	1,242	1,367	90.86
		O2	845	874	96.68	703	768	91.54
		O3	123	127	96.85	222	234	94.87
Task 4	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

Task 3: Inferring argument types from callers.
Task 4: Inferring argument types from callees.
CI: Number of callers/callees correctly inferred.
T: Number of callers/callees in the testing set.

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 3.2.

We also use F1 score (see its definition in Definition 2) to measure the performance of EKLAVYA in inferring the number of arguments; see Table 4 and Table 5 for the F1 score in inferring the number of arguments and the type of the second argument. Note that here we report the F1 score for fine-grained type inference, so that we can compare it with our approach. We only analyze the result for the second argument as it has a wider variety of different types. We also skip the analysis on callees with three or more arguments due to its small sample size. Note that MF (last column in the tables) measures the F1-score of the aggregated contributions of all classes; see its definition in Equation 4.

$$F1_i = 2 \times \frac{Pc_i \times Rc_i}{Pc_i + Rc_i} \tag{2}$$

Table 4: F1 score of EKLAVYA in inferring the number of arguments

Task	CPL	Opt	Number of Arguments									
			0	1	2	3	4	5	6	7	8	MF
Task 1	Clang	*O0*	*0.89*	*0.97*	*0.97*	*0.98*	*0.93*	*0.88*	*0.87*	*0.93*	*0.92*	*0.96*
		O1	0.78	0.93	0.92	0.93	0.89	0.86	0.85	0.85	0.88	0.91
		O2	0.83	0.94	0.93	0.95	0.95	0.88	0.88	0.86	0.92	0.93
		O3	0.82	0.94	0.94	0.97	0.95	0.97	0.99	0.89	0.80	0.95
	Gcc	*O0*	*0.92*	*0.98*	*0.98*	*0.98*	*0.97*	*0.96*	*0.97*	*0.98*	*0.94*	*0.98*
		O1	0.80	0.92	0.91	0.92	0.89	0.89	0.89	0.91	0.88	0.90
		O2	0.73	0.92	0.88	0.91	0.84	0.85	0.85	0.84	0.86	0.89
		O3	0.77	0.95	0.92	0.94	0.89	0.88	0.89	0.86	0.77	0.93
Task 2	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

Task 1: Inferring the number of arguments from callers.
Task 2: Inferring the number of arguments from callees.

Table 5: F1 scores in inferring the type of the second argument.

Task	CPL	Opt	Type of Arguments									
			int8	int16	int32	int64	float	pointer	enum	struct	union	MF
Task 3	Clang	*O0*	*0.93*	*0.80*	*0.96*	*0.82*	*0.00*	*0.94*	*0.93*	*0.83*	*0.98*	*0.95*
		O1	0.87	0.17	0.95	0.81	0.00	0.94	0.93	0.74	0.97	0.94
		O2	0.93	0.22	0.97	0.84	-	0.96	0.91	0.90	-	0.96
		O3	0.75	-	0.97	0.77	-	0.98	0.80	-	-	0.98
	Gcc	*O0*	*0.77*	*0.31*	*0.93*	*0.76*	*0.00*	*0.94*	*0.75*	*0.50*	*0.96*	*0.93*
		O1	0.62	0.10	0.89	0.78	-	0.93	0.78	0.59	-	0.92
		O2	0.66	0.00	0.91	0.80	-	0.93	0.71	0.90	-	0.92
		O3	0.86	-	0.91	0.83	-	0.95	0.80	-	-	0.95
Task 4	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

Task 3: Inferring argument types from callers.
Task 4: Inferring argument types from callees.
"-" denotes cases where the testing dataset does not have the corresponding type of argument.

where Pc_i and Rc_i are the Precision and Recall for class i, and they are defined as:

$$Pc_i = \frac{TP_i}{TP_i + FP_i} \qquad Rc_i = \frac{TP_i}{TP_i + FN_i} \tag{3}$$

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

$$\text{MF} = 2 \times \frac{\text{micro-Pc} \times \text{micro-Rc}}{\text{micro-Pc} + \text{micro-Rc}} \tag{4}$$

$$\text{micro-Pc} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} TP_i + \sum_{i=1}^{n} FP_i} \qquad \text{micro-Rc} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} TP_i + \sum_{i=1}^{n} FN_i} \tag{5}$$

where n is the number of labels in testing set.

