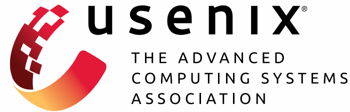

Neural Nets Can Learn Function Type Signatures From Binaries

Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang,
National University of Singapore

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

Neural Nets Can Learn Function Type Signatures From Binaries

Zheng Leong Chua* Shiqi Shen* Prateek Saxena Zhenkai Liang
National University of Singapore
{*chuazl, shiqi04, prateeks, liangzk*}@*comp.nus.edu.sg*

Abstract

Function type signatures are important for binary analysis, but they are not available in COTS binaries. In this paper, we present a new system called EKLAVYA which trains a recurrent neural network to recover function type signatures from disassembled binary code. EKLAVYA assumes no knowledge of the target instruction set semantics to make such inference. More importantly, EKLAVYA results are "explicable": we find by analyzing its model that it auto-learns relationships between instructions, compiler conventions, stack frame setup instructions, use-before-write patterns, and operations relevant to identifying types directly from binaries. In our evaluation on Linux binaries compiled with `clang` and `gcc`, for two different architectures (x86 and x64), EKLAVYA exhibits accuracy of around 84% and 81% for function argument count and type recovery tasks respectively. EKLAVYA generalizes well across the compilers tested on two different instruction sets with various optimization levels, without any specialized prior knowledge of the instruction set, compiler or optimization level.

1 Introduction

Binary analysis of executable code is a classical problem in computer security. Source code is often unavailable for COTS binaries. As the compiler does not preserve a lot of language-level information, such as types, in the process of compilation, reverse engineering is needed to recover the semantic information about the original source code from binaries. Recovering semantics of machine code is important for applications such as code hardening [54, 34, 53, 26, 52], bug-finding [39, 47, 10], clone detection [18, 38], patching/repair [17, 16, 41] and analysis [12, 22, 21]. Binary analysis tasks can vary from reliable disassembly of instructions to recovery of control-flow, data structures or full functional semantics. The higher the level of semantics desired, the more specialized the analysis, requiring more expert knowledge.

Commercial binary analysis tools widely used in the industry rely on domain-specific knowledge of compiler conventions and specialized analysis techniques for binary analysis. 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. Analysis engines need to be continuously updated as compilers evolve or newer architectures are targeted. In this work, we investigate an alternative line of research, which asks whether we can *train* machines to learn features from binary code directly, without specifying compiler idioms and instruction semantics explicitly. Specifically, we investigate the problem of *recovering function types / signatures* from binary code — a problem with wide applications to control-flow hardening [54, 34, 53] and data-dependency analysis [31, 40] on binaries — using techniques from *deep learning*.

The problem of function type recovery has two subproblems: recovering the *number* of arguments a function takes / produces and their *types*. In this work, we are interested in recovering argument counts and `C`-style primitive data types .[1] Our starting point is a list of functions (bodies), disassembled from machine code, which can be obtained using standard commercial tools or using machine learning techniques [7, 43]. Our goal is to perform type recovery without explicitly encoding any semantics specific to the instruction set being analyzed or the conventions of the compiler used to produce the binary. We restrict our study to Linux x86 and x64 applications in this work, though the techniques presented extend naturally to other OS platforms.

Approach. We use a recurrent neural network (RNN) architecture to learn function types from disassembled

*Lead authors are alphabetically ordered.

[1] `int`, `float`, `char`, `pointers`, `enum`, `union`, `struct`

binary code of functions. The goal is to ascertain if neural networks can effectively learn such types without prior knowledge of the compiler or the instruction set (beyond that implied by disassembly). Admittedly, the process of designing such a system has been experimental or ad-hoc (in our experience), fraught with trial-and-error, requiring sifting through the choice of architectures and their parameters. For instance, we considered designs wherein disassembled code was directly fed as text input, as one-hot encoded inputs, and with various training epoch sizes and network depth. In several cases, the results were unimpressive. In others, while the results were positive, we had little insight into what the model learnt from inputs.

Our guiding principle in selecting a final architecture is its *explicability*: to find evidence whether the learning network could learn something "explainable" or "comparable" to conventions we know which experts and other analysis tools use. To gather evidence on the correctness of a learning network's outputs, we employ techniques to measure its explicability using analogical reasoning, dimensionality reduction (t-SNE visualization plots), and saliency maps. Using these techniques, we select network architectures that exhibit consistent evidence of learning meaningful artifacts. Our resulting system called EKLAVYA automatically learns several patterns arising in binary analysis in general, and function type recovery specifically. At the same time, its constructional design is modular, such that its instruction set specific dependencies are separated from its type recovery tasks. EKLAVYA is the first neural network based systems that targets function signature recovery tasks, and our methodology for explaining its learnt outcomes is more generally useful for debugging and designing such systems for binary analysis tasks.

Results. We have tested EKLAVYA on a testing set consisting of a large number of Linux x86 and x64 binaries, compiled at various optimization levels. EKLAVYA demonstrates several promising results. First, EKLAVYA achieves high accuracy of around 84% for count recovery and has accuracy around 81% for type recovery. Second, EKLAVYA generalizes in a compiler-agnostic manner over code generated from `clang` and `gcc`, and works for the x86 and x64 binaries, with a modest reduction of accuracy with increase in optimization levels. In comparison to previous methods which use knowledge of instruction sets and compiler conventions in their analysis, EKLAVYA has comparable accuracy. Third, EKLAVYA's learnt model is largely "explicable". We show through several analytical techniques which input features the model emphasizes in its decisions. These features match many patterns that are familiar to human analysts and used in existing tools as rules, such as identifying calling conventions, caller- and callee- save registers, stack-based arguments, "use- before-write" instructions, function stack allocation idioms, and many more. All these are derived automatically without any explicit knowledge of the instruction semantics or compiler used.

EKLAVYA's architecture bears resemblance to other neural network architectures that have been successful in natural language processing (NLP) problems such as machine translation, automatic summarization, and sentence-generation. Specifically, we find the use of word-embedding of instructions has been particularly useful in our problem, which is used in NLP problems too. We hypothesize a deeper similarity between (problems arising in) natural language and the language of machine instructions, and consider it worthy of future work.

Contributions. We present EKLAVYA, a novel RNN-based engine that recovers functions types from x86/x64 machine code of a given function. We find in our experimental evaluation that EKLAVYA is *compiler-agnostic* and the same architecture can be used to train for different instruction sets (x86 and x64) without any specification of its semantics. On our x86 and x64 datasets, EKLAVYA exhibits *comparable accuracy* with traditional heuristics-based methods. Finally, we demonstrate that EKLAVYA's learning methods are *explicable*. Our analysis exhibits consistent evidence of identifying instruction patterns that are relevant to the task of analyzing function argument counts and types, lending confidence that it does not overfit to its training datasets or learn unexplained decision criteria. To our knowledge, ours is the first use of techniques such as t-SNE plots, saliency maps, and analogical reasoning to explain neural network models for binary analysis tasks.

2 Problem Overview

Function type recovery involves identifying the number and primitive types of the arguments of a function from its binary code. This is often a sub-step in constructing control-flow graphs and inter-procedural data dependency analysis, which is widely used in binary analysis and hardening tools.

Traditional solutions for function type recovery use such conventions as heuristics for function type recovery, which encode the semantics of all instructions, ABI conventions, compiler idioms, and so on. These are specified apriori in the analysis procedure by human analysts. Consider the example of a function in x64 binary code shown in Figure 1. The example illustrates several conventions that the compiler used to generate the code, such as:

```
000000000040051b <main>             00000000004004ed <fun>
push   %rbp                         push   %rbp                 ──▶ (a)
mov    %rsp,%rbp                    mov    %rsp, %rbp
sub    0x20,%rsp ──▶ (b,c)          mov    %edi,-0x24(%rbp) ┐
movl   $0x7d,-0x14(%rbp)            mov    %rsi,-0x30(%rbp) ┘ (d)
…                                   …
mov    -0x14(%rbp),%eax             mov    -0x24(%rbp),%edx
mov    %rdx,%rsi ┐                  mov    %edx,%eax  ┐
mov    %eax,%edi ┘ (d)              add    %eax,%eax  │ (e)
call   4004ed <fun>                 add    %edx,%eax  ┘
…                                   …
                                    pop    %rbp                 ──▶ (a)
                                    retq
```

Figure 1: Example assembly code with several idioms and conventions. `(a)` refers to the `push`/`pop` instructions for register save-restore; `(b)` refers to the instruction using `rsp` as a special stack pointer register; `(c)` refers to arithmetic instructions to allocate stack space; `(d)` refers to instructions passing the arguments using specific registers; `(e)` refers to the subsequent use of integer-typed data in arithmetic operations.

(a) the use of `push/pop` instructions for register save-restore;
(b) the knowledge of `rsp` as a special stack pointer register which allocates space for the local frame before accessing arguments;
(c) the use of arithmetic instructions to allocate stack space;
(d) the calling convention (use of specific register, stacks offset for argument passing); and
(e) subsequent use of integer-typed data in arithmetic operations only.

Such conventions or rules are often needed for traditional analysis to be able to locate arguments. Looking one step deeper, the semantics of instructions have to be specified in such analysis explicitly. For instance, recognizing that a particular byte represents a `push` instruction and that it can operate on any register argument. As compilers evolve, or existing analyses are retargeted to binaries from newer instructions sets, analysis tools need to be constantly updated with new rules or target backends. An ideal solution will minimize the use of specialized knowledge or rules in solving the problem. For instance, we desire a mechanism that could be trained to work on any instruction set, and handle a large variety of standard compilers and optimization supported therein.

In this work, we address the problem of function type recovery using a stacked neural network architecture. We aim to develop a system that automatically learns the rules to identify function types directly from binary code, with minimal supervision. Meanwhile, we design techniques to ensure that the learnt model produces explicable results that match our domain knowledge.

Problem Definition. We assume to have the following knowledge of a binary: (a) the boundaries of a function, (b) the boundary of instructions in a function, and (c) the instruction representing a function dispatch (e.g. direct `calls`). All of these steps are readily available from disassemblers, and step (a) has been shown to be learnable directly from binaries using a neural network architecture similar to ours [43]. Step (b) on architectures with fixed-length instructions (e.g. ARM) requires knowing only the instruction length. For variable-length architectures (e.g. x64/x86), it requires the knowledge of instruction encoding sufficient to recover instruction sizes (but nothing about their semantics). Step (c) is a minimalistic but simplifying assumption we have made; in concept, identifying which byte-sequences represent `call` instruction may be automatically learnable as well.

