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
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.

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.
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 system 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 “exlicable”. 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 a priori 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:


(a) the use of push/pop instructions for register save-restore;
(b) the knowledge of rsp as a special stack pointer register;
(c) the use of 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.

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 re-targeted to binaries from newer instruction sets, analysis tools need to be constantly updated with new rules or target backends. An ideal solution would 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 (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, \( \mathcal{M} \), 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 \( T_a[j] \) respectively denote the disassembled code and the \( j \)th bytes of a target function \( a \). Then, the \( k \)th instruction of function \( a \) can be defined as:

\[
I_a[k] := \langle T_a[m], T_a[m+1], ..., T_a[m+l] \rangle
\]

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:

\[
T_a := \langle I_a[1], I_a[2], ..., I_a[p] \rangle
\]

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 the instructions in \( b \) preceding the call instruction. We call this a caller snippet \( C_{b,a}[j] \), defined as:

\[
C_{b,a}[j] := \langle I_b[0], I_b[1], ..., I_b[j-1] \rangle
\]

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 \( D_a \) is defined as:

\[
D_a := T_a \cup \left( \bigcup_{b \in S_a} \left( \bigcup_{0 \leq j \leq |T_b|} C_{a,b}[j] \right) \right)
\]

where \( S_a \) is the set of functions that call \( a \).

---

\(^3\)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 $M$, which is used to decide two properties for a target function $a$, from given data $D_a$, 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:

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

Note that the above definition gives the inputs and outputs of the model $M$, which can be queried for a target function. This is called the test set. For training the model $M$, the training set has a similar representation. It consists of the disassembled functions input $D_a$ 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 context-
ual use in the binary, such as by analyzing which group
appears sequentially together or in certain contexts rela-
tive to other groups. One general approach to extracting
contextual relationships is to employ a technique called
word embedding [8]. Word embedding in EKLAVYA con-
verts 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 dis-
tance between instructions encodes relationships. For
instance, the relative distance between the vectors for
push %edi and pop %edi is similar to distance be-
tween 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 func-
tion boundaries [43]. One could represent the $i^{th}$ instruc-
tion by a vector with its $i^{th}$ element as 1 and all other
elements set to 0. For example, if there are 5 differ-
ent 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 us-
able for many different binary analysis tasks, since a sepa-
rate 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 nega-
tive sampling method outlined in the paper that intro-
duces word2vec technique for computing word em-
beddings [27]. The skip-gram is a shallow neural net-
work using the current instruction to predict the instruc-
tions 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 neu-
ral networks, one for each task related to count and type
inference. To achieve each task outlined, we train a re-
current 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 sev-
eral 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 argu-
ments recovery module, we have considered various ar-
chitectures, like a multilayer perceptron (MLP), a con-
volutional 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 percep-
tron (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 net-
work. The recurrent structure of the network allows it to
handle variable-length input sequences naturally.

In order to deal with the exploding and vanishing gra-
dients during training [9], there are few commonly de-
sign 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 cho-
sen RNN cells [43]. This mechanism acts as a stochastic
regularization technique. In our design, we experimented
with various dropout rates between 0.1 to 0.8. We exper-
imentally 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 prepa-
ration 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. Func-
tion boundaries identification with minimal reliance of
principal component analysis (PCA) \cite{19} and classical
can be visualized with scatter plots. Methods such as
t-SNE \cite{25} plots and analogical reasoning of vectors. 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. \cite{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", "queen"), 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.
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. 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 by the maximum L2-norm 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.

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 00 to 03 for both x86 and x64. We obtained the ground truth for the function arguments by parsing the DWARF debug information.

Following the dataset creation procedure used in previous work, 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 (00–03) 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, util-linux, inetutils, diffutils, and usbutils. This dataset contains 5168 different binaries, resulting from compiling each program with 4 optimization levels (00–03) 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 sanitization, 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 set 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 $P_{ci}$ and recall $R_{ci}$ are used to measure the performance of EKLAVYA for class $i$ and are defined as:

$$P_{ci} = \frac{TP_i}{TP_i + FP_i}; R_{ci} = \frac{TP_i}{TP_i + FN_i}$$

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:

$$Acc = \frac{1}{n} \sum_{i=1}^{n} P_i \times R_{ci}$$

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 $R_{ci}$ 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 occurring 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 $r<9$ 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.