As shown in Table 4, the F1 score for callers (task 1) with zero arguments is relatively low. It shows that existing deep learning 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 task 2 in Table 4 delivers a similar message that callees compiled with optimization has 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 5 shows that the F1-score for 64-bit integers (int64) is low but that for pointers is higher, which implies EKLAVYA does not have a good performance in distinguishing between 64-bit integers and pointers.

3.2 Four Key Scenarios that Contribute to the Lower Accuracy

The overall statistics presented in Section 3.1 show that existing deep learning approaches fall short on analyzing optimized binaries. 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. We compile our findings into the following four key scenarios.

3.2.1 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. Figure 1a and 1b show such an example in which all the arguments of function `lua_toboolean` are accessed in helper function `index2adr`. If the input to the deep learning 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 deep learning model since the sample label indicates multiple arguments whereas the function body (helper excluded) does not have the corresponding argument reading instructions.

*Arguments are not used (denoted as **Unread**).* There are also cases with unused arguments in callees, usually due to fixed prototype of the function (e.g., virtual functions). However, the label given to the deep learning engine in existing approaches always have these unused arguments counted, which would confuse the training process. As shown in Figure 2, the first and third arguments of `jpeg_free_large` are not used, but the label used in the training set indicates that it has three arguments.

*3.2.2 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 Figure 3, function ar_emul_append is a wrapper function and existing deep learning approaches only use

```
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

Figure 1: 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>
```

Figure 2: Not reading argument registers

```
1 ar_emul_append:
2 mov  0x2f8e19(%rip),%rax
3 test %rax,%rax
4 je   342bf
5 sub  $0x8,%rsp
6 call 352b0
```

Figure 3: Missing argument-preparing instructions

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 deep learning models and in turn 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 4.

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

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

(a) %esi used to pass argument **(b) %esi used to store temporary**

Figure 4: Indistinguishable argument register usage

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

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

(a) tr: append_char_class **(b) tr: xmemdup**

Figure 5: Example of Indistinguishable types

3.2.3 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 [14], all argument registers could also be used as scratch registers to store temporary values. This serves as noise to the deep learning 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 Figure 4.

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.

*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 problems also makes it hard for a deep learning 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 5.2. 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. Figure 5 shows such an example.

*3.2.4 Irrelevant instructions (denoted as **Irrelevance**).* The last contributor to lower accuracy in processing optimized binaries refer to noise in the training and testing samples. Current deep learning approaches take the entire function body as input, where many instructions are not related to the identification of the number (types) of arguments. Such noise may affect the performance of machine learning as shown by Sharma et al. [28]. As shown in Figure 1a, many instructions are not relevant to argument reading, such as instructions at Line 9 and Line 11.

Table 6: Number of intricacy cases

Opt	#*Helper*	#*Unread*	#*Wrapper*	#*Temp*
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

Table 6 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 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 larger number of functions compared to O2 and O3 binaries. In other words, the numbers reported in Table 6 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 would have irrelevant instructions.

4 RESIL: REHABILITATING DEEP LEARNING ON OPTIMIZED BINARIES

In the previous section, we detail four complication scenarios that contribute to the lower accuracy of deep learning 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 deep learning models regain its learning capability.

4.1 Missing Argument-Related Instructions

As discussed in Section 3.2, 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 deep learning 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 [20]), 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 [14]:

- `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 deep learning engine what these newly introduced opcodes mean, as the deep learning 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 Figure 1a and 1b, 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 Figure 1c[3]. 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 such potential "noise" for the deep learning engine to figure out, as our additional treatment here is not to replace the deep learning engine but to present to 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 Figure 2 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 enforcements always apply exactly the same policy due to the conservative inferencing at callees and callers [21, 31].

