Can a Deep Learning Model for One Architecture Be Used for Others? Retargeted-Architecture Binary Code Analysis

Junzhe Wang, *George Mason University;* Matthew Sharp, *University of South Carolina;* Chuxiong Wu, Qiang Zeng, and Lannan Luo, *George Mason University*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junzhe

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Can a Deep Learning Model for One Architecture Be Used for Others? Retargeted-Architecture Binary Code Analysis

Junzhe Wang
George Mason University

Matthew Sharp
University of South Carolina

Chuxiong Wu
George Mason University

Qiang Zeng
George Mason University

Lannan Luo ✉
George Mason University

Abstract

NLP-inspired deep learning for binary code analysis demonstrates notable performance. Considering the diverse Instruction Set Architectures (ISAs) on the market, it is important to be able to analyze code of various ISAs. However, training a deep learning model usually requires a large amount of data, which poses a challenge for certain ISAs such as PowerPC that suffer from the "*data scarcity*" issue. For instance, acquiring a large dataset of PowerPC malware proves to be challenging. Moreover, given a binary analysis task and multiple ISAs, it takes much time and effort (e.g., for data collection, labeling and cleaning, and parameter tuning) to train one model *per* ISA. We propose a new direction, *retargeted-architecture binary code analysis*, to handle the data scarcity issue and alleviate the per-ISA effort. Our idea is to *transfer knowledge from one ISA to others*—that is, a model, trained with rich data and much time and effort for one ISA, can perform prediction for others *without any modification*. We showcase the idea through two important tasks: malware detection and function similarity detection. An extensive evaluation involving four ISAs (x86, ARM, MIPS, and PowerPC) demonstrates the effectiveness of the approach and the high performance is interpreted.

1 Introduction

Given a closed-source program, such as most proprietary software and malware, binary code analysis is indispensable for a variety of tasks, for example, code plagiarism detection [32,45], malware classification [66,79], and vulnerability discovery [22, 61]. Traditional binary code analysis often suffers from being inaccurate or unscalable. For instance, symbolic execution of binary code, such as BitBlaze [9] and BAP [7], is accurate in extracting code semantics, but is slow and does not scale well [45]. As another example, code analysis based on system calls is scalable [72], but it ignores large chunks of other semantic information.

In 2018, `InnerEye` [80] proposed to adapt deep learning techniques developed for natural language processing (NLP) to binary analysis. Since then we have witnessed a surge of NLP-inspired binary analysis [20,21,40,52,60,64]. This direction shows noticeably better performances over traditional methods in accuracy and scalability. However, two notable challenges hinder its wide applications to various ISAs.

Challenge 1: Data Scarcity for Some Binary Analysis Tasks with Certain ISAs. Training a deep learning model usually requires a large amount of data. As a result, for some binary analysis tasks (e.g., malware detection), the rich datasets (e.g., malware) of certain ISAs, such as x86, lead to a disproportionate focus on them and a negligence of other ISAs, such as PowerPC, where few or even no labeled datasets exist (which we call *low-resource* ISAs). Moreover, it is labor-intensive and time-consuming to collect data samples and manually label them to build datasets for low-resource ISAs. Dealing with the data scarcity issue in low-resource ISAs for certain binary analysis tasks is an unresolved challenge.

Challenge 2: Massive Effort to Cover Multiple ISAs. Software is often compiled for various ISAs. For instance, a library is compiled into IoT firmware for different ISAs, which causes a single vulnerability at source-code level to spread across binaries of different ISAs. Thus, it is important to extend binary analysis capacity to multiples ISAs [22,25,61,75,80].

There are over one hundred different ISAs [15,23]. Given a binary analysis task and multiple ISAs, however, it takes massive time and effort (due to data collection, cleaning and labelling) and computing resources (such as parameter tuning) to train a model for every individual ISA.

Our Goal. We consider a new direction, named *retargeted-architecture binary analysis*, to cope with the data scarcity problem and alleviate the per-ISA effort. Our goal is to train a model on one ISA (e.g., x86) and reuse it for other ISAs (e.g., PowerPC and MIPS), *without any modification*. Through this, we can greatly save the computing resources for training a large number of models and the effort in collecting datasets for each ISA, especially for low-resource ISAs.

Our Approach. A binary, after being disassembled, is expressed in an assembly language. Given this insight,

InnerEye approaches binary code analysis by adopting deep learning techniques for NLP [80]. Our work focuses on such NLP-inspired binary analysis, which has gained much attention because of their exceptional performance. We aim at developing techniques that enable the reuse of an NLP-based model for multiple ISAs. In NLP-based models [20, 21, 80], instructions are considered as words and represented as embeddings to facilitate further processing.

To enable retargeted-architecture binary analysis, we propose *multi-architecture instruction embeddings* (MAIE), where similar instructions, regardless of their ISAs, have close embeddings in a shared vector space. Equipped with such a shared space, we can *transfer knowledge from one ISA to other ISAs.* As a result, a model trained only using data of one ISA (e.g., x86) simultaneously obtains the capability to perform prediction for others (such as PowerPC and MIPS). The reason this is feasible is that *MAIE bridges divergences of different ISAs*, as illustrated in the evaluation.

We design an *unsupervised* method for learning MAIE, which contains two main steps. First, we train instruction embeddings for each ISA *separately*, which we call *mono-architecture instruction embeddings* (OAIE). Then, we *map* OAIE of different ISAs to a shared space via a transformation to learn MAIE. The key question is thus how to compute the transformation. Our insight is that, given the similarity matrices (detailed in Section 5.2) of OAIE, two semantically similar instructions of different ISAs tend to (no need to always) have similar distributions of similarity values. Based on this, we can induce an initial instruction mapping and iteratively improve it until convergence.

Results. We have implemented the novel unsupervised method for learning MAIE, named UniMap, and evaluated it on four ISAs: x86, ARM, MIPS, and PowerPC (PPC).[1] As we aim to reuse a model trained on a data-rich ISA for other ISAs, we transfer knowledge from x86 to the other three ISAs. The effectiveness of knowledge transfer in a different direction requires further investigation. (1) In the *intrinsic evaluation*, we conduct the instruction similarity task to evaluate the *quality* of MAIE—whether they capture the semantic information of instructions across ISAs. (2) In the extrinsic evaluation, we conduct two critical downstream tasks to evaluate the *transferability* of MAIE. For the malware detection task, when a model trained on x86 is then reused for ARM, MIPS, and PPC, the accuracy only *decreases by 2%, 4%, and 5%*, respectively, compared to that for x86. For the function similarity detection task, the accuracy *only decreases by 3%, 4%, and 3%*. The results show that UniMap can effectively generate high-quality MAIE with good transferability. Below are our contributions:

- We propose the direction of *retargeted-architecture binary code analysis* to alleviate the *per*-ISA effort and cope with the data scarcity issue. Given a task, one model of a data-rich ISA is trained and then reused for others.
- To enable retargeted-architecture binary code analysis, we propose *multi-architecture instruction embeddings* (MAIE), such that semantically similar instructions, *regardless of their ISAs*, have close embeddings. We design an unsupervised method for learning MAIE. The learning of MAIE is a one-time effort: once learned, MAIE can be reused in various binary analysis tasks.
- We have implemented the system[2] and conducted extensive evaluations on four ISAs: x86, ARM, MIPS and PPC. The results demonstrate the effectiveness of the approach and the high performance is interpreted.