The input to our final model $\mathscr{M}$ is a *target* function for which we are recovering the type signature, and set of functions that call into it. Functions are represented in disassembled form, such that each function is a sequence of instructions, and each instruction is a sequence of bytes. The bytes do not carry any semantic meaning with them explicitly. We define this clearly before giving our precise problem definition.

Let T_a and $T_a[i]$ respectively denote the disassembled code and the ith bytes of a target function a. Then, the kth instruction of function a can be defined as:

$$I_a[k] := < T_a[m], T_a[m+1], ..., T_a[m+l] >$$

where m is the index to the start byte of instruction $I_a[k]$ and l is the number of bytes in $I_a[k]$. The disassembled form of function a consisting of p instructions is defined as:

$$T_a := < I_a[1], I_a[2], ..., I_a[p] >$$

With the knowledge of a `call` instruction, we determine the set of functions that call the target function a. If a function b has a direct call to function a, we take all[2] the instructions in b preceding the `call` instruction. We call this a *caller snippet* $C_{b,a}[j]$, defined as:

$$C_{b,a}[j] := < I_b[0], I_b[1] ... I_b[j-1] >$$

where $I_b[j]$ is a direct `call to` a. If $I_b[j]$ is not a direct `call to` a, $C_{b,a}[j] := \emptyset$. We collect all caller snippets calling a, and thus the input $\mathscr{D}_a$ is defined as:

$$\mathscr{D}_a := T_a \cup \Big(\bigcup_{b \in S_a} \Big(\bigcup_{0 \le j \le |T_b|} C_{a,b}[j] \Big) \Big)$$

where S_a is the set of functions that call a.

[2] In our implementation, we limit the number of instructions to 500 for large functions.

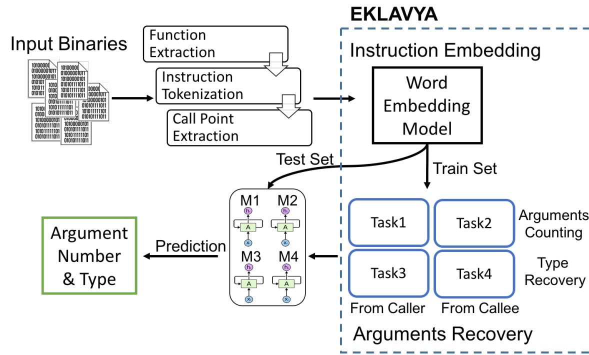

Figure 2: EKLAVYA Architecture. It takes in the binaries as input and performs a pre-processing step on it. Then it performs instruction embedding to produce embedded vectors for train and test dataset. The argument recovery module trains 4 RNN models $M_1, M_2, M_3, M_4$to recover the function argument count and types.

With the above definitions, we are now ready to state our problem definition. Our goal is to learn a model $\mathscr{M}$, which is used to decide two properties for a target function a, from given data $\mathscr{D}_a$, stated below:

Definition. (**Arguments Counts**) The number of arguments passed to function a.

Definition. (**Argument Types**) For each argument of function a, the C-style types defined as:

$$\tau ::= \texttt{int} | \texttt{char} | \texttt{float} | \texttt{void*} | \texttt{enum} | \texttt{union} | \texttt{struct}$$

Note that the above definition gives the inputs and outputs of the model $\mathscr{M}$, which can be queried for a target function. This is called the *test* set. For training the model $\mathscr{M}$, the training set has a similar representation. It consists of the disassembled functions input $\mathscr{D}_a$ as well as labels (the desired outputs) that represent the ground truth, namely the true number and types of each argument. For the training set, we extract the ground truth from the debug symbols generated from source code.

3 Design

EKLAVYA employs neural network to recover argument counts and types from binaries. The overall architecture is shown in Figure 2. EKLAVYA has two primary modules: a) instruction embedding module and an b) argument recovery module. The instruction embedding module learns the *semantics* of instructions by observing their use in our dataset of binaries (from one instruction set). It is possible to have one neural network that does not treat these as two separate substeps. However, in this case, the instruction semantics learnt may well be very specialized to the task of argument recovery. In our design, we train to extract semantics of the instruction set from binaries separately, independent to the task of further analysis at hand. This makes the design modular and allows reusing the embedding module in multiple binary analysis tasks. In addition, instead of keeping the semantics as an implicit internal state, explicitly outputting the semantics allows us to verify the correctness of each step independently. This makes the process of designing and debugging the architecture easier, thus motivating our choice of two modules.

The instruction embedding module takes as input a stream of instructions, represented as symbols. It outputs a vector representation of each instruction in a 256-dimensional space, hence *embedding* the instructions in a vector space. The objective is to map symbol into vectors, such that distances between vectors capture inter-instruction relationships.

Given the instructions represented as vectors, EKLAVYA trains a recurrent neural network (RNN) over the sequence of vectors corresponding to the function body. This is done in the argument recovery module. In some cases, EKLAVYA may only have the target of the function body to analyze, and in others it may have access to a set of callers to the target function. For generality, EKLAVYA trains models for four *tasks* defined below:

(a) **Task1**: Counting arguments for each function based on instructions from the caller;
(b) **Task2**: Counting arguments for each function based on instructions from the callee;
(c) **Task3**: Recovering the type of arguments based on instructions from the caller;
(d) **Task4**: Recovering the type of arguments based on instructions from the callee;

We train one model for each task, over the same outputs of the instruction embedding module. For each instruction set, we learn a different instruction embedding and RNN set. For a function to be tested, the user can use the predictions of any or all of these tasks; our default is to report the output of Task2 for argument counts and Task4 for types since this is analyzable from just the callee's function body (without knowing callers).

3.1 Instruction Embedding Module

The first key step in EKLAVYA is to uncover the semantic information of each instruction through learning. Note that the inputs to our learning algorithm are functions represented as raw binaries, with known boundaries of functions and instructions. In this representation, the

learning algorithm does not have access to any high-level semantics of an instruction. Intuitively, the goal is to infer the semantics of instructions from their *contextual use* in the binary, such as by analyzing which group appears sequentially together or in certain contexts relative to other groups. One general approach to extracting contextual relationships is to employ a technique called *word embedding* [8]. Word embedding in EKLAVYA converts each instruction's raw symbol into a vector. All instructions are thus represented in a high-dimensional space (256 dimensions in our case). Intuitively, the distance between instructions encodes relationships. For instance, the relative distance between the vectors for `push %edi` and `pop %edi` is similar to distance between `push %esi` and `pop %esi`. We demonstrate the kinds of relationships this module learns in Section 5 through examples. In summary, the output of this module is a map from instructions to a 256-dimensional vector.

There are other alternatives to word embedding, which we have considered. One can employ one-hot encoding analogous to a previous work on identifying function boundaries [43]. One could represent the ith instruction by a vector with its ith element as 1 and all other elements set to 0. For example, if there are 5 different instructions, the second instruction is represented as $[0,1,0,0,0]$. However, this technique is computationally inefficient if we expect to learn instruction semantics usable for many different binary analysis tasks, since a separate sub-network will likely be needed to re-learn the relationship between one-hot-encoded vectors for each new analysis task.

For word embedding, we use the skip-gram negative sampling method outlined in the paper that introduces `word2vec` technique for computing word embeddings [27]. The skip-gram is a shallow neural network using the current instruction to predict the instructions around it. Compared to other approaches like continuous bag-of-words (CBOW) technique [27], skip-gram shows better performance on the large-scale dataset and extracts more semantics for each instruction in our experience. To train the word embedding model, we tokenize the hexadecimal value of each instruction and use them as the training input to the embedding model. For example, the symbol or token for the instruction `push %ebp` is its hexadecimal opcode `0x55`. Note that the hexadecimal opcode is used just as a name much like 'john' or 'apple' and bears no numerical effects on the embedding. We train the embedding model for 100 epochs with the learning rate of 0.001.

3.2 Arguments Recovery Module

The function arguments recovery module trains four neural networks, one for each task related to count and type inference. To achieve each task outlined, we train a recurrent neural network (RNN). The input for training the model is the sequence of vectors (each representing an instruction) produced by word embedding, together with labels denoting the number of arguments and types (the ground truth). For argument type recovery, we have several design choices. We could learn one RNN for the first argument, one RNN for the second argument, and so on. Alternatively, we can have one RNN that predicts the type tuple for all the arguments of a function. Presently, we have implemented the first choice, since it alleviates any dependency on counting the number of arguments.

Recurrent Neural Networks. To design the arguments recovery module, we have considered various architectures, like a multilayer perceptron (MLP), a convolutional neural network (CNN) and a recurrent neural network (RNN). We find that an RNN is a suitable choice because it handles variable-length inputs gracefully, and has a notion of "memory". A key difference between feedforward neural networks like a multi-layer perceptron (MLP) and a recurrent neural network (RNN) is that an RNN incorporates the state of the previous input as an additional input to the current time. Effectively, this input represents an internal state that can accumulate the effects of the past inputs, forming a memory for the network. The recurrent structure of the network allows it to handle variable-length input sequences naturally.

In order to deal with the exploding and vanishing gradients during training [9], there are few commonly design options. One could use an LSTM network or use an RNN model with gated recurrent units (GRUs). We use GRUs since it has the control of whether to save or discard previous information and may train faster due to the fewer parameters. We find that an RNN with 3 layers using GRUs is sufficient for our problem.

To avoid overfitting, we use the dropout mechanism, which de-activates the output of a set of randomly chosen RNN cells [48]. This mechanism acts as a stochastic regularization technique. In our design, we experimented with various dropout rates between 0.1 to 0.8. We experimentally find the dropout rate of 0.8, corresponding to randomly dropping 20% of the cell's output, leads to a good result. Our models appeared to overfit with higher dropout rates.

3.3 Data Preprocessing & Implementation

We briefly discuss the remaining details related to preparation of the inputs to EKLAVYA, and its implementation.