[3]The opcode for the instruction at Line 4 is 0x3b with 0x48 being the prefix used to indicate use of a 64-bit register.

```
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
```

Figure 6: Inserted instructions at caller sites

```
1  402650 <lua_toboolean>:
2  402650: push   %rax
3  402651: callq  4021f0
4  402656: mov    %rax,%rcx
5  402659:  mov    0x8(%rax),%eax
6  40265c:  test   %eax,%eax
7  40265e: je     402674
8  402660:  cmp    $0x1,%eax
9  402663: jne    40266f
10 402665:  xor    %eax,%eax
11 402667: cmpl   $0x0,(%rcx)
12 40266a:  setne  %al
13 40266d: pop    %rcx
14 40266e: retq
```

Figure 7: Irrelevant instruction removal

ReSIL obtained the corrected ground-truth label by examining the presence or absence of attribute `__attribute__((unused))` or by performing static binary analysis [20] to find out the number of argument registers that are actually used. Such analysis is only performed for the training samples.

*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 [20]) with our newly introduced opcodes, and then insert them into the wrapper function. Figure 6 shows the result of our insertion (Line 2 to Line 6) to the example shown in Figure 3. 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.

4.2 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 deep learning 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 Figure 5b, the top five outputs for the type of the second argument are <*pointer*, *int*64, *int*32, *float*, *int*8> with probabilities <0.876, 0.124, 3.18e-05, 2.23e-05, 2.44e-07>. We can see that it has a nonnegligible probability being a 64-bit integer.

4.3 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_toboolean` are shown in Figure 7 with light gray color, which are not included as input to the deep learning engine (for both training and testing samples) in ReSIL.

4.4 Classification Output of ReSIL

With our corrections to the deep learning 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: $\tau ::= int8|int16|int32|int64|pointer|struct|float$. 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.

5 EVALUATION

In this section, we evaluate the accuracy of ReSIL in inferring the number and types of arguments. 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 [17] pass and on DWARF v4 debugging information [7] 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.

We use the same dataset described in Section 3.1 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 6 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.

5.1 Performance Evaluation

5.1.1 Accuracy. Our goal is to evaluate the accuracy of inferences for the four tasks by using the approach we proposed in Section 4 to correct the complication scenarios we identified. Results are shown in Table 7 and Table 8, which are the average accuracy for the five folds under different compilers with different optimization levels. Note that we present 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 [11].

Table 7: Accuracy in inferring the number of arguments. *Numbers in gray backgroud show that ReSIL improves the accuracy in inferring the number of arguments even only using the top 1 output as the inference number.*

Task	Opt	App	Clang				Gcc			
			CI	T	Acc%	CS%	CI	T	Acc%	CS%
Task 1	*O0*	*R5*	*4,996*		*96.88*	*96.88*	*4,236*		*97.78*	*95.94*
		E	*4,946*	*5,157*	*95.91*	*99.00*	*4,232*	*4,332*	*97.69*	*98.99*
		R1	*4,947*		*95.93*	*99.57*	*4,202*		*97.00*	*99.69*
	O1	R5	4,046		94.62	96.62	3324		92.51	93.49
		E	3,903	4,276	91.28	98.80	3248	3,593	90.40	98.39
		R1	3,969		92.82	99.19	3221		89.65	98.69
	O2	R5	1,511		96.36	97.94	1,751		90.91	93.89
		E	1,463	1,568	93.30	99.01	1,706	1,926	88.58	98.55
		R1	1,492		95.15	99.56	1,694		87.95	98.70
	O3	R5	230		98.29	98.42	531		93.49	94.41
		E	223	234	95.30	99.28	526	568	92.61	98.72
		R1	228		97.44	99.66	517		91.02	99.09
Task 2	*O0*	*R5*	*2,006*		*99.21*	*99.55*	*2,181*		*98.73*	*98.78*
		E	*2,000*	*2,022*	*98.91*	*99.30*	*2,175*	*2,209*	*98.46*	*99.37*
		R1	*2,002*		*99.01*	*99.46*	*2,167*		*98.10*	*99.54*
	O1	R5	1,936		92.68	95.12	1,444		93.64	95.78
		E	1,772	2,089	84.83	98.86	1,349	1,542	87.48	98.90
		R1	1,894		90.67	99.13	1,415		91.76	99.21
	O2	R5	756		94.86	97.01	1,063		93.57	95.86
		E	725	797	90.97	99.28	1,008	1,136	88.73	99.08
		R1	746		93.60	99.30	1,044		90.00	99.14
	O3	R5	152		95.60	96.32	406		95.53	96.57
		E	150	159	94.34	99.70	390	425	91.76	99.27
		R1	150		94.34	99.63	399		93.88	99.08

Task 1: Inferring the number of arguments for callers.
Task 2: Inferring the number of arguments for callees.
Opt: Optimization level.
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.