[1] We are aware that ARM does not have the data scarcity issue. Given the importance of ARM, our evaluation involves it.

2 Related Work

2.1 Traditional Code Analysis

Mono-architecture. Most traditional approaches work on a *single* ISA. Some are source code based [34, 37, 41, 43, 70]. Others analyze binary code [44, 76–78], e.g., using symbolic execution [46], but are quite expensive [13, 48–50]. Dynamic approaches include API birthmark [11], system call birthmark [71], and core-value birthmark [33]. *Extending them to other ISAs would be hard*; code coverage is another challenge.

Cross-architecture. Recent works have applied traditional approaches to the *cross-architecture* scenario [12, 17, 18, 22, 61]. Multi-MH and Multi-k-MH [61] are the first two for comparing functions in different ISAs, but the fuzzing-based basic-block similarity comparison is expensive. discovRE [22] uses pre-filtering to boost the matching process. Esh [17] compares blocks using a SMT solver and is unscalable. GitZ [18] lifts binaries to IR to create function signatures.

2.2 Machine/Deep Learning-based Analysis

Mono-architecture. Recent research has applied machine/deep learning to code analysis [14, 20, 30, 39, 40, 47, 59, 68]. Asm2Vec [20] uses the Paragraph Vector model [38] to generate a vector for each function. Transformer-based models have been applied to analyze binary code. As an example, PalmTree [40] trains the BERT model [19] to generate embeddings for each token of an instruction. However, these works only focus on a *single* ISA.

Cross-architecture. Existing works explore *pairwise*-architecture binary analysis [12, 25, 75, 80], where a pair of ISAs is studied. InnerEye [80] is the first that proposes to use deep learning techniques developed for NLP to binary analysis. It considers instructions as words and basic blocks as sentences, and gets inspired by neural machine translation to compare code across ISAs. Also for cross-ISA code comparison, TREX [60] uses execution traces of functions from

[2] https://github.com/lannan/UniMap

different ISAs to pretrain a transformer model, and then fine-tunes the pretrained model using the semantically similar function pairs. The fine tuning step is impeded by the data scarcity issue. Moreover, *model reuse is not their goal*. In contrast, our retargeted-architecture binary analysis focuses on reusing a model trained on one ISA for other ISAs.

The prior work [64] is the most related one. There are three major differences. (1) We aim to learn *multi*-architecture instruction embeddings (MAIE), which work for multiple ISAs, while the prior work learns *pairwise*-architecture instruction embeddings, which only work for a pair of ISAs. (2) The prior work is based on *supervised* learning and needs equivalent instruction sequences for training; ours is based on *unsupervised* learning. (3) More importantly, our goal is to demonstrate, given a binary analysis task, how a model trained for one ISA can be used for others, which is not studied in the prior work.

2.3 Statistics of Targeted ISAs

We analyzed the machine/deep learning-based binary analysis works published between 2010 and December 2022. We only consider those targeting compiled languages (i.e., C/C++), but exclude that on (i) interpreted languages (like JavaScript, PHP, and Python) and (ii) Java, where the bytecode compiled from the same source code is identical, regardless of ISAs. There are 234 papers, where 197 focus on a single architecture and 37 on cross-architecture. Plus, 83% of mono-architecture approaches target x86, and cross-architecture ones are mostly limited to x86 and ARM. This calls for research to cover more ISAs, including less-discussed ones. Powered by MAIE, our work covers four ISAs (x86, ARM, MIPS, and PPC) without tedious effort and the prediction accuracies keep high.

3 Background

3.1 Word Embeddings

Many NLP tasks use word embeddings, which capture the contextual semantic meaning of words. The Skip-gram model [55] is a popular model to learn word embeddings that are good at predicting the surrounding words of a current word. The training is *unsupervised*. Once trained, semantically similar words have close embeddings.

3.2 Multi/Cross-lingual Word Embeddings

A wide variety of multi/cross-lingual NLP tasks [8, 10, 27, 65] have motivated recent work in training *multi/cross-lingual word embeddings*, where similar words in different human languages have close embeddings in a shared space.

Existing methods can be classified into two categories. (1) *Joint methods* simultaneously learn cross-lingual word embeddings in different languages using *parallel* corpora [26, 36, 53]. The prior work [64] is based on jointly learning. (2) *Mapping methods* first independently train word embeddings in each language, and then map them to a shared space through transformations [6, 16, 56]. *Mapping methods* have the advantage of requiring little or no cross-lingual supervision. Inspired by advances made by the second category, we seek the possibility of obtaining a shared space for instructions of different ISAs.

4 Overview

4.1 Types of Datasets and ISAs

We first define two types of datasets as below.

- ***General Datasets*** can be obtained via cross-compilation. Specifically, we collect opensource programs, and compile them for different ISAs using cross compilers. Given the wide availability of opensource code, this requires little effort. General datasets are used for training MAIE.
- ***Task-Specific Datasets*** are related to binary analysis tasks, such as malware for the classification task, and IoT firmware for the vulnerability discovery task. Given a binary analysis task, there may exist rich *task-specific* datasets for certain ISAs, which we call ***high-resource ISAs***, while for other ISAs, there may be few or no *task-specific* datasets, which we call ***low-resource ISAs***.

Clarifications. We clarify that (1) *data scarcity is an issue regarding task-specific datasets*, rather than general datasets used to train MAIE, and (2) some (but not all) binary analysis tasks are affected by the data scarcity issue. As an example, for the task of malware detection, gathering sufficient data (e.g., malware) poses challenges for certain ISAs (like PowerPC). Moreover, as the source code of malware is usually unavailable, the cross-compilation method that generates binaries across ISAs from source code does not work. Thus, such tasks suffer from the data scarcity issue.

4.2 Motivation

As there are diverse ISAs on the market and software is often compiled for various ISAs, it is critical to extend binary analysis capacity to multiples ISAs. Given a binary analysis task and multiple ISAs, however, it takes massive effort and computing resources to train a model *per* ISA. Not to mention sometimes it is difficult to collect sufficient data in low-resource ISAs for certain binary analysis tasks.

We propose a new direction, named *retargeted-architecture binary code analysis*, to alleviate the per-ISA effort and cope with the data scarcity issue. Figure 1 compares our approach with mono-architecture and pairwise-architecture approaches.