The input for EKLAVYA is the disassembly binary code of the target function. To obtain this data, the first step is to identify the function boundaries. Function boundaries identification with minimal reliance of

instruction set semantics is an independent problem of interest. Previous approaches range from traditional machine learning techniques [7] to neural networks [43] to applying function interface verification [35]. In this work, we assume the availability and the correctness of function boundaries for recovering function arguments. To implement this step, we downloaded the dataset Linux packages and compiled them with both `clang` and `gcc` with debugging symbols. The function boundaries, argument counts and types are obtained by parsing the DWARF entries from the binary. Our implementation uses the `pyelftools` which parses the DWARF information [2]; additionally, to extract the argument counts and types, we implemented a Python module with 179 lines of code. We extract the start and end of function boundaries using the standard Linux `objdump` utility [1]. According to Dennis et al. [6], modern disassemblers are highly accurate at performing instruction level recovery for non-obfuscated binaries, especially for binaries generated by `gcc` and `clang`. Thus we use this as the ground truth, ignoring the marginal noise that errors may create in the dataset. After disassembly, we identify call sites and the caller snippets. Our total additional code implementation to perform these steps consists of 1273 lines of Python code.

To train the instruction embedding model and RNNs, we use Google Tensorflow [4]. Our implementation for the instruction embedding and RNN learning is a total of 714 lines of Python code.

4 Explicability of Models

Our guiding principle is to create models that exhibit learning of reasonable decision criteria. To explain what the models learn, we use a different set of techniques for the two parts of EKLAVYA: the instruction embedding model and the learnt RNNs.

4.1 Instruction Embedding

Recall the instruction embedding module learns a mapping between instructions of an architecture to a high-dimensional vector space. Visualizing such large dimensionality vector space is a difficult challenge. To understand these vectors, two common techniques are used — t-SNE [25] plots and analogical reasoning of vectors.

t-SNE Plots. t-SNE is a way to project high-dimensional vectors into a lower dimension one while preserving any neighborhood structures that might exist in the original vector space. Once projected, these can be visualized with scatter plots. Methods such as principal component analysis (PCA) [19] and classical multidimensional scaling [50] use linear transformations to project onto the low dimension space. Though powerful, these techniques often miss important non-linear structure in the data. The primary advantage of t-SNE is that it captures non-linear relationships in the local and global structure of the dataset.[3] For example, if word embedding learns that two instructions are similar, then they will be nearby in the high-dimensional space. t-SNE is expected to preserve this structure in low-dimensional plots, which we can visually analyze to check if it matches our knowledge of instruction semantics and their similarity. Note that t-SNE does not necessarily exhibit all the neighborhood structures that may exist in high- dimensional space, but is a best-effort tool at visualizing relationships.

Analogical Reasoning. Another way to infer relationships between instructions represented as vectors is by *analogical reasoning*. To understand the idea intuitively, we point to how this technique is used in natural language processing tasks. In natural language, analogy question tests the ability to define relationships between words and the understanding of the vocabulary. An analogical question typically consist of two pairs of word, e.g., (“man”, “king”) (“woman”, “queen”). To answer how related the two pairs are, the analogy “man is to king as woman is to queen” is formed of which the validity is tested. The vector offset method proposed by Mikolov et al. [29] frames this using vector arithmetic. The analogical question can be represented as $I_1 - I_2 \approx I_3 - I_4$ where I_1, I_2, I_3 and I_4 are the embedding vectors. Specifically, given the analogical question (“man”, “king”), (“woman”, ?), we can formulate it as $I_3 - I_1 + I_2 \approx I_4$. To get the approximated result, we first compute $d = I_3 - I_1 + I_2$. I_4 is the vector that has the greatest cosine similarity with d. Applying the idea to our problem setting, we can find similar analogies between one pairs of instructions and others. If such analogies match our prior knowledge of certain conventions or idioms that we expect in binary code, we can confirm that EKLAVYA is able to infer these similarities in its instruction embedding representation.

4.2 RNNs for Argument Recovery

We wish to determine for a given test function to an RNN, which instructions the RNN considers as important towards the prediction. If these instruction intuitively correspond to our domain knowledge of instructions that access arguments, then it increases our confidence in the RNN learning the desired decision criteria.

[3] A short primer on its design is presented in the Appendix B for the interested reader.

One way to analyze such properties is to employ saliency maps.

Saliency Map. Saliency maps for trained networks provide a visualization of which parts of an input the network considers important in a prediction. 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. to obtain the gradient by back-propagation [44]. Specifically, we calculate the derivative of the output of the penultimate layer with respect to each input instruction (which is a vector). This results in a Jacobian matrix. Intuitively, each element in a Jacobian matrix tells us how each dimension of the instruction vector will affect the output of a specific class (a single dimension of the output). In this case, we just want to know how much effect a particular dimension has over the entire output, so we sum 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 have over the input. In order for us to visualize our saliency map, we need a scalar representation of the gradient vector. This scalar should represent the relative *magnitude* of change the entire input over the output. As such, 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 ($\mathtt{max}(L2 - norms)$) in the function.

5 Evaluation

Our goal is to experimentally analyze the following:

1. The accuracy in identifying function argument counts and types (Section 5.2); and
2. Whether the trained models learn semantics that match our domain-specific knowledge (Section 5.3).

Our experiments are performed on a server containing 2, 14-core Intel Xeon 2GHz CPUs with 64GB of RAM. The neural network and data processing routines are written in Python, using the Tensorflow platform [4].

5.1 Dataset

We evaluated EKLAVYA with two datasets. The binaries for each dataset is obtained by using two commonly used compilers: *gcc* and *clang*, with different optimization levels ranging from `O0` to `O3` for both x86 and x64. We obtained the ground truth for the function arguments by parsing the `DWARF` debug information [3].

Following the dataset creation procedure used in previous work [43], our first dataset consists of binaries from 3 popular Linux packages: *binutils*, *coreutils* and *findutils* making up 2000 different binaries, resulting from compiling each program with 4 optimization levels (`O0-O3`) using both compilers targeting both instruction sets. For x86 binaries, there are 1,237,798 distinct instructions which make up 274,285 functions. Similarly for x64, there are 1,402,220 distinct instructions which make up 274,288 functions. This dataset has several duplicate functions, and we do not use it to report our final results directly. However, an earlier version of the paper reported on this dataset; for full disclosure, we report results on this dataset in the Appendix.

For our second dataset, we extended the first dataset with 5 more packages, leading to a total of 8 packages: *binutils*, *coreutils*, *findutils*, *sg3utils*, *utillinux*, *inetutils*, *diffutils*, and *usbutils*. This dataset contains 5168 different binaries, resulting from compiling each program with 4 optimization levels (`O0-O3`) using both compilers targeting both instruction sets. For x86 binaries, there are 1,598,937 distinct instructions which constitute 370,317 functions while for x64, there are a total of 1,907,694 distinct instructions which make up 370,145 functions.

Sanitization. For our full (second) dataset, we removed functions which are duplicates of other functions in the dataset. Given that the same piece of code compiled with different binaries will result in different offsets generated, naively hashing the function body is insufficient to identify duplicates. To work around this, we chose to remove all direct addresses used by instructions found in the function. For example, the instruction '`je 0x98`' are represented as '`je `'. After the substitution, we hash the function and remove functions with the same hashes. Other than duplicates, we removed functions with less than four instructions as these small functions typically do not have any operation on arguments.

After sanitation, for x86 binaries, there are 60,061 unique functions in our second dataset. Similarly for x64, there are 59,291 functions. All our final results report on this dataset.

We use separate parts of these datasets for *training* and *testing*. We randomly sample 80% binaries of each package and designate it as the training set; the remaining 20% binaries are used for testing. Note that the training set contains all binaries of one instruction set, compiled with multiple optimization levels from both compilers. EKLAVYA is tasked to generalize from these collectively. The test results are reported on different categories of optimizations within each instruction set, to see the impact of compiler and optimization on EKLAVYA's accuracy.

Imbalanced classes. Our dataset has a different number of samples for different labels or classes. For instance, the pointer datatype is several hundred times more frequent than unions; similarly, functions with less than 3 arguments are much more frequent that those with 9 arguments. We point out that this is a natural distribution of labels in real-world binaries, not an artifact of our choice. Since training and testing on labels with very few samples is meaningless, we do not report our test results on functions with more than 9 arguments for arguments counts recovery, and the "union" and "struct" datatypes here. The overall ratio of these unreported labels totals less than 0.8% of the entire dataset. The label distributions of the training dataset are reported in the rows labeled "data distribution" in Table 1 and Table 2.

5.2 Accuracy

Our first goal is to evaluate the precision, recall, and accuracy of prediction for each of the four tasks mentioned in Section 3. Precision Pc_i and recall Rc_i are used to measure the performance of EKLAVYA for class i and are defined as:

$$Pc_i = \frac{TP_i}{TP_i + FP_i}; Rc_i = \frac{TP_i}{TP_i + FN_i}$$

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

We evaluate the accuracy of EKLAVYA by measuring the fraction of test inputs with correctly predicted labels in the test set. Readers can check that accuracy Acc can alternatively be defined as:

$$Acc = \sum_{i=1}^{n} P_i \times Rc_i$$

where n is the number of labels in testing set and P_i is the fraction of samples belonging to label i in the test runs. P_i can be seen as an estimate of the occurrence of label i in the real-world dataset and Rc_i is the probability of EKLAVYA labelling a sample as i given that its ground truth is label i.

Given that our training and testing datasets have imbalanced classes, it is helpful to understand EKLAVYA's accuracy w.r.t to the background distribution of labels in the dataset. For instance, a naive classifier that always predicts one particular label i irrespective of the given test input, will have accuracy p_i if the underlying label occurs p_i naturally in the test run. However, such a classifier will have a precision and recall of zero on labels other than i. Therefore, we report both the background data distribution of each label as well as precision and recall to highlight EKLAVYA's efficiency as a classifier.