Table 7 shows that ReSIL (with its top one output) generally outperforms EKLAVYA with its most significant improvement in inferring the number of arguments at callees (Task 2) when optimization is enabled, especially for callees compiled by O1. This agrees with the statistics of the complication scenarios in Table 6 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 (Task 3) is comparable with the result of EKLAVYA; see Table 8. If we only use ReSIL's top one output as the inferred label, 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 and between 64-bit integers and pointers. 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

Table 8: Accuracy in inferring the type of argument.

CPL	Arg	Opt	ReSIL			EKLAVYA			Top 1 of ReSIL			Top 1 of ReSIL coarse-grained type			T
			CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS%	CI	Acc%	CS %	
Clang	1st	*O0*	*4,254*	*97.35*	*98.43*	*4,236*	*96.93*	*99.60*	*4,224*	*96.66*	*99.69*	*4,237*	*96.96*	*99.65*	*4,370*
		O1	3,892	97.37	98.79	3,868	96.77	99.62	3,870	96.82	99.75	3,883	97.15	99.72	3,997
		O2	1,458	99.18	98.86	1,440	97.96	99.81	1,449	98.57	99.86	1,452	98.78	99.84	1,470
		O3	213	98.16	99.24	212	97.70	99.82	212	97.70	99.91	214	98.62	99.87	217
	2nd	*O0*	*3,028*	*94.45*	*95.79*	*2,981*	*92.98*	*99.19*	*2,967*	*92.55*	*99.23*	*2,989*	*93.23*	*99.16*	*3,206*
		O1	2,777	93.98	96.00	2,727	92.28	99.29	2,726	92.25	99.23	2,756	93.27	99.13	2,955
		O2	1,134	97.26	96.61	1,115	95.63	99.50	1,115	95.63	99.56	1,120	96.05	99.50	1,166
		O3	153	98.08	97.33	151	96.79	99.64	150	96.15	99.69	151	96.79	99.61	156
	3rd	*O0*	*1,753*	*93.79*	*96.34*	*1,729*	*92.51*	*99.10*	*1,723*	*92.19*	*99.32*	*1,740*	*93.10*	*99.19*	*1,869*
		O1	1,630	93.52	97.12	1,615	92.66	99.24	1,608	92.25	99.53	1,628	93.40	99.44	1,743
		O2	849	97.14	97.69	845	96.68	99.57	840	96.11	99.54	846	96.80	99.44	874
		O3	125	98.43	99.59	123	96.85	99.43	125	98.43	99.75	125	98.43	99.75	127
Gcc	1st	*O0*	*3,980*	*97.41*	*98.49*	*3,977*	*97.33*	*99.63*	*3,952*	*96.72*	*99.67*	*3,975*	*97.28*	*99.62*	*4,086*
		O1	3,279	97.04	97.87	3,257	96.39	99.49	3,248	96.12	99.56	3,276	96.95	99.47	3,379
		O2	1,751	96.37	97.74	1,740	95.76	99.52	1,733	95.38	99.56	1,746	96.09	99.50	1,817
		O3	534	97.62	98.49	537	98.17	99.53	531	97.07	99.62	534	97.62	99.55	547
	2nd	*O0*	*2,611*	*93.12*	*94.42*	*2,567*	*91.55*	*99.06*	*2,540*	*90.58*	*99.01*	*2,588*	*92.30*	*98.86*	*2,804*
		O1	2,151	92.20	93.43	2,114	90.61	98.91	2,082	89.24	98.88	2,137	91.60	98.60	2,333
		O2	1,077	93.25	93.74	1,052	91.08	99.11	1,029	89.09	98.80	1,056	91.43	98.58	1,155
		O3	333	93.80	95.86	334	94.08	99.36	328	92.39	98.98	332	93.52	98.81	355
	3rd	*O0*	*1,443*	*90.75*	*94.39*	*1,448*	*91.07*	*98.93*	*1,402*	*88.18*	*99.10*	*1,453*	*91.38*	*98.85*	*1,590*
		O1	1,246	91.15	95.22	1,242	90.86	98.83	1,220	89.25	99.04	1,262	92.32	98.80	1,367
		O2	705	91.80	94.97	703	91.54	99.11	688	89.58	99.06	711	92.58	98.88	768
		O3	219	93.59	96.65	222	94.87	99.40	216	92.31	99.57	219	93.59	99.46	234