(I) ***Mono-architecture approach:*** Given a binary analysis task, as shown Figure 1(a), the mono-architecture approach trains a model using task-specific data for one ISA and also tests on that ISA. As a result, for n ISAs, it needs to collect n task-specific datasets and train n models. Moreover,

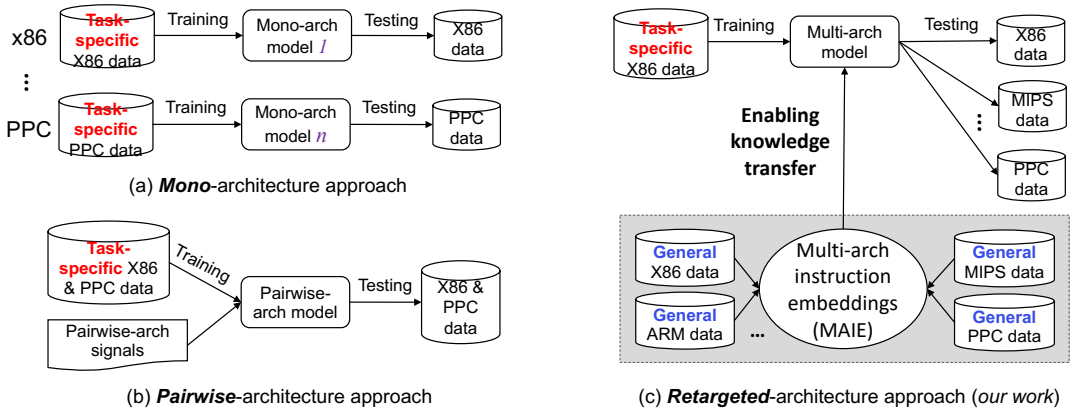


Figure 1: Comparison between mono-, pairwise- and retargeted-architecture approaches. (a) Mono-architecture approach trains a model for *each* ISA. (b) Pairwise-architecture approach trains a model for *each pair* of ISAs. (c) Our retargeted-architecture approach only trains a model on one ISA and reuses it for other ISAs.

for certain binary analysis tasks, such as malware detection, collecting annotated malware samples is not easy for low-resource ISAs, causing the difficulty in training a model that can detect malware of low-resource ISAs.

(II) ***Pairwise-architecture approach:*** As shown in Figure 1(b), the pairwise-arch approach [12, 25, 75, 80] studies the relation of the code of a pair of ISAs. They need task-specific data of each pair of ISAs for training. Plus, to cover n ISAs, it still needs to train many models.

(III) ***Retargeted-architecture approach:*** In contrast, as illustrated in Figure 1(c), our retargeted-architecture approach only needs to train ***one*** model on one ISA (e.g., x86), and ***reuses*** it for other ISAs (e.g., PowerPC), greatly saving the effort in collecting task-specific data for each ISA (especially for low-resource ISAs) and training many models.

4.3 Observations and Interpretations

Our work aim at developing techniques that enable the reuse of NLP-based models for multiple ISAs. These models regard *instructions as words* and represent instructions as embeddings. We first seek to understand why a model trained using the mono-architecture approach cannot test on other ISAs. We thus trained instruction embeddings for four ISAs, x86, ARM, MIPS, and PPC, *separately*, which we call *mono-architecture instruction embeddings* (OAIE), and visualized them in 3D spaces, as shown in Figure 2(a). We can observe that OAIE of different ISAs are separated in different clusters, which can be attributed to syntactic variations among ISAs. As a result, for a binary analysis model whose input layer encodes each instruction as an instruction embedding, if trained on one ISA, it cannot be directly used to test on other ISAs.

4.3.1 Why Not IR?

Intermediate representation (IR) can be used to represent code of different ISAs. For example, VEX IR [4] is an architecture-agnostic and side-effect-free representation that can represent instruction sets of different ISAs in a uniform style. Thus, this raises a question whether we can achieve retargeted-architecture binary analysis by lifting binaries to IR.

Based on our investigation and experiments, we find that: given two binaries of different ISAs, compiled from the **same** piece of source code, after we lift them in a common IR, the resulting IR code looks very different—e.g., the lengths of their IR statements and the types of IR statements differ from each other (see Figure 1 and Figure 3 of [61]; more detailed examples are discussed in Appendix A.1).

Therefore, existing works that leverage IR for analyzing binaries across ISAs have to perform further *advanced* analysis on the IR code [18,51,52,61]. For example, (1) Multi-MH [61] uses fuzzing to detect whether two pieces of VEX IR code are similar. (2) GitZ [18] conducts complex re-optimization on IR code to compare function similarity. (3) GeneDiff [51] applies deep learning analysis to VEX IR code for cross-architecture binary clone detection. Thus, IR is not magic and a "bridge" (MAIE in our work) is needed for enabling a model trained for one ISA to be reused for other ISAs.

4.3.2 Our Solution

We propose to learn *multi-architecture instruction embeddings* (MAIE), where similar instructions, *regardless of their ISAs*, have close embeddings in a shared space. Such a shared space can enable the *transfer of knowledge from one ISA to others*. Fig. 3(a) shows the visualization of MAIE for four ISAs. A quick inspection shows that MAIE cluster together (unlike OAIE which appear in separated clusters in Fig. 2(a)).

To learn MAIE, our main idea is to first *independently train* OAIE for each ISA (which is a resolved problem [20, 80] and can be trained using *general data*), and then *map* them to a shared space via a linear transformation. For this to work, embedding spaces of different ISAs should have

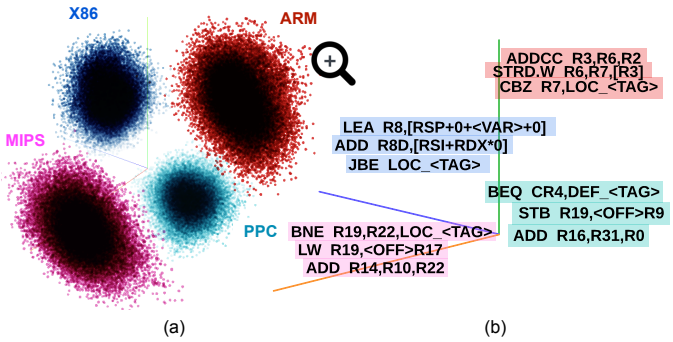


Figure 2: Visualization of all OAIE. Similar instructions have similar geometric arrangements, suggesting that it is possible to learn a linear transformation from one space to another.

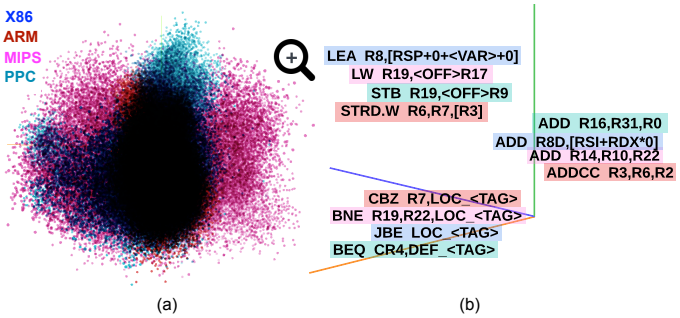


Figure 3: Visualization of MAIE. After transformation, similar instructions, *regardless of their ISAs*, have close embeddings in a shared space. (Section 7 presents an in-depth evaluation and more examples.)

approximately the same structure (i.e., isomorphic; otherwise, it would be hopeless to find a linear transformation). Although this is found true for word embeddings in NLP [56], it is unclear whether this also holds for instruction embeddings.