<table>
<thead>
<tr>
<th>Arch</th>
<th>Task</th>
<th>Opt.</th>
<th>Metrics</th>
<th>Number of Arguments</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.9113</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.8348</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.8053</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.8391</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.9270</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.6934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.6660</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.6534</td>
</tr>
<tr>
<td>x86</td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.9203</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.8602</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.8380</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.8279</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.9748</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.9270</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.6934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.6660</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.6534</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.9203</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.8602</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.8380</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.8279</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.9748</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.9270</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.6934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Distribution</td>
<td></td>
<td>0.6660</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Precision</td>
<td></td>
<td>0.6534</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Recall</td>
<td></td>
<td>0.9203</td>
</tr>
</tbody>
</table>
Table 2: Evaluation result for argument type recovery from callers and callees for different optimization levels given different architectures. Columns 4-7 report the evaluation result of EKLA VY A on test dataset with different instruction sets ranging from O0 to O3. "-" denotes that the specific metric cannot be calculated.

<table>
<thead>
<tr>
<th>Arch</th>
<th>Task</th>
<th>Opt.</th>
<th>Metrics</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>char</td>
<td>int</td>
<td>pointer</td>
</tr>
<tr>
<td>O0</td>
<td>Data Distribution</td>
<td>0.0015</td>
<td>0.1500</td>
<td>0.0000</td>
<td>0.2926</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.0500</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0500</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.0000</td>
<td>0.1666</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Task3</td>
<td>O1</td>
<td>Data Distribution</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>O2</td>
<td>Data Distribution</td>
<td>0.0015</td>
<td>0.1500</td>
<td>0.0000</td>
<td>0.2926</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.0500</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0500</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.0000</td>
<td>0.1666</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>O3</td>
<td>Data Distribution</td>
<td>0.0015</td>
<td>0.1500</td>
<td>0.0000</td>
<td>0.2926</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.0500</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0500</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.0000</td>
<td>0.1666</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>x64</td>
<td>Data Distribution</td>
<td>0.0015</td>
<td>0.1500</td>
<td>0.0000</td>
<td>0.2926</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>0.0500</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0500</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
<td>0.0000</td>
<td>0.1666</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
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 x86. 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.

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 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* and *mov $constant, %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, %esp, add $0x30, %esp* and the distance between the pair (*sub $0x20, %esp, add $0x20, %esp*) 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Relative Score</th>
<th>Instruction</th>
<th>Relative Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp</td>
<td>0.149406</td>
<td>subq $0x80, %esp</td>
<td>1.000000</td>
</tr>
<tr>
<td>movl %esp, %ebp</td>
<td>0.265591</td>
<td>leaq (%esp), %rdi</td>
<td>0.411254</td>
</tr>
<tr>
<td>pushl %ebx</td>
<td>0.179169</td>
<td>xorl %eax, %eax</td>
<td>0.161366</td>
</tr>
<tr>
<td>subl $0x14, %esp</td>
<td>0.370329</td>
<td>movq %rsi, 0x10(%rbp)</td>
<td>0.923426</td>
</tr>
<tr>
<td>movl $0x%ebp, %eax</td>
<td>1.000000</td>
<td>movq %rcx, %r15</td>
<td>0.214237</td>
</tr>
<tr>
<td>movl %eax, %ecx</td>
<td>0.509958</td>
<td>movq %rcx, %r15</td>
<td>0.149166</td>
</tr>
<tr>
<td>leal 0x090227, %edx</td>
<td>0.372616</td>
<td>movq %rsi, 0x10(%esp)</td>
<td>0.336599</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movq %rdi, 0x20(%esp)</td>
<td>0.255754</td>
</tr>
</tbody>
</table>

(a) “print_without_quoting” compiled with clang and O0 on 32-bit (having 2 arguments)  
(b) “parse_stab_struct_fields” compiled with clang and O0 on 64-bit (having 5 arguments)  
(c) “EmptyTerminal” compiled with clang and O0 on 32-bit (having 0 arguments)  
(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 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

110 26th USENIX Security Symposium USENIX Association
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.