Findings. Table 1 and Table 2 show the final results over some classes in the test dataset for each task. We have five key findings from these two tables:

(a) EKLAVYA has accuracy of around 84% for count recovery and 81% for type recovery tasks on average, with higher accuracy of over 90% and 80% respectively for these tasks on unoptimized binaries;
(b) EKLAVYA generalizes well across both compilers, `gcc` and `clang`;
(c) EKLAVYA performs well even on classes that occur less frequently, which includes samples with labels occuring as low as 2% times in the training dataset;
(d) In comparison to x86, codename has higher accuracy on x64 for count and type recovery; and,
(e) With increase in optimization levels, the accuracy of EKLAVYA drops on count recovery tasks but stays the same on type recovery tasks.

First, EKLAVYA has higher accuracy on unoptimized functions compared with previous work. The reported accuracy of previous work that uses principled use-def analysis and liveness analysis to count arguments is 78% for callers and 83% for callees [51]. It uses domain-specific heuristics about the calling convention to identify number of arguments — for example, their work mentions that if `r9` is used by a function then the function takes 6 arguments or more. However, EKLAVYA does not need such domain knowledge and obtain higher accuracy for count recovery. For example, the accuracy of EKLAVYA on x86 and x64 are 91.13% and 92.03% respectively from callers, while 92.70% and 97.48% separately from callees. For the task of type recovery, the accuracy of EKLAVYA, averaged for the first three arguments, on x86 and x64 are 77.20% and 84.55% respectively from callers, and 78.18% and 86.77% correspondingly from callees. A previous work on retargetable compilation recovers types without using machine learning techniques; however, a direct comparison is not possible since the reported results therein adopt a different measure of accuracy called conservativeness rate which cannot be translated directly to accuracy [14].

Second, EKLAVYA generalizes well over the choice of two compilers, namely `clang` and `gcc`. The accuracy of count recovery for x86 from callers and callees are 86.22% and 75.49% respectively for `gcc` binaries, and 85.30% and 80.05% for `clang` binaries. Similarly, the accuracy of type recovery (averaged for the first three arguments) on x86 from callers and callees is 80.92% and 79.04% respectively for `gcc` binaries, whereas it is 75.58% and 73.91% respectively for `clang` binaries. Though the average accuracy of `gcc` is slightly higher than `clang`, this advantage does not consistently exhibit across all classes.

Table 1: Evaluation result for argument count recovery from callers and callees for different optimization levels given different architectures. Columns 3-50 report the evaluation result of EKLAVYA on test dataset with different instruction set ranging from O0 to O3. “-” denotes that the specific metric cannot be calculated.

Arch	Task	Opt.	Metrics	Number of Arguments										Accuracy
				0	1	2	3	4	5	6	7	8	9	
x86	Task1	O0	**Data Distribution**	0.059	0.380	0.288	0.170	0.057	0.023	0.012	0.004	0.004	0.001	**0.9113**
			Precision	0.958	0.974	0.920	0.868	0.736	0.773	0.600	0.388	0.231	0.167	
			Recall	0.979	0.953	0.899	0.913	0.829	0.795	0.496	0.562	0.321	0.200	
		O1	**Data Distribution**	0.059	0.374	0.290	0.169	0.059	0.026	0.013	0.003	0.004	0.001	**0.8348**
			Precision	0.726	0.925	0.847	0.819	0.648	0.689	0.569	0.474	0.456	0.118	
			Recall	0.872	0.911	0.836	0.756	0.759	0.703	0.719	0.444	0.758	0.133	
		O2	**Data Distribution**	0.056	0.375	0.266	0.187	0.057	0.032	0.015	0.004	0.005	0.001	**0.8053**
			Precision	0.692	0.907	0.828	0.758	0.664	0.620	0.606	0.298	0.238	0.250	
			Recall	0.810	0.912	0.801	0.645	0.782	0.730	0.637	0.262	0.357	0.300	
		O3	**Data Distribution**	0.045	0.387	0.275	0.184	0.051	0.029	0.016	0.004	0.005	0.002	**0.8391**
			Precision	0.636	0.935	0.862	0.801	0.570	0.734	0.459	0.243	0.231	0.200	
			Recall	0.760	0.921	0.849	0.724	0.691	0.747	0.637	0.196	0.375	0.167	
	Task2	O0	**Data Distribution**	0.068	0.307	0.313	0.171	0.070	0.034	0.018	0.009	0.005	0.002	**0.9270**
			Precision	0.935	0.956	0.910	0.957	0.910	0.789	0.708	0.808	0.429	0.500	
			Recall	0.911	0.975	0.963	0.873	0.856	0.882	0.742	0.568	0.692	0.600	
		O1	**Data Distribution**	0.066	0.294	0.320	0.173	0.073	0.034	0.019	0.009	0.005	0.003	**0.6934**
			Precision	0.725	0.821	0.667	0.692	0.463	0.412	0.380	0.462	0.182	0.000	
			Recall	0.697	0.822	0.795	0.574	0.420	0.466	0.284	0.115	0.167	0.000	
		O2	**Data Distribution**	0.065	0.283	0.326	0.179	0.068	0.036	0.021	0.011	0.005	0.002	**0.6660**
			Precision	0.721	0.761	0.655	0.639	0.418	0.535	0.484	0.667	0.200	0.000	
			Recall	0.607	0.798	0.792	0.495	0.373	0.434	0.517	0.308	0.286	0.000	
		O3	**Data Distribution**	0.051	0.248	0.346	0.188	0.076	0.038	0.023	0.013	0.008	0.003	**0.6534**
			Precision	0.600	0.788	0.626	0.717	0.297	0.452	0.250	0.200	0.143	0.000	
			Recall	0.682	0.822	0.801	0.509	0.321	0.326	0.190	0.071	0.167	0.000	
x64	Task1	O0	**Data Distribution**	0.061	0.385	0.288	0.166	0.056	0.021	0.012	0.004	0.004	0.0	**0.9203**
			Precision	0.858	0.957	0.914	0.916	0.818	0.891	0.903	0.761	0.875	0.333	
			Recall	0.913	0.941	0.936	0.930	0.719	0.853	0.829	0.944	0.667	0.800	
		O1	**Data Distribution**	0.057	0.379	0.283	0.174	0.060	0.022	0.013	0.005	0.004	0.001	**0.8602**
			Precision	0.734	0.897	0.843	0.884	0.775	0.829	0.882	0.788	0.778	0.500	
			Recall	0.766	0.899	0.901	0.817	0.677	0.815	0.714	0.839	0.359	0.818	
		O2	**Data Distribution**	0.055	0.384	0.260	0.187	0.061	0.027	0.014	0.004	0.006	0.001	**0.8380**
			Precision	0.624	0.900	0.816	0.842	0.775	0.741	0.866	0.708	0.667	0.545	
			Recall	0.686	0.886	0.863	0.822	0.667	0.764	0.785	0.836	0.519	0.600	
		O3	**Data Distribution**	0.044	0.382	0.290	0.173	0.054	0.028	0.018	0.004	0.002	0.002	**0.8279**
			Precision	0.527	0.908	0.767	0.832	0.654	0.878	0.848	0.613	0.667	0.600	
			Recall	0.680	0.864	0.867	0.794	0.602	0.761	0.857	0.826	0.444	0.600	
	Task4	O0	**Data Distribution**	0.071	0.309	0.312	0.170	0.068	0.032	0.018	0.009	0.005	0.002	**0.9748**
			Precision	0.971	0.988	0.986	0.991	0.952	0.962	0.733	0.839	0.714	1.000	
			Recall	0.981	0.992	0.985	0.980	0.972	0.969	0.873	0.565	0.556	0.500	
		O1	**Data Distribution**	0.066	0.297	0.319	0.175	0.070	0.034	0.019	0.010	0.005	0.002	**0.7624**
			Precision	0.625	0.811	0.690	0.891	0.780	0.773	0.531	0.576	0.333	-	
			Recall	0.649	0.833	0.853	0.662	0.697	0.780	0.680	0.487	0.059	0.000	
		O2	**Data Distribution**	0.059	0.272	0.336	0.179	0.071	0.037	0.020	0.012	0.006	0.003	**0.7749**
			Precision	0.669	0.814	0.733	0.911	0.785	0.761	0.486	0.353	0.333	-	
			Recall	0.658	0.833	0.882	0.697	0.688	0.761	0.548	0.273	0.167	0.000	
		O3	**Data Distribution**	0.048	0.213	0.361	0.190	0.086	0.042	0.029	0.013	0.006	0.004	**0.7869**
			Precision	0.636	0.824	0.775	0.912	0.913	0.720	0.400	0.250	0.000	-	
			Recall	0.875	0.884	0.912	0.722	0.764	0.720	0.429	0.111	0.000	0.000	

Table 2: Evaluation result for argument type recovery from callers and callees for different optimization levels given different architectures. Columns 4-67 report the evaluation result of EKLAVYA on test dataset with different instruction sets ranging from O0 to O3. “-” denotes that the specific metric cannot be calculated.