(a) From instructions of callers

CPL	Arg	Opt	ReSIL			EKLAVYA			Top 1 of ReSIL			Top 1 of ReSIL coarse-grained type			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
		O3	148	98.67	99.54	147	98.00	99.84	148	98.67	99.87	148	98.67	99.87	150
	2nd	*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	1109	88.58	98.90	1,252
		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
	3rd	*O0*	*514*	*88.77*	*92.40*	*489*	*84.46*	*98.87*	*494*	*85.32*	*98.65*	*495*	*85.49*	*98.60*	*579*
		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
Gcc	1st	*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
		O3	381	97.69	98.52	381	97.69	99.74	379	97.18	99.66	380	97.44	99.61	390
	2nd	*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
		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
	3rd	*O0*	*552*	*87.20*	*92.93*	*528*	*83.41*	*98.92*	*532*	*84.04*	*98.62*	*536*	*84.68*	*98.58*	*633*
		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
		O3	146	93.59	95.61	143	91.67	99.30	143	91.67	99.04	144	92.31	99.05	156

(b) From instructions of callees

CI: Number of callers/callees correctly inferred. T: Total number of callers/callees. CS: Confidence score.

EKLAVYA as possible, and present the results in the last three columns of Table 8. 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 also helps argument type inferencing as they give evidence to the deep learning model on the number of bits of the argument that are being accessed.

Regarding recovery of argument types from the callee site (Task 4), 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 3.1, most of the first arguments are 32-bit integers and pointers which are immune to the complication scenarios we identified in Section 3.2.3; therefore, our accuracy in identifying them is much higher.

5.1.2 F1 scores. We also use F1 score to measure the performance of ReSIL for each class; see Table 9 and Table 10 for the F1 scores improvement in different tasks.

Table 9 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,

Table 9: Improvement of F1 scores ($F1_{ReSIL} - F1_{EKLAVYA}$)% in inferring the number of arguments

Task	CPL	Opt	Number of Arguments									
			0	1	2	3	4	5	6	7	8	MF
Task 1	Clang	*O0*	*3.12*	*0.87*	*0.73*	*0.31*	*0.35*	*4.12*	*3.17*	*1.33*	*1.19*	*0.96*
		O1	9.20	2.45	2.93	2.21	3.74	5.76	5.44	3.31	5.80	3.23
		O2	8.30	2.24	2.56	2.43	1.32	4.60	4.02	7.89	5.18	2.75
		O3	9.79	3.87	2.43	1.85	3.01	0.69	-0.10	8.42	20.00	2.72
	Gcc	*O0*	*4.89*	*-0.14*	*-0.34*	*-0.35*	*-0.41*	*1.16*	*0.39*	*-1.13*	*2.39*	*0.10*
		O1	7.99	2.19	1.59	1.17	1.70	2.14	2.60	3.90	2.47	2.11
		O2	9.12	1.78	1.58	0.90	3.72	5.15	0.66	1.05	3.78	2.23
		O3	0.76	0.36	-0.31	1.12	4.29	6.86	-1.28	-1.76	-2.14	0.88
Task 2	Clang	*O0*	*1.16*	*0.26*	*0.16*	*0.41*	*0.06*	*-0.04*	*51.67*	*0.00*	*0.00*	*0.27*
		O1	14.97	7.07	6.48	6.78	10.12	7.87	7.36	19.90	20.04	7.86
		O2	7.14	4.47	3.41	4.08	1.40	-0.68	5.72	-0.28	3.71	3.81
		O3	3.34	1.02	-0.17	0.87	2.87	1.35	9.53	19.44	-11.11	1.36
	Gcc	*O0*	*0.64*	*0.41*	*0.39*	*0.04*	*-0.36*	*0.50*	*22.22*	*-50.00*	*0.00*	*0.28*
		O1	15.60	6.26	5.05	5.31	4.91	3.25	4.50	10.76	6.06	6.12
		O2	9.99	4.72	4.32	4.33	5.05	5.28	0.94	3.52	10.71	4.89
		O3	7.14	3.81	3.04	2.39	5.37	3.54	4.50	18.48	14.00	3.72