**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 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Relative Score</th>
<th>Instruction</th>
<th>Relative Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl $0x10, %esp</td>
<td>0.297477</td>
<td>subq $0x328, %esp</td>
<td>0.774363</td>
</tr>
<tr>
<td>movq $0x28(%rdx), %rcx</td>
<td>1.000000</td>
<td>movq $0x28(%rdx), %rcx</td>
<td>0.814174</td>
</tr>
<tr>
<td>movl $0x10, %esi</td>
<td>0.332725</td>
<td>movq $9x14, %esi</td>
<td>0.458216</td>
</tr>
<tr>
<td>movq %rsi, %rsi</td>
<td>0.481093</td>
<td>movq %rsi, %rsi</td>
<td>0.36804</td>
</tr>
<tr>
<td>cmpb $1, %cl</td>
<td>0.249921</td>
<td>movq %rdi, %rdi</td>
<td>0.442176</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>movl (%rsp), %r13</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

(a) “bfd_coverage” compiled with gcc-32-O3 (1st argument - float) (b) “do_fprint” compiled with clang-64-01 (2nd argument - pointer)
Table 5: x86 multiple type mispredictions for second arguments.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Relative Score</th>
<th>Instruction</th>
<th>Relative Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl $0x1c, %edi</td>
<td>0.719351</td>
<td>pushl %edi</td>
<td>0.543965</td>
</tr>
<tr>
<td>movl $0x24(,%edi, 4) %ecx</td>
<td>1.000000</td>
<td>movl %edi, %edi</td>
<td>0.395979</td>
</tr>
<tr>
<td>movl $0x10(%edi, 1 (%edi, %ecx))</td>
<td>0.244679</td>
<td>movl %edi, %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>movl $0x20, %ebp</td>
<td>0.021946</td>
<td>pushl %edi</td>
<td>0.068469</td>
</tr>
<tr>
<td>movl $0, 0x1c(%esp)</td>
<td>0.021946</td>
<td>movl %edi, %edi</td>
<td>0.154597</td>
</tr>
<tr>
<td>movl %edx, %edi</td>
<td>0.457177</td>
<td>pushl %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>movl %eax, %ebx</td>
<td>0.260028</td>
<td>movl %edi, %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>pushl %ebx</td>
<td>0.068469</td>
<td>pushl %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>movl %ebx, %ecx</td>
<td>0.418808</td>
<td>movl %edi, %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>addl $0x1c, %esp</td>
<td>0.260028</td>
<td>pushl %edi</td>
<td>0.021946</td>
</tr>
<tr>
<td>pushl %edi</td>
<td>0.021946</td>
<td>movl %edi, %edi</td>
<td>0.021946</td>
</tr>
</tbody>
</table>

(a) “quoting/*plat” compiled with gcc and O1 (true type is char but predicted as int)

Table 6: x64 mispredictions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Relative Score</th>
<th>Instruction</th>
<th>Relative Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %r8d</td>
<td>0.175079</td>
<td>pushl %edi</td>
<td>0.025531</td>
</tr>
<tr>
<td>movq %r8d, %r9d</td>
<td>0.032229</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>callq %r8d</td>
<td>0.000000</td>
<td>pushl %edi</td>
<td>0.025531</td>
</tr>
<tr>
<td>testq %rax, %rax</td>
<td>0.525375</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>je 0x1004</td>
<td>0.579351</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>popq %r8d</td>
<td>0.164043</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>retq</td>
<td>0.315274</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>movq %r8d, %r9d</td>
<td>0.366585</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
<tr>
<td>callq %r8d</td>
<td>0.665486</td>
<td>movq %r8d, %r9d</td>
<td>0.025531</td>
</tr>
</tbody>
</table>

(a) “x64/*pop” compiled with clang and O1 (true type is first argument is pointer but predicted as int)

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 correspond to one of them. 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 5 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 this 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 [22, 5, 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, 5, 30, 20, 36, 15] and condition 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, 55].