Visualization Based Interpretations. To answer this question, we zoom into Figure 2(a), and randomly select three x86 instructions and their semantically-similar counterparts from the other three ISAs, which are plot in Figure 2(b). Instructions appear nearby when the Euclidean distances between their embeddings are small. In Figure 2(a), instructions from the same ISAs have the same color (e.g., the instructions with blue are from x86). As we visualize OAIE, we notice that similar instructions across different ISAs exhibit *similar geometric arrangements* (e.g., distances to other instructions).

We seek to understand the reason. The datasets used to train OAIE of different ISAs stem from the same source code, and thus share the same semantics. As a result, *the context of an x86 instruction tends to be similar to that of its counterparts in other ISAs*. As we adopted the Skip-gram model to train OAIE, which takes each word as input and predicts the context for the word, it causes a strong similarity between the vector spaces for different ISAs. Thus, it is promising to learn a linear transformation (e.g., rotation and scaling) to capture the relation between vector spaces representing different ISAs.

Figure 3(b) visualizes the three sets of similar instruction after linear transformation (how to conduct the transformation is one of the research questions of this work). We can observe that after transformation, similar instructions, *regardless of their ISAs*, are close. E.g., the set of instructions related to the `ADD` operation (from different ISAs) are close to each other.

5 Model Design

This section presents our model design, including two steps: (1) learning OAIE of a single ISA, and (2) mapping OAIE of different ISAs to a shared space for learning MAIE.

5.1 Learning OAIE

To learn OAIE of a given ISA, we first build the training dataset (i.e., *general data*; see Section 4.1). As a basic block is a straight-line of instructions, we regard basic blocks as sentences and build a dataset containing a large number of basic blocks. To mitigate the OOV (Out-Of-Vocabulary) problem, we normalize the instructions by applying two types of rules.

Common Rules for All ISAs. The first type includes common rules that are applied to all ISAs.

- (R1) Number constants are replaced with 0, and minus signs are preserved.
- (R2) String literals are replaced with `<STR>`.
- (R3) Function names are replaced with `<FOO>`.
- (R4) Stack variables (with prefix `var_`) are replaced with `<VAR>`, and arguments (with prefix `arg_`) with `<ARG>`.
- (R6) We use IDA Pro [67] to disassemble binaries, which generates dummy names [31]. A dummy name consists of a type-dependent prefix and a suffix which is usually address-dependent. E.g., i) `locret_` represents a return instruction; ii) `word_`, `dword_`, and `qword_` represent data with different lengths; iii) `flt_` represents floating point data. We replace each dummy name with its prefix.
- (R7) Other symbols are replaced with `<TAG>`.

ISA-specific Rules. As the assembly languages of different ISAs have varying grammar peculiarities, we also apply some ISA-specific rules.

- (S1) ARM uses *curly braces* to indicate the operand is *a list of registers* [1]. E.g., `PUSH {R4, R5, R6}` means to push registers in the order of `R6`, `R5` and `R4` to the stack. As *curly braces* are unique to ARM and other ISAs do not use it, we expand an ARM instruction containing a curly brace to a sequence of instructions. For the example, `PUSH {R4, R5, R6}` is expanded to three instructions: `PUSH R6`, `PUSH R5`, and `PUSH R4`.

Table 1: Vocabulary (i.e., set of unique instructions) sizes.

ISA	*Before normalization*	*After normalization*
x86	3,330,240	39,311
ARM	3,436,450	44,903
MIPS	3,406,282	40,467
PPC	3,513,131	49,486

- (S2) Some registers in ARM are aliases of others [2]. E.g., `SP` is an alias of `R13` (stack pointer); `LR` is an alias of `R14` (link register); `PC` is an alias of `R15` (program counter). We remove all aliases and replace `SP`, `LR`, and `PC` with `R13`, `R14`, and `R15`, respectively.
- (S3) Some registers in MIPS are aliases of others [3]. For example, `GP` is an alias of `R28` (global pointer), `SP` is an alias of `R29` (stack pointer), `FP` is an alias of `R30` (frame pointer), and `RA` is an alias of `R31` (return address). Similarly, we also remove all alias names.
- (S4) MIPS uses the `$zero` register to hold the constant 0. We replace it with 0.

Table 1 shows the vocabulary sizes for each ISA before and after normalization. There are two observations. (1) First, after normalization, the vocabulary sizes decrease dramatically (e.g., the vocabulary size for x86 decreases from $3,330,240$ to $39,311$). (2) More importantly, before normalization, there is a big gap of the vocabulary sizes among the four ISAs. E.g., the gap between x86 and ARM is $106,210$ ($= 3,436,450 - 3,330,240$). After normalization, the gap is reduced: e.g., the gap between x86 and ARM is reduced to $5,592$.

Reducing the gap of vocabulary sizes is an important step that makes learning MAIE possible. In order to learn a linear transformation that maps OAIE of different ISAs into a shared space, these OAIE need to have approximately the same geometric structure (see Section 4.3), which naturally requires they have relatively similar sizes. Therefore, the instruction normalization not only resolves the OOV problem, but is also essential for our approach to work. We acknowledge that determining the maximum gap between vocabulary sizes of different ISAs that allows this approach to work is an interesting question. We leave it for future work.

We adopt Skip-gram in `fastText` [24] to learn OAIE. The result is an embedding matrix $\mathbf{X} \in \mathbb{R}^{V \times d}$ for *each* ISA, where V is the number of unique instructions in the vocabulary, and d the dimensionality of the instruction embedding.

5.2 Learning MAIE

Given N ISAs, one is chosen as the target (e.g. x86), and others as the sources. We learn $N-1$ transformations (one for each of the $N-1$ ISAs) to map the corresponding OAIE to the space of the target ISA. Below, we use two ISAs to present our approach, which can be easily extended to multiple ones.

Let $\mathbf{X}$ and $\mathbf{Y}$ be the matrices storing OAIE of two ISAs, respectively, where $\mathbf{X}$ is the source and $\mathbf{Y}$ the target. The ith row $\mathbf{X}_{i*}$ and $\mathbf{Y}_{i*}$ are the embeddings of the ith instruction in their respective vocabularies. As $\mathbf{X}$ and $\mathbf{Y}$ are not aligned according to the vocabularies (i.e., the instructions corresponding to $\mathbf{X}_{i*}$ and $\mathbf{Y}_{i*}$ are not similar), we build a dictionary, denoted as a sparse matrix $\mathbf{D}$, where $\mathbf{D}_{ij} = 1$ if the ith instruction in the source vocabulary is similar to the jth instruction in the target vocabulary. Our goal is to find a transformation matrix $\mathbf{T}$ such that the sum of squared Euclidean distances between the mapped embeddings $\mathbf{X}_{i*}\mathbf{T}$ and the target embeddings $\mathbf{Y}_{j*}$ for the dictionary entries $\mathbf{D}_{ij}$ is minimized, as shown below:

$$\mathbf{T} = \underset{\mathbf{T}}{\texttt{arg min}} \sum_i \sum_j \mathbf{D}_{ij} \|\mathbf{X}_{i*}\mathbf{T} - \mathbf{Y}_{j*}\|^2 \quad (1)$$