Arch	Task	Opt.	Metrics	Type of Arguments														
				1st					2nd					3rd				
				char	int	float	pointer	enum	char	int	float	pointer	enum	char	int	float	pointer	enum
x86	Task3	O0	Data Distribution	0.0075	0.1665	0.0008	0.8008	0.0220	0.0097	0.3828	0.0002	0.5740	0.0304	0.0094	0.4225	0.0002	0.5588	0.0078
			Precision	0.5939	0.6630	1.0000	0.8954	0.5938	0.3929	0.6673	1.0000	0.8258	0.4141	0.3158	0.6245	-	0.8337	0.1429
			Recall	0.6766	0.6469	0.1429	0.9145	0.4546	0.2391	0.7302	0.0556	0.8171	0.2405	0.4615	0.7905	0.0000	0.6954	0.1111
			Accuracy			**0.8385**					**0.7547**					**0.7228**		
		O1	Data Distribution	0.0065	0.1634	0.0005	0.8101	0.0178	0.0082	0.3663	0.0001	0.5894	0.0336	0.0092	0.4274	0.0002	0.5535	0.0082
			Precision	0.5315	0.6138	1.0000	0.9027	0.8202	0.3462	0.7108	1.0000	0.8282	0.6222	0.1613	0.7220	-	0.7890	0.3200
			Recall	0.4370	0.5913	0.1539	0.9218	0.7559	0.2368	0.7482	0.1500	0.8303	0.3836	0.3333	0.7262	-	0.7867	0.2667
			Accuracy			**0.8475**					**0.7762**					**0.7537**		
		O2	Data Distribution	0.0015	0.1664	0.0002	0.8056	0.0260	0.0084	0.3505	0.0000	0.5959	0.0446	0.0072	0.4031	0.0002	0.5768	0.0116
			Precision	0.0000	0.6029	-	0.9262	0.7647	0.2500	0.6544	-	0.8193	0.6818	0.1429	0.7859	1.0000	0.7007	0.2222
			Recall	0.0000	0.6874	0.0000	0.9126	0.7879	0.0833	0.6642	0.0000	0.8442	0.3214	1.0000	0.6647	1.0000	0.8269	0.1000
			Accuracy			**0.8606**					**0.7627**					**0.7328**		
		O3	Data Distribution	0.0012	0.1731	0.0002	0.8032	0.0218	0.0069	0.3763	0.0001	0.5633	0.0523	0.0074	0.4165	0.0001	0.5647	0.0101
			Precision	0.0000	0.6561	1.0000	0.9331	0.7273	0.0000	0.6582	1.0000	0.8464	0.8462	0.5000	0.7410	-	0.8048	0.4286
			Recall	0.0000	0.7210	0.1429	0.9287	0.8000	-	0.6656	0.6667	0.8390	0.9296	1.0000	0.7613	0.0000	0.8079	0.2000
			Accuracy			**0.8794**					**0.7878**					**0.7742**		
	Task4	O0	Data Distribution	0.0056	0.1944	0.0015	0.7910	0.0052	0.0073	0.3151	0.0003	0.6654	0.0086	0.0102	0.3828	0.0016	0.5931	0.0107
			Precision	0.7500	0.7620	0.6000	0.9024	0.0870	0.5882	0.5359	1.0000	0.8856	0.3333	0.1111	0.5516	-	0.8278	0.5000
			Recall	0.5000	0.6536	0.3000	0.9400	0.2609	0.7692	0.7165	0.2500	0.7874	0.1111	0.2500	0.6447	0.0000	0.7896	0.0476
			Accuracy			**0.8582**					**0.7618**					**0.7254**		
		O1	Data Distribution	0.0060	0.2156	0.0012	0.7707	0.0049	0.0075	0.3243	0.0001	0.6587	0.0065	0.0127	0.3976	0.0020	0.5732	0.0127
			Precision	0.3333	0.5998	0.5000	0.8909	0.5833	0.0000	0.4776	-	0.8424	0.5833	0.0588	0.5268	0.0000	0.7991	0.4000
			Recall	0.1500	0.5977	0.1667	0.9049	0.2258	0.0000	0.5238	0.0000	0.8269	0.2414	0.1667	0.5765	0.0000	0.7743	0.1177
			Accuracy			**0.8305**					**0.7435**					**0.7012**		
		O2	Data Distribution	0.0041	0.2219	0.0006	0.7682	0.0050	0.0071	0.2995	0.0002	0.6841	0.0081	0.0097	0.3636	0.0000	0.6132	0.0122
			Precision	0.0000	0.7396	-	0.9125	0.0000	1.0000	0.4940	-	0.8297	1.0000	0.0000	0.4439	-	0.7633	1.0000
			Recall	0.0000	0.7188	0.0000	0.9321	0.0000	0.2500	0.5061	0.0000	0.8343	0.2500	-	0.5901	0.0000	0.6775	0.1539
			Accuracy			**0.8737**					**0.7447**					**0.6269**		
		O3	Data Distribution	0.0032	0.2050	0.0000	0.7869	0.0047	0.0039	0.2856	0.0000	0.6996	0.0103	0.0086	0.3503	0.0026	0.6221	0.0164
			Precision	0.0000	0.6759	1.0000	0.9438	0.0000	0.0000	0.4142	1.0000	0.8864	0.0000	0.0000	0.3858	-	0.8309	0.0000
			Recall	0.0000	0.7337	0.5000	0.9394	0.0000	-	0.5690	0.3333	0.8079	-	0.0000	0.6129	0.0000	0.7125	0.0000
			Accuracy			**0.8974**					**0.7607**					**0.6624**		
x64	Task3	O0	Data Distribution	0.0077	0.1721	0.0008	0.7935	0.0232	0.0101	0.3907	0.0003	0.5650	0.0307	0.0099	0.4296	0.0002	0.5508	0.0083
			Precision	0.9579	0.8404	0.5000	0.9342	0.7829	0.2381	0.7421	0.0000	0.8711	0.5818	0.2222	0.7491	-	0.8362	0.0000
			Recall	0.4893	0.7577	0.0500	0.9747	0.7126	0.2778	0.7551	0.0000	0.8974	0.2743	0.2500	0.7254	0.0000	0.8661	0.0000
			Accuracy			**0.9156**					**0.8182**					**0.8028**		
		O1	Data Distribution	0.0062	0.1608	0.0005	0.8073	0.0235	0.0079	0.3795	0.0003	0.5761	0.0338	0.0081	0.4389	0.0001	0.5438	0.0077
			Precision	0.9474	0.8457	-	0.9202	0.5872	0.3846	0.6831	1.0000	0.8578	0.5562	0.2222	0.7447	0.0000	0.8375	0.0000
			Recall	0.3000	0.6871	0.0000	0.9771	0.7214	0.1852	0.7340	0.2353	0.8573	0.2973	0.2857	0.7481	0.0000	0.8491	0.0000
			Accuracy			**0.9038**					**0.7870**					**0.7985**		
		O2	Data Distribution	0.0016	0.1520	0.0001	0.8193	0.0265	0.0077	0.3632	0.0001	0.5797	0.0483	0.0079	0.4417	0.0001	0.5404	0.0090
			Precision	-	0.8630	1.0000	0.9269	0.7217	0.0000	0.6712	1.0000	0.8783	0.8556	0.0000	0.7741	1.0000	0.8426	0.0000
			Recall	0.0000	0.7408	0.1000	0.9745	0.8925	0.0000	0.7004	0.1250	0.8918	0.4477	-	0.7608	0.5000	0.8733	0.0000
			Accuracy			**0.9121**					**0.8181**					**0.8135**		
		O3	Data Distribution	0.0007	0.1847	0.0001	0.7932	0.0206	0.0079	0.3933	0.0000	0.5461	0.0513	0.0070	0.4200	0.0000	0.5569	0.0142
			Precision	-	0.8633	-	0.9271	0.8611	0.0000	0.7021	-	0.8739	0.7273	0.0000	0.6885	-	0.8286	0.0000
			Recall	0.0000	0.7538	0.0000	0.9755	0.8378	-	0.7003	0.0000	0.8754	0.7742	-	0.7395	0.0000	0.8085	0.0000
			Accuracy			**0.9155**					**0.8203**					**0.7663**		
	Task4	O0	Data Distribution	0.0057	0.2006	0.0015	0.7843	0.0055	0.0074	0.3223	0.0005	0.6581	0.0083	0.0107	0.3879	0.0013	0.5891	0.0094
			Precision	0.6842	0.8987	0.8000	0.9777	0.2000	0.6000	0.7214	1.0000	0.9221	0.1429	0.2500	0.5782	1.0000	0.8880	0.4444
			Recall	0.6191	0.9301	0.4000	0.9789	0.0625	0.5455	0.7314	0.1667	0.9260	0.0769	1.0000	0.7398	0.1250	0.8199	0.1333
			Accuracy			**0.9562**					**0.8725**					**0.7742**		
		O1	Data Distribution	0.0055	0.2033	0.0010	0.7830	0.0058	0.0069	0.3128	0.0005	0.6685	0.0090	0.0116	0.3816	0.0011	0.5938	0.0103
			Precision	0.7143	0.7936	0.4000	0.9714	0.1429	0.1818	0.6157	1.0000	0.9071	0.3333	0.2857	0.4703	-	0.8677	0.1667
			Recall	0.3125	0.9053	0.2500	0.9444	0.0769	0.2222	0.6801	0.4000	0.8828	0.0909	1.0000	0.7457	0.0000	0.7035	0.0345
			Accuracy			**0.9240**					**0.8267**					**0.6918**		
		O2	Data Distribution	0.0042	0.2261	0.0002	0.7639	0.0051	0.0056	0.2956	0.0003	0.6897	0.0074	0.0090	0.3667	0.0013	0.6110	0.0110
			Precision	0.0000	0.8067	-	0.9726	-	0.0000	0.6014	1.0000	0.9014	-	0.0000	0.5206	-	0.8428	0.0000
			Recall	0.0000	0.9311	0.0000	0.9473	0.0000	0.0000	0.6692	0.5000	0.8777	0.0000	-	0.7128	0.0000	0.7569	0.0000
			Accuracy			**0.9305**					**0.8240**					**0.7093**		
		O3	Data Distribution	0.0030	0.2341	0.0004	0.7576	0.0049	0.0058	0.2917	0.0005	0.6937	0.0083	0.0152	0.3660	0.0000	0.5989	0.0200
			Precision	-	0.7250	-	0.9894	-	-	0.6136	-	0.9172	-	-	0.4700	-	0.8553	0.3333
			Recall	0.0000	0.9667	-	0.9256	-	-	0.6750	-	0.8944	-	-	0.6912	0.0000	0.7647	0.0769
			Accuracy			**0.9256**					**0.8507**					**0.6980**		

Third, EKLAVYA has high precision and recall on categories that occur relatively less frequently in our dataset. For example, the inputs with 4 arguments only count for around 6% in our training set, whereas the precision and recall of count recovery from callers are around 67% and 78% separately on x86. Similarly, inputs whose first argument is "enum" data type only occupy around 2% over our training set. However, the precision and recall of type recovery are around 76% and 69% from callers on x86.

Fourth, the accuracy of EKLAVYA on x64 is higher than x84. As shown in Table 1, the average accuracy of EKLAVYA for counts recovery task are 1.4% (from callers) and 9.0% (from callees) higher for x64 binaries than x86. Type recovery tasks exhibit a similar finding. Table 2 shows that the accuracy averaged for the task of recovering types for the first, second, and third arguments. EKLAVYA has an average accuracy $3-9\%$ higher for a given task on x64 than of the same task on x86 binaries. This is possibly because x86 has fewer registers, and most argument passing is stack-based in x86. EKLAVYA likely recognizes registers better than stack offsets.