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. [25] formulate the usage of different data types in binaries to do type reconstruction. In addition, ElWazeer et al. [4] 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 Valentijn 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


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:

\[
p_{ij} = \frac{\exp\left(-\frac{\|I_i - I_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|I_i - I_k\|^2}{2\sigma_i^2}\right)}
\]

where \( n \) is the number of data points and \( \sigma \) is the variance of distribution which is centered at each data point \( x_i \). Here, t-SNE determines the value of \( \sigma \) 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:

\[
\text{perplexity}(p_i) = e^{H(p_i)}
\]

\[
H(p_i) = -\sum_j p_{ji} \log_2 p_{ji}
\]

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:

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

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 EKLA VYA 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.

<table>
<thead>
<tr>
<th>Arch</th>
<th>Task</th>
<th>O0 Median F1</th>
<th>Max F1</th>
<th>Min F1</th>
<th>Task 1</th>
<th>O1 Median F1</th>
<th>Max F1</th>
<th>Min F1</th>
<th>Task 2</th>
<th>O2 Median F1</th>
<th>Max F1</th>
<th>Min F1</th>
<th>Task 3</th>
<th>O3 Median F1</th>
<th>Max F1</th>
<th>Min F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>Task1</td>
<td>0.978</td>
<td>0.994</td>
<td>0.923</td>
<td>0.983</td>
<td>0.960</td>
<td>0.991</td>
<td>0.925</td>
<td>0.972</td>
<td>0.968</td>
<td>0.997</td>
<td>0.938</td>
<td>0.977</td>
<td>0.967</td>
<td>0.998</td>
<td>0.936</td>
</tr>
<tr>
<td></td>
<td>Task2</td>
<td>0.984</td>
<td>0.993</td>
<td>0.952</td>
<td>0.986</td>
<td>0.965</td>
<td>0.998</td>
<td>0.971</td>
<td>0.967</td>
<td>0.970</td>
<td>0.982</td>
<td>0.948</td>
<td>0.973</td>
<td>0.966</td>
<td>0.984</td>
<td>0.944</td>
</tr>
<tr>
<td></td>
<td>Task3</td>
<td>0.970</td>
<td>0.998</td>
<td>0.919</td>
<td>0.974</td>
<td>0.954</td>
<td>0.990</td>
<td>0.940</td>
<td>0.963</td>
<td>0.950</td>
<td>0.993</td>
<td>0.985</td>
<td>0.993</td>
<td>0.956</td>
<td>0.994</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>x64</td>
<td>Task1</td>
<td>0.983</td>
<td>0.996</td>
<td>0.967</td>
<td>0.985</td>
<td>0.975</td>
<td>0.997</td>
<td>0.873</td>
<td>0.971</td>
<td>0.978</td>
<td>0.997</td>
<td>0.934</td>
<td>0.979</td>
<td>0.977</td>
<td>0.999</td>
</tr>
<tr>
<td></td>
<td>Task2</td>
<td>0.997</td>
<td>0.999</td>
<td>0.975</td>
<td>0.998</td>
<td>0.976</td>
<td>0.998</td>
<td>0.942</td>
<td>0.976</td>
<td>0.980</td>
<td>0.991</td>
<td>0.946</td>
<td>0.979</td>
<td>0.979</td>
<td>0.991</td>
<td>0.950</td>
</tr>
<tr>
<td></td>
<td>Task3</td>
<td>0.984</td>
<td>0.992</td>
<td>0.967</td>
<td>0.984</td>
<td>0.938</td>
<td>0.992</td>
<td>0.871</td>
<td>0.965</td>
<td>0.954</td>
<td>0.993</td>
<td>0.993</td>
<td>0.993</td>
<td>0.956</td>
<td>0.994</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>Task4</td>
<td>0.983</td>
<td>0.981</td>
<td>0.887</td>
<td>0.959</td>
<td>0.984</td>
<td>1.000</td>
<td>0.727</td>
<td>0.990</td>
<td>0.985</td>
<td>1.000</td>
<td>0.804</td>
<td>0.994</td>
<td>0.988</td>
<td>1.000</td>
<td>0.880</td>
</tr>
</tbody>
</table>