To solve the objective function, we use $\mathbf{D}$ to learn $\mathbf{T}$, and in turn, use $\mathbf{T}$ to induce a new $\mathbf{D}$ iteratively until convergence. Figure 4 shows the workflow. The instructions colored in red and green are from two ISAs and projected in two spaces. We first obtain the initial dictionary $\mathbf{D}$, and learn $\mathbf{T}$ using $\mathbf{D}$ and rotate $\mathbf{X}$ accordingly so that $\mathbf{XT}$ and $\mathbf{Y}$ are in the same space. We then induce a better $\mathbf{D}$ using nearest neighbors. This process is iterated until convergence, resulting in close embeddings between $\mathbf{XT}$ and $\mathbf{Y}$. The outputs are MAIE, containing both $\mathbf{XT}$ and $\mathbf{Y}$. Below we present three steps: 1) normalizing OAIE, 2) learning an initial dictionary $\mathbf{D}$, and 3) iteratively improving $\mathbf{D}$ and learning $\mathbf{T}$.

5.2.1 Embedding Normalization

We first normalize OAIE to be unit vectors and mean center each dimension, such that training instances contribute equally to the optimization goal. Then, Equation 1 is equivalent to maximizing the sum of cosine similarities for $\mathbf{X}_{i*}\mathbf{T}$ and $\mathbf{Y}_{j*}$:

$$\mathbf{T} = \underset{\mathbf{T}}{\texttt{arg max}} \sum_i \sum_j \mathbf{D}_{ij} \cos(\mathbf{X}_{i*}\mathbf{T}, \mathbf{Y}_{j*}) \quad (2)$$

5.2.2 Learning Initial Dictionary

As $\mathbf{X}$ and $\mathbf{Y}$ are not aligned (i.e., the instructions corresponding to the ith row $\mathbf{X}_i$ and $\mathbf{Y}_i$ are not similar), we need to build a reasonable initial dictionary $\mathbf{D}$, rather than using a random one which may result in poor local optima.

Observation. We notice that *two similar instructions in different ISAs tend to have similar distributions of similarity values*. Assume $\mathbf{X}$ and $\mathbf{Y}$ store OAIE of two ISAs, respectively. We calculate their similarity matrices: $\mathbf{S}^X = \mathbf{X}\mathbf{X}^T$ and $\mathbf{S}^Y = \mathbf{Y}\mathbf{Y}^T$. The i-th row $\mathbf{S}^X_{i*}$ (or $\mathbf{S}^Y_{i*}$) contains the similarity values of the ith instruction with all instructions in its respective vocabulary (note that OAIE have been normalized to be unit vectors). As an example, Figure 5 shows the distributions of similarity values of two x86 instructions, `ADD RSP,0` and `CALL FOO`, and one ARM instruction, `ADD SP,SP,0`. We can see that a pair of similar x86 and ARM instructions, `ADD RSP,0` and `ADD SP,SP,0`, have more similar distributions than a dissimilar pair, `ADD SP,SP,0` and `CALL FOO`.

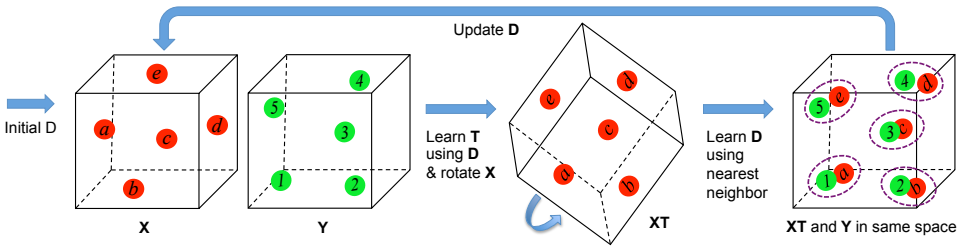
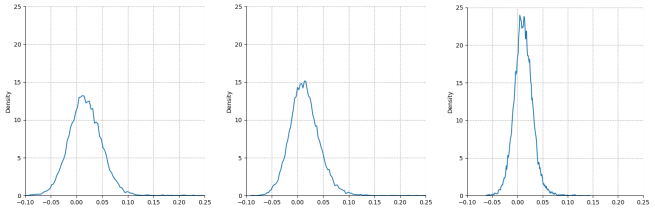


Figure 4: Workflow of UNIMAP. It uses the dictionary **D** to learn the transformation matrix **T**, and in turn, uses the transformation matrix **T** to learn the dictionary **D** until convergence.



(a) x86: `ADD RSP,0` (b) ARM: `ADD SP,SP,0` (c) x86: `CALL FOO`

Figure 5: Distributions of similarity values of three preprocessed instructions. Semantically similar x86 and ARM instructions (`ADD RSP,0` and `ADD SP,SP,0`) have more similar distributions than dissimilar ones (`CALL FOO` and `ADD SP,SP,0`).

Solution. Based on the observation, we can induce the initial dictionary **D**. Specifically, for a given instruction, we can find its initially-mapped counterpart in another ISA based on whether their distributions of similarity values are most similar. We first sort the values in each row of $\mathbf{S}^X$ and $\mathbf{S}^Y$ *independently*, resulting in matrices $\texttt{sorted}(\mathbf{S}^X)$ and $\texttt{sorted}(\mathbf{S}^Y)$. Then, for each instruction in $\texttt{sorted}(\mathbf{S}^X)$, we apply nearest neighbor retrieval over all rows of $\texttt{sorted}(\mathbf{S}^Y)$ to find its similar one. Finally, we assign $\mathbf{D}_{ij} = 1$ if the ith instruction of **X** is similar to the jth instruction of **Y**. The top 10% frequent instructions are selected to build the initial dictionary.

5.2.3 Learning D and T

We use the initial dictionary **D** to learn **T**, and in turn, use **T** to induce a new **D**. The process is repeated until convergence.

Learning T. Given the dictionary **D**, we learn **T** by solving Equation 2 using stochastic gradient descent [35].

Learning D. Given **T**, we first compute the similarity matrix **S** between **XT** and **Y** using Cross-domain Similarity Local Scaling (CSLS) [16]. CSLS resolves the hubness problem of nearest neighbors (i.e., a few points, known as *hubs*, are nearest neighbors of many others). Given two vectors x and y, CSLS first computes $k^Y(x)$ and $k^X(y)$, which are the average cosine similarities of x and y for their k nearest neighbors in another ISA, respectively. Then, the similarity value between x and y is calculated as $\texttt{CSLS}(x,y) = 2\texttt{cos}(x,y) - k^Y(x) - k^X(y)$.

We set k = 10. We keep some similarity values in **S** with the probability p and set others to 0. The smaller the p is, the more the induced dictionary varies from iteration to iteration, thus enabling to escape poor local optima. The modified **S** is used to induce **D**: for each instruction in **Y** (*resp.* **X**), we find its most similar one in **X** (*resp.* **Y**) using CSLS and set the corresponding element in **D** to 1.