Finally, the accuracy of the model with respect to the optimization levels is dependent on type of task. Optimization levels do *not* have a significant effect on the accuracy of the predictions in type recovery tasks , whereas the EKLAVYA performs better on `O0` than on `O1 - O3` for arguments counts recovery. For example, the accuracy of type recovery for the first argument from callers on `O0 - O3` are nearly the same, which is around 85% on x86. But, the accuracy for count recovery from callers on `x86`, for instance, is 91.13%, which drops to 83.48% when we consider binaries compiled with `O1`. The accuracy for count recovery does not change significantly for optimization levels `O1` to `O3`.

5.3 Explicability of Models

Our guiding principle in selecting the final architecture is its explicability. In this section, we present our results from qualitatively analyzing what EKLAVYA learns. We find that EKLAVYA automatically learns the semantics and similarity between instructions or instruction set, the common compiler conventions or idioms, and instruction patterns that differentiate the use of different values. This strengthens our belief that the model learns information that matches our intuitive understanding.

5.3.1 Instruction Semantics Extraction

In this analysis, we employ t-SNE plots and analogical reasoning to understand the relations learned by the word embedding model between instructions.

Figure 3: t-SNE visualization of `mov` instructions on x64. Each dot represent one `mov` instruction. Red dots are where Figure 4 is.

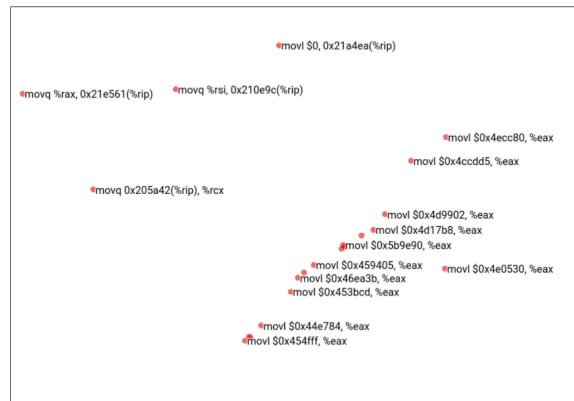

Figure 4: t-SNE visualization of a cluster of `mov $constant, %register` instructions on x64.

Semantic clustering of instructions. t-SNE plots allow us to project the points on the 256 dimension space to that of a two-dimensional image giving us a visualization of the clustering. Figure 3 shows one cluster corresponding to `mov` family of instructions, which EKLAVYA learns to have similarity. Due to a large amount of instructions (over a million), a complete t-SNE plot is difficult to analyze. Therefore, we randomly sampled 1000 instructions from the complete set of instructions, and select all instructions belonging to the `mov` family. This family consists of 472 distinct instruction vectors which we project onto a two-dimension space using t-SNE.

Then we "zoom-in" Figure 3 and show two interesting findings. These two findings are shown in Figure 4 and Figure 5. In Figure 4, we recognize `mov $constant, %register` instructions, which indicates that EKLAVYA recognizes the similarity between

Figure 5: t-SNE visualization of `mov constant(%rip), %register` and `mov $constant, %register` instructions on x64.

all instructions that assign constant values to registers, and abstract out the register. Figure 5 shows that EKLAVYA learns the similar representation for `mov constant(%rip), %register` instructions. These two findings show the local structures that embedding model learned within "mov" family.

Relation between instructions. We use analogical reasoning techniques to find similarity between sets of instructions. In this paper, we show two analogies that our embedding model learned. The first example is that cosine distance between the instructions in the pair (`push %edi`, `pop %edi`) is nearly the same as the distance between instructions in the pair (`push %esi`, `pop %esi`). This finding corresponds to the fact that the use of `push-pop` sequences on one register is analogous to the use of `push-pop` sequences on another register. In essence, this finding shows that the model abstracts away the operand register from the use of `push-pop` (stack operation) instructions on x86/x64. As another example, we find that the distance between the instructions in the pair (`sub $0x30,%rsp`, `add $0x30,%rsp`) and the distance between the pair (`sub $0x20,%rsp`, `add $0x20,%rsp`) is nearly the same. This analysis exhibits that EKLAVYA recognizes that the integer operand can be abstracted away from such sequences (as long the same integer value is used). These instruction pairs are often used to allocate / deallocate the local frame in a function, so we find that EKLAVYA correctly recognizes their analogical use across functions. Due to space reasons, we limit the presented examples to three. In our manual investigation, we find several such semantic analogies that are auto-learned.

Table 3: The relative score of importance generated by saliency map for each instruction from four distinct functions to determine the number of arguments given the whole function.

Instruction	Relative Score
pushl %ebp	0.149496
movl %esp, %ebp	0.265591
pushl %ebx	0.179169
subl $0x14, %esp	**0.370329**
movl 0xc(%ebp), %eax	**1.000000**
movl 8(%ebp), %ecx	**0.509958**
leal 0x8090227, %edx	0.372616
...	

(a) "print_name_without_quoting" compiled with clang and O0 on 32-bit (having 2 arguments)

Instruction	Relative Score
...	
subq $0x38, %rsp	**0.356728**
movq %r8, %r13	**1.000000**
movq %rcx, %r15	0.214237
movq %rdx, %rbx	0.140916
movq %rsi, 0x10(%rsp)	0.336599
movq %rdi, 0x28(%rsp)	0.253754
...	

(b) "parse_stab_struct_fields" compiled with clang and O1 on 64-bit (having 5 arguments)

Instruction	Relative Score
...	
subq $0x80, %rsp	**1.000000**
leaq (%rsp), %rdi	**0.683561**
xorl %eax, %eax	0.161366
movl $0x10, %ecx	**0.658702**
...	
movl %ecx, %eax	0.049905
...	

(c) "EmptyTerminal" compiled with clang and O1 on 64-bit (having 0 arguments)

Instruction	Relative Score
...	
subq $0x40, %rsp	**0.411254**
movq %rdi, -0x10(%rbp)	**0.548005**
movq %rsi, -0x18(%rbp)	**1.000000**
movq %rdx, -0x20(%rbp)	**0.725123**
movq %rcx, -0x28(%rbp)	**0.923426**
movq -0x10(%rbp), %rcx	**0.453617**
movq %rcx, -0x30(%rbp)	0.129167
...	
addq %rdx, %rcx	0.093260
...	

(d) "check_sorted" compiled with clang and O0 on 64-bit (having 4 arguments)

5.3.2 Auto-learning Conventions

Next, we analyze which input features are considered important by EKLAVYA towards making a decision on a given input. We use the saliency map to score the relative importance of each instruction in the input function. Below, we present our qualitative analysis to identify the conventions and idioms that EKLAVYA auto-learns. For each case below, we compute saliency maps for 20 randomly chosen functions for which EKLAVYA correctly predicts signatures, and inspect them manually.

We find that instructions that are marked as high in relative importance for classification suggest that EKLAVYA auto-learns several important things. We find consistent evidence that EKLAVYA learns calling conventions and idioms, such as the argument passing conventions, "use-before-write" instructions, stack frame allocation instructions, and setup instructions for stack-based arguments to predict the number of arguments accepted by the function. EKLAVYA consistently identifies instructions that differentiate types (e.g. `pointers` from `char`) as important.

Identification of argument registers. We find that the RNN model for counting arguments discovers the specific registers used to pass the arguments. We selected 20 sample functions for which types were correctly predicted, and we consistently find that the saliency map marks instructions processing caller-save and callee-save registers as most important. Consider the function `parse_stab_struct_fields` shown in Table 3

as example, wherein the RNN model considers the instruction `movq %r8, %r13`; `movq %rcx, %r15`; `movq %rdx, %rbx`; `movq %rsi, 0x10(%rsp)` and `movq %rdi, 0x28(%rsp)` as the relatively most important instructions for determining the number of arguments, given the whole function body. This matches our manual analysis which shows that `rdi`, `rsi`, `rdx`, `rcx`, `r8` are used to pass arguments. We show 4 different functions taking different number of arguments as parameters in Table 3. In each example, one can see that the RNN identifies the instructions that first use the incoming arguments as relatively important compared to other instructions.

Further, EKLAVYA seems to correctly place emphasis on the instruction which reads a particular register before writing to it. This matches our intuitive way of finding arguments by identifying "use-before-write" instructions (with liveness analysis). For example, in the function `check_sorted` (Table 3(d)), the register `rcx` is used in a number of instructions. The saliency map marks the most important instruction to be the correct one that uses the register before write. Finally, the function `EmptyTerminal` also shows evidence EKLAVYA is not blindly memorizing register names (e.g. `rcx`) universally for all functions. It correctly de-emphasizes that the instruction `movq %ecx, %eax` is not related to argument passing. In this example, `rcx` has been clobbered before in the instruction `movl $0x10, %ecx` on `rcx` before reaching the `movq` instruction, and EKLAVYA accurately recognizes that `rcx` is not used as an argument here. We have manually analyzed this finding consistently on 20 random samples we analyzed.

Argument accesses after local frame creation. In our analyzed samples, EKLAVYA marks the arithmetic instruction that allocates the local stack frame as relatively important. This is because in the compilers we tested, the access to arguments begins after the stack frame pointer has been adjusted to allocate the local frame. EKLAVYA learns this convention and emphasizes its importance in locating instructions that access arguments (see Table 3).

We highlight two other findings we have confirmed manually. First, EKLAVYA correctly identifies arguments passed on the stack as well. This is evident in 20 functions we sampled from the set of functions that accept arguments on stack, which is a much more common phenomenon in x86 binaries that have fewer registers. Second, the analysis of instructions passing arguments from the body of the caller is nearly as accurate as that from that of callees. A similar saliency map based analysis of the caller's body identifies the right registers and setup of stack-based arguments are consistently marked as relatively high in importance. Due to space reasons, we have not shown the salience maps for these examples here.

Table 4: The relative score of importance generated by saliency map for each instruction from four distinct functions to determine the type of arguments given the whole function.

Instruction	Relative Score
subl $0xc, %esp	0.297477
movl 0x10(%esp), %edx	**0.861646**
movzbl 0x28(%edx), %eax	**1.000000**
movl %eax, %ecx	0.332725
andl $7, %ecx	0.481093
cmpb $1, %cl	0.248921
...	