Task 1: Inferring the number of arguments from callers.
Task 2: Inferring the number of arguments from callees.

Table 10: Improvement of F1 scores ($F1_{ReSIL} - F1_{EKLAVYA}$)% in inferring the type of the second argument

Task	CPL	Opt	Type of Arguments							
			int8	int16	int32	int64	float	pointer	struct	MF
Task 3	Clang	*O0*	*1.44*	*3.81*	*1.56*	*2.63*	*0.00*	*0.91*	*17.44*	*-0.34*
		O1	2.22	-16.67	1.79	3.96	0.00	1.34	22.29	-0.10
		O2	-1.76	0.00	1.41	6.12	-	1.45	9.52	0.56
		O3	25.00	-	0.49	12.14	-	0.67	-	0.38
	Gcc	*O0*	*4.73*	*8.67*	*2.72*	*5.61*	*0.00*	*1.51*	*29.68*	*0.20*
		O1	8.28	30.48	3.61	4.37	-	1.80	40.67	0.09
		O2	5.62	0.00	1.14	3.59	-	1.36	3.33	-0.29
		O3	-5.19	-	1.49	3.33	-	0.76	-	-1.14
Task 4	Clang	*O0*	*4.36*	*16.67*	*2.43*	*7.70*	*56.67*	*1.25*	*10.16*	*0.42*
		O1	15.43	-8.33	4.79	5.90	33.33	1.52	3.33	0.69
		O2	7.90	100.00	3.52	5.13	0.00	1.12	-88.89	0.21
		O3	-25.00	-	4.37	9.40	-	1.84	-	1.78
	Gcc	*O0*	*0.42*	*-10.00*	*2.68*	*6.43*	*51.11*	*1.37*	*-69.05*	*0.14*
		O1	16.04	0.00	5.41	4.22	-50.00	1.37	-11.11	0.89
		O2	6.18	16.67	2.94	6.50	-16.67	1.23	33.33	0.41
		O3	3.14	-	0.72	-1.76	0.00	-0.18	-	-0.82

Task 3: Inferring argument types from callers.
Task 4: Inferring argument types from callees.
"-" denotes cases where the testing dataset does not have the corresponding type of arguments.

we can see that ReSIL has a higher F1 score for callees which have zero arguments.

Table 10 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 12.14% and 9.40%, respectively.

5.1.3 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 value 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 deep learning 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 11.

Table 11: Number of complication scenarios incorrectly recovered

	Opt	Fold															
		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	
Unread	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	
Helper	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	
Temp	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	
Wrapper	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 five output. R1: ReSIL with top one output.

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 limited improvement in inferring the number of arguments for other callees since the cases of ***Helper*** and ***Unread*** will confuse the deep learning 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.

5.2 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) [27]. 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.

5.2.1 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 COOP paper [27] 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 [27] to show how ReSIL could stop such an exploit.

The details about the gadgets can be found in Figure 8. Function `nsMultiplexInputStream::Close` is used as the ML-G gadget, while when we 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 are 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 stop the Firefox COOP exploit.