Convergence Criterion. For each iteration i, we compute the distance d_i between **XT** and **Y** using Equation 2. In the first iteration, we set the distance as the best one d_{best}. If the current distance d_i is lower than d_{best} over a threshold, we set d_i as d_{best} and store the current iteration number i. If the distance is not reduced over 50 iterations, the probability p is increased to $2p$. This process is repeated until p reaches 1. As we keep tracking the best objective (i.e., the smallest distance) when increasing p, we obtain the highest one when p reaches 1 (i.e., the procedure is converged), and the resulting **T** makes **XT** and **Y** most similar.

6 Retargeted-Architecture Binary Analysis

To achieve retargeted-architecture binary analysis, our idea is to enable knowledge transfer from one ISA to others, such that a model trained with rich data of one ISA can perform prediction on multiple ISAs, which we call a multi-architecture model. Our work targets NLP-based binary analysis approaches and aims at developing techniques that facilitate obtaining multi-architecture models for such approaches.

Figure 6 shows how to achieve retargeted-architecture binary analysis by applying MAIE. A critical step is to *integrate our MAIE into the input layer* of a mono-architecture model (trained and tested on the *same* ISA), in order to derive a multi-architecture model (trained on one ISA and reused for other ISAs), as illustrated in Figure 6(a) and (b). This enables the input layer to represent each instruction as its MAIE. Through this, we can transform the mono-architecture model into a multi-architecture model. As a result, the model can be trained only using data in one ISA (e.g., x86), and reused, *without any modification*, to perform prediction on data in other ISAs, as shown in Figure 6(c).

This is viable because *the input layer of the multi-architecture model represents each instruction as its MAIE*,

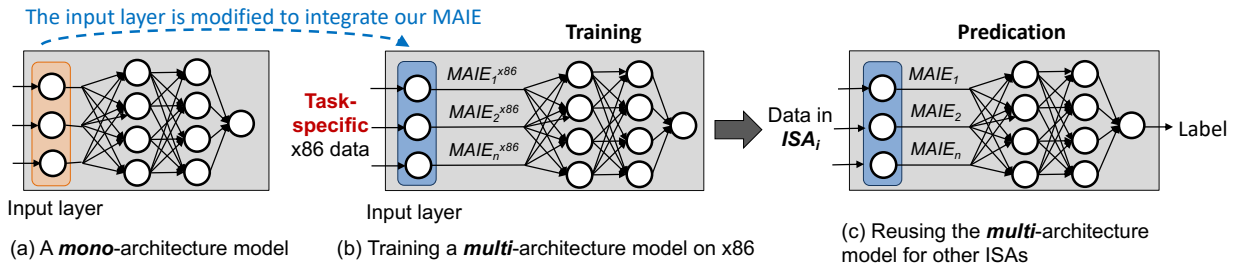


Figure 6: Making use of MAIE to achieve retargeted-architecture binary analysis.

which captures semantic relationships of instructions across ISAs, such that similar instructions, *regardless of their ISAs*, have close embedding (as illustrated in our evaluation).

7 Evaluation

We first describe the evaluation methodology (Section 7.1) and experimental settings (Section 7.2). We then examine the *quality* of MAIE by conducting the *intrinsic* evaluation (Section 7.3). Next, we evaluate the *transferability* of MAIE by conducting the *extrinsic* evaluation, including the malware detection (Section 7.4) and function similarity detection task (Section 7.5). For both evaluations, we also compare our approach with prior work. Finally, we evaluate the model *efficiency* (Section 7.6) and study hyperparameters (Section 7.7).

7.1 Evaluation Methodology

Intrinsic Evaluation. We conduct the instruction similarity task (Section 7.3). The goal is to evaluate the ***qualify*** of MAIE—whether they tolerate the architecture differences and capture semantic information of instructions across ISAs.

Extrinsic Evaluation. We conduct two downstream tasks: the malware detection (Section 7.4) and function similarity detection task (Section 7.5). We aim to evaluate the ***transferability*** of MAIE—whether the knowledge learned from one ISA can be transferred to other ISAs. Specifically, for each task, we train a model on x86, and compute how much the model's accuracy is decreased when tested on ARM, MIPS, and PPC, respectively. *If the decrease is small, we consider MAIE has good transferability.* However, we clarify that it needs further studies and testing to understand the effectiveness of the approach in other binary analysis tasks.

7.2 Experimental Settings

We implemented UNIMAP in Python using NumPy and CuPy. We set the embedding dimension as 200. The starting probability is set to 0.1 and the threshold to 0.000001 (Section 5.2.3). All the experiments were conducted on a computer with a 64-bit 2.50 GHz Intel® Core(TM) i7 CPU, a Nvidia GeForce RTX 3080, 64 GB RAM, and 2 TB HD.

General Datasets for Learning MAIE. We consider four ISAs: x86-64, ARM, MIPS, and PPC. We first build a general dataset for each ISA via cross-compilation (Section 4.2). We collect a set of programs, including *OpenSSL-1.1.1p*, *Binutils-2.35.2*, *Findutils-4.6.0*, and *Libgpg-error-1.45*, and compiled each one with different optimization levels (O0-O3). We use IDA Pro [67] to disassemble them to get the assembly code.

The general dataset of each ISA contains a set of basic blocks in this ISA. In total, we obtain 5,154,010 x86 blocks, 4,641,452 ARM blocks, 5,509,818 MIPS blocks, and 5,094,218 PPC blocks. Our vocabulary contains 39,311 distinct x86 instructions, 44,903 ARM instructions, 40,467 MIPS instructions, and 49,486 PPC instructions (see Table 1).

In NLP, it is widely recognized that a comprehensive dataset, which ensures that the vocabulary covers a wide range of words, is essential for training effective word embeddings. We also access the adequacy of our general datasets. Specifically, we study the vocabulary growth as we incrementally include programs. We find that including *OpenSSL-1.1.1p* results in a vocabulary size of 29,222 for x86. The size increases to 37,235 (a growth of 27%) when *Binutils-2.35.2* is added, and then increases to 39,218 (a growth of 5%) and 39,311 (a growth of 0.2%) when *Findutils-4.6.0* and *Libgpg-error-1.45* are included, respectively. The growth trend is similar for other ISAs. It shows that the vocabulary barely grows in the end when more programs are added. According to the vocabulary growth trend as well as the high performance achieved, our general datasets are adequate to cover instructions and enable effective learning of MAIE.

It is worth noting that, in our evaluation, the general datasets used for training MAIE have no overlap with the testing datasets in the downstream tasks, the details of which are introduced in Sections 7.4 and 7.5.

We consider x86 as the target ISA, which is arguably the most high-resource ISA, and three other ISAs (ARM, MIPS, and PPC) as the source ISAs. We clarify that it needs careful testing before ascertaining the generalizability of this approach to other target and source ISAs. We first learn OAIE of each ISA *separately*, and then train a model to map each OAIE to the x86 space for learning MAIE. Note that learning MAIE is a one-time effort: once MAIE are learned, they can be reused in different downstream tasks.