(a) "bfd_set_symtab" compiled with gcc-32-O2 (1st argument - pointer)

Instruction	Relative Score
...	
subq $0x328, %rsp	0.774363
movq %rcx, %r12	0.881474
movq %rdx, %r15	0.452816
movq %rsi, %rbx	**0.363804**
movq %rdi, %r14	0.442176
movl (%rbx), %eax	**1.000000**
...	

(b) "do_fprintf" compiled with clang-64-O1 (2nd argument - pointer)

Instruction	Relative Score
pushl %ebx	0.235036
subl $0x10, %esp	0.383451
fldl 0x1c(%esp)	**1.000000**
movl 0x18(%esp), %ecx	**0.511937**
flds 0x8050a90	0.873672
fxch %st(1)	0.668212
...	

(c) "dtotimespec" compiled with gcc-32-O3 (2nd argument - float)

Instruction	Relative Score
...	
movl %ecx, %r15d	0.431204
movq %rdx, %r14	0.399483
movzbl (%rsi), %ebp	**1.000000**
testb $0x20, 0x20b161(%rip)	0.336855
jne 0x2d	0.254520
movl 0x18(%r14), %eax	0.507721
movq 0x20b15c(%rip), %rcx	0.280275
...	

(d) "print_icmp_header" compiled with clang-64-O1 (2nd argument - pointer)

Operations to type. With a similar analysis of saliency maps, we find that EKLAVYA learns instruction patterns to identify types. For instance, as shown in examples of Table 4, the saliency map highlights the relative importance of instructions. One can see that instructions that use byte-wide registers (e.g. `dl`) are given importance when EKLAVYA predicts the type to be `char`. This matches our semantic understanding that the `char` type is one byte and will often be used in operands of the corresponding bit-width. Similarly, we find that in cases where EKLAVYA predicts the type to be a `pointer`, the instructions marked as important have indirect register base addressing with the right registers carrying the pointer values. Where `float` is correctly predicted, the instructions highlighted involve XMM registers or floating point instructions. These findings consistently exhibit in our sampled sets, showing that EKLAVYA mirrors our intuitive understanding of the semantics.

5.3.3 Network Mispredictions

We provide a few concrete examples of EKLAVYA mispredictions. These examples show that principled program analysis techniques would likely discern such errors; therefore, EKLAVYA does *not* mimic a full liveness tracking function yet. To perform this analysis, we inspect a random subset of the mispredictions for each of the tasks using the saliency map. In some cases, we can speculate the reasons for mispredictions, though there are best-effort estimates. Our findings are presented in the form of 2 case studies below.

As shown in Table 5, the second argument is mis-

Table 5: x86 multiple type mispredictions for second arguments.

Instruction	Relative Score
subl $0x1c, %esp	0.719351
movsbl 0x24(%esp), %eax	1.000000
movl %eax, 8(%esp)	0.246975
movl $0xffffffff, 4(%esp)	0.418808
movl 0x20(%esp), %eax	0.485717
movl %eax, (%esp)	0.260028
calll 0xffffff3e	0.801598
addl $0x1c, %esp	0.403249
retl	0.383143

(a) "quotearg_char" compiled with gcc and O1 (true type is char but predicted as int)

Insturction	Relative Score
pushl %edi	0.545965
movl %edx, %edi	0.145597
pushl %esi	0.021946
pushl %ebx	0.068469
movl %eax, %ebx	0.188693
subl $0x20, %esp	0.446094
movl 0xc(%eax), %eax	0.890956
movl $0, 0x1c(%esp)	1.000000
leal 0x1c(%esp), %esi	0.805058
cmpb %dl, (%eax)	0.824601
...	

(b) "d_exprlist" compiled with gcc and O2 (true type is char but predicted as pointer)

Table 6: x64 mispredictions.

Instruction	Relative Score
pushq %rbx	0.175079
movq %rdi, %rbx	0.392229
callq 0x3fc	1.000000
testq %rax, %rax	0.325375
je 0x1004	0.579551
popq %rbx	0.164043
retq	0.135274
movq %rbx, %rdi	0.365685
callq 0xe6d	0.665486

(a) "ck_fopen" compiled with clang and O1 (true type of first argument is pointer but predicted as int)

Instruction	Relative Score
...	
pushq %rbx	0.025531
subq $0x100, %rsp	0.163929
movq %rdi, -0xe8(%rbp)	0.314619
movq %rsi, -0xf0(%rbp)	0.235489
movl %edx, %eax	0.308323
movq %rcx, -0x100(%rbp)	0.435364
movl %r8d, -0xf8(%rbp)	0.821577
movq %r9, -0x108(%rbp)	1.000000
movb %al, -0xf4(%rbp)	0.24482
...	

(b) "prompt" compiled with gcc and O0 (number of arguments is 6 but predicted as 7)

predicted as an `integer` in the first example, while in the second case study, the second argument is mispredicted as a `pointer`. From these two examples, it is easy to see how the model has identified instructions which provide hints to what the types are. In both cases, the highlighted instructions suggest possibilities of multiple types and the mispredictions corresponds to one of it. The exact reasons for mispredictions are unclear but this seems to suggest that the model is not robust against situations where there can be multiple type predictions for different argument positions. We speculate that this is due to the design choice of training for each specific argument position a separate sub-network which potentially requires the network to infer calling conventions from just type information.

In the same example as above, the first argument is mispredicted as well. The ground truth states that the first argument is a `pointer`, whereas EKLAVYA predicts an `integer`. This shows another situation where the model makes a wrong prediction, namely when the usage of the argument within the function body provides insufficient hints for the type usage.

We group all mispredictions we have analyzed into three categories: insufficient information, high argument counts and off-by-one errors. A typical example of a misprediction due to lack of information is when the function takes in more arguments than it actually uses. The first example in Table 6 shows an example of it.

Typically, for a functions with high argument counts (greater than 6), the model will highlight the use of `%r9` and some subsequent stack uses. However in example 2 of Table 6, it shows how the model focuses on `%r9` but still made the prediction of an argument count of 7. The lack of training data for such high argument counts may be a reason for lack of robustness.

Off-by-one errors are those in which the network is able to identify instructions which indicate the number of arguments but the prediction is off by one. For example, the network may identify the use of `%rcx` as important but make the prediction that there are 5 arguments instead of 4 arguments. No discernible reason for these has emerged in our analysis.

6 Related Work

Machine Learning on Binaries. Extensive literature exists on applying machine learning for other binaries analysis tasks. Such tasks include malware classification [42, 5, 30, 20, 36, 15] and function identification [37, 7, 43]. The closest related work to ours is by Shin et al. [43], which apply RNNs to the task of function boundary identification. These results have high accuracy, and such techniques can be used to create the inputs for EKLAVYA. At a technical level, our work employs word-embedding techniques and we perform in-depth analysis of the model using dimensionality reduction, analogical reasoning and saliency maps. These analysis techniques have not been used in studying the learnt models for binary analysis tasks. For function identification, Bao et al. [7] utilize weighted prefix trees to improve the efficiency of function identification. Many other works use traditional machine learning techniques such as n-grams analysis [42], SVMs [36], and conditional random fields [37] for binary analysis tasks (different from ours).

Word embedding is a commonly used technique in such tasks, since these tasks require a way to represent words as vectors. These word embeddings can generally be categorized into two approaches, count-based [13, 32] and prediction-based [24, 28]. Neural networks are also frequently used for tasks like language translation [11, 49], parsing [46, 45].

Function Arguments Recovery. In binary analysis, recovery of function arguments [51, 23, 14] is an important component used in multiple problems. Some examples of the tasks include hot patching [33] and fine-grained control-flow integrity enforcement [51]. To summarize, there are two main approaches used to recover the function argument: liveness analysis and heuristic methods based on calling convention and idioms. Veen et. al. [51] in their work make use of both these methods

to obtain the function argument counts. Lee et. al. [23] formulate the usage of different data types in binaries to do type reconstruction. In addition, ElWazeer et al.[14] apply liveness analysis to provide a fine-grained recovery of arguments, variables and their types. A direct comparison to this work is difficult because their work considers a different type syntax than our work. At a high level, EKLAVYA provides a comparable level of accuracy, albeit on more coarse-grained types.

7 Conclusion

In this paper, we present a neural-network-based system called EKLAVYA for addressing function arguments recovery problem. EKLAVYA is compiler and instruction-set agnostic system with comparable accuracy. In addition, we find that EKLAVYA indeed learns the calling conventions and idioms that match our domain knowledge.

8 Acknowledgements

We thank the anonymous reviewers of this work for their helpful feedback. We thank Shweta Shinde, Wei Ming Khoo, Chia Yuan Cho, Anselm Foong, Jun Hao Tan and RongShun Tan for useful discussion and feedback on earlier drafts of this paper. We also thank Valentin Ghita for helping in the preparation of the final dataset. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21). This research is also supported in part by a research grant from DSO, Singapore. All opinions expressed in this paper are solely those of the authors.

References

[1] GitHub - eliben/pyelftools: Pure-python library for parsing ELF and DWARF. `https://github.com/eliben/pyelftools`.

[2] GNU Binutils. `https://www.gnu.org/software/binutils/`.

[3] The DWARF Debugging Standard. `http://www.dwarfstd.org/`.

[4] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] ABOU-ASSALEH, T., CERCONE, N., KESELJ, V., AND SWEIDAN, R. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International* (2004), vol. 2, IEEE, pp. 41–42.

[6] ANDRIESSE, D., CHEN, X., VAN DER VEEN, V., SLOWINSKA, A., AND BOS, H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium* (2016).

[7] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *USENIX Security* (2014), pp. 845–860.

[8] BENGIO, Y., DUCHARME, R., VINCENT, P., AND JAUVIN, C. A neural probabilistic language model. *Journal of machine learning research 3*, Feb (2003), 1137–1155.

[9] BENGIO, Y., SIMARD, P., AND FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks 5*, 2 (1994), 157–166.

[10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices 46*, 3 (2011), 265–278.

[11] CHO, K., VAN MERRIËNBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)* (2014).

[12] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)* (2005), IEEE, pp. 32–46.

[13] DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. Indexing by latent semantic analysis. *Journal of the American society for information science 41*, 6 (1990), 391.