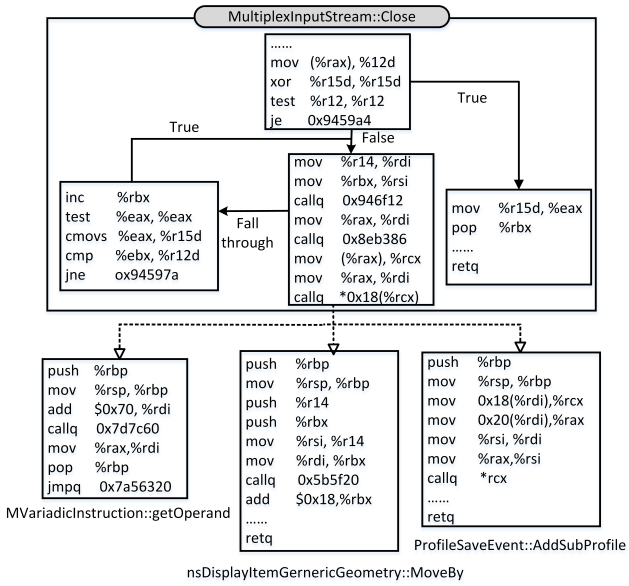

Figure 8: Gadgets used in COOP's 64-bit Firefox exploit

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 deep learning model to correctly recover the number of arguments for it.

5.2.2 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 [3]. In addition, as shown by Jain et al. [15], the inferred function signature can be used to improve fuzzing and type-based fuzzing can 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 [16, 22].

5.3 Limitations of ReSIL

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

- **Callees (Callers) with more than six arguments.** We find it more difficult to 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 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 on the first argument `%rdi` and cause 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 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 can infer 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 Figure 9. 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) output that this callee has zero arguments.

```
000000000041e810 <stringer>:
#18 instructions
41e84e: je     41e8b8 <stringer+0xa8>
#7  instructions
41e86b: call   402070
#21 instructions
41e8ce: test   %dil,0x1
```

Figure 9: Number of arguments inferred correctly by EKLAVYA but wrongly by ReSIL

6 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.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful and helpful comments. This presearch 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.

REFERENCES

[1] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. *BYTEWEIGHT*: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium*. 845–860.

[2] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium*. 353–368.

[3] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. 2009. *Binary code extraction and interface identification for security applications*. Technical Report. California Univ Berkeley Dept of Electrical Engineering and Computer Science.

[4] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proceedings of the 22nd Network and Distributed System Security Symposium*.

[5] Ligeng Chen, Zhongling He, and Bing Mao. 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, 88–98.

[6] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium*. 99–116.

[7] DWARF Debugging Information Format Committee et al. 2010. DWARF debugging information format, version 4. *Free Standards Group* (2010).

[8] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. 2008. Digging for Data Structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, Vol. 8. 255–266.

[9] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 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, 51–60.

[10] Ilfak Guilfanov. 2008. Decompilers and beyond. *Black Hat USA* (2008), 1–12.

[11] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International Conference on Machine Learning*. PMLR, 1321–1330.

[12] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*. ACM, 1667–1680.

[13] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. IEEE, 63–72.

[14] INC INTEL. 2018. Intel® 64 and ia-32 architectures software developer's manual. (2018).

[15] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 505–517.

[16] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *arXiv preprint arXiv:2011.10749* (2020).

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd international symposium on Code generation and optimization*. IEEE.

[18] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Network and Distributed System Security Symposium*.

[19] Yan Lin, Xiaoyang Cheng, and Debin Gao. 2019. Control-flow carrying code. In *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*. ACM, 3–14.

[20] Yan Lin and Debin Gao. 2021. When Function Signature Recovery Meets Compiler Optimization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. IEEE.

[21] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 423–444.

[22] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. 2017. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 341–355.

[23] Davide Pizzolotto and Katsuro Inoue. 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, 232–242.

[24] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.. In *Proceedings of the 22nd Network and Distributed System Security Symposium*.

[25] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*. ACM, 100–110.

[26] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, 225–236.

[27] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 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, 745–762.

[28] Abhishek Sharma, Yuan Tian, and David Lo. 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, 449–458.

[29] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium*. 611–626.

[30] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[31] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 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, 934–953.

[32] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-aware machine learning for function recognition in binary code. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*. IEEE, 388–398.

[33] Dongrui Zeng and Gang Tan. 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, 366–376.

[34] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[35] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*. 337–352.