Table 2: Nearest neighbor instructions *cross-architecturally* as measured by cosine similarity of instruction embeddings. The top-two similar ARM, MIPS, and PPC instructions are shown for each of nine high-frequency x86 instructions.

	ADD R8D,[RSI+RDX*0]	*Score*	**SUB R13,RBP**	*Score*	**CMP R12W,AX**	*Score*
ARM	ADDCC R3,R6,R2	0.32821	SUBS R5,R6,0	0.52357	CMP R8,R0	0.39722
	ADDCC R1,R2,R3	0.32304	SUBS R5,R6,R5	0.46674	CMP R8,R2	0.36116
MIPS	ADD R14,R10,R22	0.39065	SUB R2,R4,R3	0.43441	SLT R3,R2,R15	0.45835
	ADD R14,R10,R23	0.36297	SUB R3,R4,R3	0.42120	SLT R3,R2,R12	0.44797
PPC	ADD R16,R31,R0	0.38367	ADDI R10,R28,0	0.51422	CMPW CR7,R19,R24	0.34893
	ADD R17,R17,R0	0.38328	ADDIC R10,R28,0	0.46657	CMPW CR7,R19,R27	0.32845
	AND BPL,AL	*Score*	**XOR R11D,EDI**	*Score*	**MOV R12,[R14-0]**	*Score*
ARM	AND.W R7,R7,R0	0.31910	ORR.W R4,R1,R7,LSL0	0.48886	MOVCS R5,0	0.40010
	AND.W R3,R7,R0	0.28989	ORR.W R1,R1,R7,LSR0	0.483976	MOVCS R11,R7	0.37647
MIPS	AND R7,R22,R4	0.40560	OR R22,R23,R22	0.40146	MOV R2,R3,R9	0.41806
	SUB R4,R22,R4	0.38260	OR R22,R14	0.38174	MOV R5,R3,R9	0.40347
PPC	AND R18,R10,R7	0.30094	OR R6,R8,R10	0.30091	MR R4,R26	0.35644
	ADD R7,R14,R11	0.30094	ANDI. R0,R7,0	0.30634	MR R7,R26	0.29521
	JBE LOC_<TAG>	*Score*	**PUSH RBP**	*Score*	**LEA R8,[RSP+0+<VAR>+0]**	*Score*
ARM	CBZ R7,LOC_<TAG>	0.40610	PUSH R6	0.55333	STRD.W R6,R7,[R3]	0.62602
	CBZ R7,LOCRET_<TAG>	0.39407	PUSH LR	0.55142	STRD.W R6,R7,[R2]	0.60275
MIPS	BNE R19,R22,LOC_<TAG>	0.28085	SW R28,0	0.45103	LW R19,<OFF>R17	0.56656
	BEQ R19,R22,LOC_<TAG>	0.27817	SW R31,0	0.42857	LH R19,<OFF>R17	0.55023
PPC	BEQ CR4,DEF_<TAG>	0.37506	STW R0,0	0.42499	STB R19,<OFF>R29	0.56327
	BL DEF_<TAG>	0.35735	STWU R1,<OFF>R1	0.40904	LWZ R18,<OFF>R29	0.51616

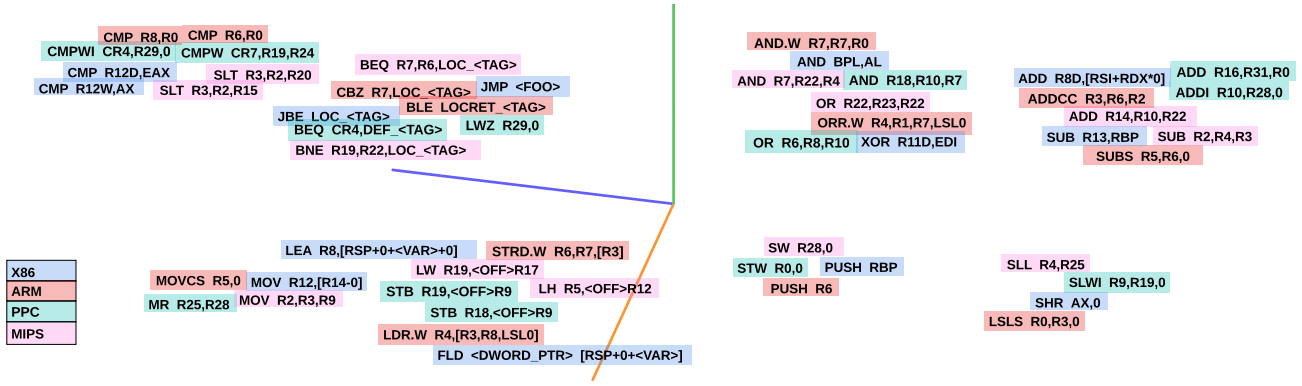


Figure 7: Visualization of MAIE for thirteen high-frequency x86 instructions and their top-one most similar instructions in the other three ISAs (ARM, MIPS and PPC), based on the cosine similarity of their MAIE.

7.3 Instruction Similarity Task

This task measures whether two semantically similar instructions, *regardless of their ISAs*, have close embeddings. Unlike word embeddings, which have many existing word-aligned corpora to evaluate their quality, we do not have such data. We thus create the datasets ourselves, which contain a set of manually-labeled instruction pairs. We rely on the assembly language references [5, 57, 62, 73] to create our datasets.

Specifically, we first categorize the x86 instructions into 6 categories (including data transfer, arithmetic, logical, shift and rotate, bit and byte, and control transfer). We then randomly select 20 x86 instructions from each category. For each selected x86 instruction, we find their corresponding similar instructions from the other three ISAs based on whether their opcodes share similar semantics (i.e., perform the same operation). E.g., `MOVQ` from x86 and `STR` from ARM can be used to store data in registers; thus, it is reasonable to consider them as similar. `BL` from ARM represents a branch with link; we thus consider `MOVQ` and `BL` as dissimilar. All the authors who are familiar with the assembly languages worked together to determine the instruction pairs. If there was any disagreement, we skipped this one and selected another x86 instruction with the same category to replace this one. Finally, we create three datasets: D_1 contains 120 similar and 120 dissimilar pairs of x86 and ARM instructions; D_2 contains the same number of pairs of x86 and MIPS instructions; and D_3 contains the same number of pairs of x86 and PPC instructions.

Our Results. Given each pair of cross-ISA instructions in the datasets, we calculate the cosine similarity of their MAIE to measure their similarity. For D_1, D_2, and D_3, we achieve an accuracy of 0.7625, 0.6625, and 0.6791, respectively. Note that in NLP, the accuracy of the cross-lingual word embedding similarity testing is around 0.68~0.73 [74].