[14] ELWAZEER, K., ANAND, K., KOTHA, A., SMITHSON, M., AND BARUA, R. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Notices 48*, 6 (2013), 51–60.

[15] FIRDAUSI, I., ERWIN, A., NUGROHO, A. S., ET AL. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on* (2010), IEEE, pp. 201–203.

[16] FRIEDMAN, S. E., AND MUSLINER, D. J. Automatically repairing stripped executables with cfg microsurgery. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2015 IEEE International Conference on* (2015), IEEE, pp. 102–107.

[17] GHORMLEY, D. P., RODRIGUES, S. H., PETROU, D., AND ANDERSON, T. E. Slic: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference* (1998), vol. 98.

[18] HEMEL, A., KALLEBERG, K. T., VERMAAS, R., AND DOLSTRA, E. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011), ACM, pp. 63–72.

[19] HOTELLING, H. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology 24*, 6 (1933), 417.

[20] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 470–478.

[21] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14* (2005), USENIX Association, pp. 11–11.

[22] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting kernel-level rootkits through binary analysis. In *Computer Security Applications Conference, 2004. 20th Annual* (2004), IEEE, pp. 91–100.

[23] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs.

[24] LEVY, O., GOLDBERG, Y., AND DAGAN, I. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics 3* (2015), 211–225.

[25] MAATEN, L. V. D., AND HINTON, G. Visualizing data using t-sne. *Journal of Machine Learning Research 9*, Nov (2008), 2579–2605.

[26] MCCAMANT, S., AND MORRISETT, G. Evaluating sfi for a cisc architecture. In *Usenix Security* (2006), vol. 6.

[27] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *CoRR abs/1301.3781* (2013).

[28] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[29] MIKOLOV, T., YIH, W.-T., AND ZWEIG, G. Linguistic regularities in continuous space word representations. In *Proceedings of NAACL-HLT* (2013), pp. 746–751.

[30] MOSKOVITCH, R., FEHER, C., TZACHAR, N., BERGER, E., GITELMAN, M., DOLEV, S., AND ELOVICI, Y. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*. Springer, 2008, pp. 204–215.

[31] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In In Proceedings of the 12th Network and Distributed Systems Security Symposium* (2005), Citeseer.

[32] PENNINGTON, J., SOCHER, R., AND MANNING, C. D. Glove: Global vectors for word representation. In *EMNLP* (2014), vol. 14, pp. 1532–1543.

[33] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., ET AL. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 87–102.

[34] PRAKASH, A., HU, X., AND YIN, H. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS* (2015).

[35] QIAO, RUI AND SEKAR, R. Effective Function Recovery for COTS Binaries using Interface Verification. Tech. rep., Department of Computer Science, Stony Brook University, May 2016.

[36] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), Springer, pp. 108–125.

[37] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *AAAI* (2008), pp. 798–804.

[38] SÆBJØRNSEN, A., WILLCOCK, J., PANAS, T., QUINLAN, D., AND SU, Z. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), ACM, pp. 117–128.

[39] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (2009), ACM, pp. 225–236.

[40] SAXENA, P., SEKAR, R., AND PURANIK, V. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), ACM, pp. 74–83.

[41] SCHULTE, E. M., WEIMER, W., AND FORREST, S. Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015), ACM, pp. 847–854.

[42] SCHULTZ, M. G., ESKIN, E., ZADOK, F., AND STOLFO, S. J. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 38–49.

[43] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 611–626.

[44] SIMONYAN, K., VEDALDI, A., AND ZISSERMAN, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *ICLR Workshop* (2014).

[45] SOCHER, R., LIN, C. C., MANNING, C., AND NG, A. Y. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (2011), pp. 129–136.

[46] SOCHER, R., MANNING, C. D., AND NG, A. Y. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop* (2010), pp. 1–9.

[47] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security* (2008), Springer, pp. 1–25.

[48] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[49] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.

[50] TORGERSON, W. S. Multidimensional scaling: I. theory and method. *Psychometrika 17*, 4 (1952), 401–419.

[51] VAN DER VEEN, V., GÖKTAS, E., CONTAG, M., PAWLOWSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[52] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 299–308.

[53] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 559–573.

[54] Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *Usenix Security* (2013), vol. 13.

A Evaluation on the First Dataset

In this section, we will highlight the importance of having a good dataset. To do this, we will look at the accuracy evaluation using the dataset consisting of only *coreutils*, *binutils* and *findutils*. Table 7 depicts the results of the evaluation. Qualitative analysis of the results remains largely the same. For example, the high median and low minimum F1 indicates that EKLAVYA mispredicts for some cases of which we have verified that these mis-predicted classes correspond to classes that are under-represented in our training set. However, a key difference we observed is the actual accuracy of the results. The accuracy of the smaller, unsanitized dataset is consistently high even in cases where we expect otherwise. For example, the F1 score for argument counting task is consistently over 0.90 even across optimization levels. We speculate that the difference in the accuracy is due to the presence of similar functions across the binaries. Manual inspection into the dataset confirms that there is indeed significant shared code amongst the binaries skewing the results. We find that it is not uncommon for programs within the same package, or even across packages to share the same static libraries or code. This problem is especially pronounced in binaries within the same package as these binaries typically share common internal routines. Note that this problem exists for binaries between packages too. There have been examples of functions of binaries from different packages having different names but is nearly identical in terms of the binary code. In our paper, we propose a simple method to remove similar functions but a better way of quantifying the similarities can be utilized to generate a more robust dataset. Finally, we hope that this can be built upon into a high quality, publicly available binary dataset where future binary learning approaches can be evaluated on.

B Short Primer on t-SNE

To maintain the neighborhood identity, t-SNE first use the conditional probabilities to represent the euclidean distance between high-dimension dataset. For instance, the similarity between two distinct instruction I_i and I_j is represented as the conditional probability p_{ij}. The conditional probability has following definition:

$$p_{j|i} = \frac{exp(-\left\|I_i - I_j\right\|^2/(2\sigma_i^2))}{\sum_{k\neq i} exp(-\|I_i - I_k\|^2/(2\sigma_i^2))}$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

where n is the number of data points and σ is the variance of distribution which is centered at each data point x_i. Here, t-SNE determines the value of σ_i by binary search with the given perplexity value.

The perplexity can be considered as the measurement of valid number of neighbors, which is defined as:

$$perplexity(p_i) = x^{H(p_i)}$$

$$H(p_i) == \sum_j p_{j|i} log_2 p_{j|i}$$

The second step is to minimize the difference between the conditional probability between high-dimensional dataset and low-dimensional dataset. For the conditional probability q_{ij} of low-dimensional data point y_i and y_j, t-SNE applies similar method:

$$q_{ij} = \frac{(1 + \left\|y_i - y_j\right\|^2)^{-1}}{\sum_{k\neq m}(1 + \|y_k - y_m\|^2)^{-1}}$$

Given the conditional probabilities, we can apply gradient descent method to do the minimization task.

Table 7: Evaluation result on the first dataset for count recovery and type recovery tasks from callers and callees for different optimization levels given different architectures.Columns 3-18 report the evaluation result of EKLAVYA on test dataset with different optimization level ranging from O0 to O3. The median, max, and min F1 are calculated over the reported labels, whereas the accuracy is calculated over the whole test set.

Arch	Task		O0				O1				O2				O3			
			Median F1	Max F1	Min F1	Acc	Median F1	Max F1	Min F1	Acc	Median F1	Max F1	Min F1	Acc	Median F1	Max F1	Min F1	Acc
x86	Task1		0.978	0.994	0.923	0.983	0.960	0.991	0.925	0.972	0.968	0.997	0.938	0.977	0.967	0.998	0.936	0.979
	Task2		0.984	0.993	0.952	0.986	0.965	0.988	0.933	0.967	0.970	0.982	0.948	0.973	0.966	0.982	0.942	0.972
	Task3	1st	0.915	0.989	-	0.979	0.934	0.990	0.400	0.983	0.950	0.991	-	0.985	0.968	0.993	-	0.988
		2nd	0.981	1.000	0.904	0.976	0.980	1.000	0.909	0.976	0.981	1.000	-	0.984	0.984	1.000	-	0.984
		3rd	0.962	0.982	-	0.978	0.976	0.993	0.500	0.981	0.988	1.000	0.926	0.985	0.977	1.000	0.667	0.984
	Task4	1st	0.983	0.994	0.857	0.989	0.994	1.000	0.945	0.990	0.997	1.000	0.750	0.994	0.972	0.997	0.857	0.994
		2nd	0.980	1.000	0.975	0.987	0.989	1.000	0.976	0.988	0.984	0.996	-	0.993	0.985	0.996	-	0.993
		3rd	0.986	1.000	0.714	0.991	0.983	0.998	0.727	0.989	0.985	1.000	0.800	0.989	0.986	1.000	0.667	0.989
x64	Task1		0.985	0.996	0.967	0.985	0.975	0.997	0.873	0.971	0.978	0.997	0.934	0.979	0.977	0.999	0.946	0.982
	Task2		0.997	0.999	0.975	0.998	0.976	0.988	0.942	0.976	0.980	0.991	0.946	0.979	0.979	0.991	0.950	0.978
	Task3	1st	0.934	0.992	0.667	0.984	0.938	0.992	0.400	0.985	0.954	0.993	-	0.987	0.969	0.994	-	0.989
		2nd	0.984	1.000	0.975	0.980	0.985	1.000	0.978	0.982	0.985	1.000	-	0.986	0.987	0.990	-	0.990
		3rd	0.970	0.991	0.667	0.987	0.988	0.997	0.800	0.991	0.993	1.000	0.988	0.992	0.995	1.000	0.990	0.994
	Task4	1st	0.987	0.997	0.667	0.995	0.981	0.995	0.667	0.991	0.970	0.996	0.857	0.993	0.971	0.997	0.857	0.994
		2nd	0.991	1.000	0.667	0.989	0.984	0.993	0.667	0.989	0.997	1.000	-	0.996	0.997	1.000	-	0.995
		3rd	0.983	0.993	0.857	0.989	0.984	1.000	0.727	0.990	0.985	1.000	0.800	0.991	0.988	1.000	0.800	0.992