Comparison with Prior Work. The prior work [64] is the

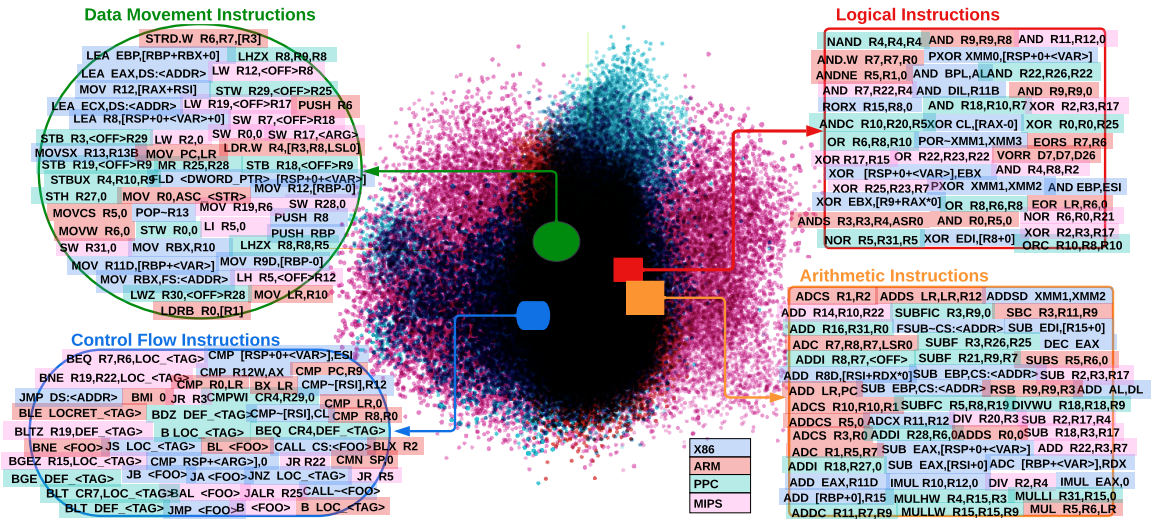


Figure 8: Visualization of all MAIE in four ISAs. Four categories of instructions are demonstrated: data movement, control flow, logical, and arithmetic, which show that similar instructions, regardless of their ISAs, have close embeddings.

most related to ours. As discussed in Section 2, there are three main differences between the prior work and ours. As the prior work is based on *supervised* learning, we cannot use our dataset to train its model. Hence, we use its open-sourced trained model [64] for comparison. We use the *same* testing dataset created by [64], which contains 25 similar and 25 dissimilar pairs of x86 and ARM instructions (the prior work only focuses on x86 and ARM). We achieve a higher AUC = 0.78, while the prior work 0.72. Thus, although ours is unsupervised learning, the learned MAIE have better quality in capturing instruction semantics across ISAs.

Nearest Neighbor Instructions. We next examine, for a given x86 instruction, its top-K similar instructions in other ISAs. Specifically, we select nine (9) high-frequency x86 instructions containing different opcodes. For each selected x86 instruction, we search for the top-two similar instructions in ARM, MIPS, and PPC, respectively, based on the cosine similarity of their MAIE. The results are shown in Table 2. We can see that for a given x86 instruction, its top-two similar instructions in the other ISAs share similar semantics, as predicted. E.g., for the x86 instruction `ADD R8D,[RSI+RDX*0]`, we find the relevant ARM instructions `ADDCC R3,R6,R2` and `ADDCC R1,R2,R3`, MIPS instructions `ADD R14,R10,R22` and `ADD R14,R10,R23`, and PPC instruction `ADD R16,R31,R0` and `ADD R17,R17,R0`, where all them add the values in two operands and store the result in the destination operand.

Visualization. We plot thirteen (13) high-frequency x86 instructions (including the previous nine instructions) and their corresponding *top-one* most similar instructions in the other ISAs (ARM, MIPS and PPC) in Figure 7 using t-SNE [54].

We can see that the instructions with similar semantics, regardless of their ISAs, have close embeddings (and thus appear nearby in the figure). For example, the four instructions, `AND BPL,AL` (x86), `AND.W R7,R7,R0` (ARM), `AND R7,R22,R4` (MIPS), and `AND R18,R10,R7` (PPC), related to the bitwise AND operation, appear nearby. As another example, the eight instructions related to the comparison operation from the four ISAs appear nearby, as shown in the top-left of the figure. Similarly, the twelve instructions related to the data movement operation from the four ISAs are also close together, as shown in the left-bottom of this figure.

We also visualize all MAIE in the four ISAs using t-SNE, as shown in Figure 8. Four categories of instructions are selected for demonstration: data movement, control flow, logical, and arithmetic. We can see that the data movement instructions across the four ISAs are close together. This is similar to the other categories. Therefore, our learned MAIE can successfully capture semantic relationships of instructions *within the same ISA* as well as *across ISAs*.

7.4 Malware Detection Task

To evaluate the transferability of MAIE, we conduct two downstream tasks, malware detection and function similarity detection. This section presents the result of the first task.

To evaluate the transferability of MAIE, we integrate MAIE into a mono-architecture model, and reuse the trained model for other ISAs (see Section 6). We do not claim the design of mono-architecture models as our contribution. Instead our contribution is the learning of MAIE that support knowledge transfer to achieve retargeted-architecture binary analysis.

Model. In this experiment, we use the Long Short Term Memory (LSTM) model proposed in [29] to detect malware. In

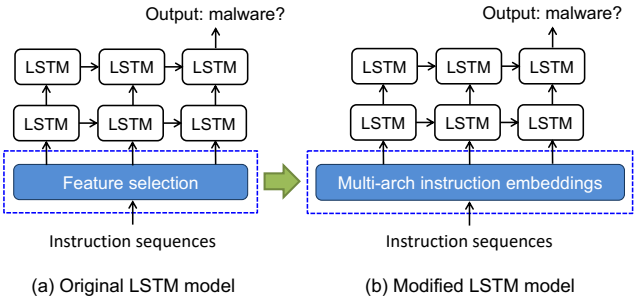


Figure 9: Integrating MAIE into the `LSTM` model for retargeted-architecture analysis.

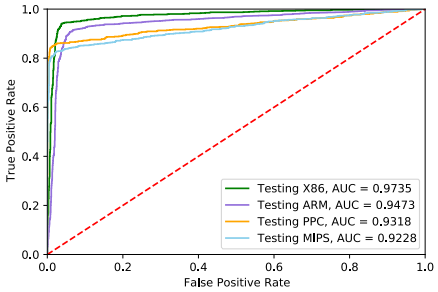


Figure 10: ROC for retargeted-architecture malware detection task.

Table 3: Comparing results between our approach and the first baseline method for the malware detection task.

	Train	Test	AUC
Train and test on the same ISA (*baseline*)	x86	x86	0.9735
	ARM	ARM	0.9938
	MIPS	MIPS	0.9525
	PPC	PPC	0.9021
Train on x86 and test on other ISAs (***ours***)	x86	ARM	0.9473
		MIPS	0.9318
		PPC	0.9228

Table 4: Performance changes as the training dataset size varies. The testing dataset remains the same. (Legend: *M* and *B* stands for *malware* and *benign*, respectively.)

Train	Training Size	Test	Testing Size	AUC
PPC	304(M) + 304(B)	PPC	203(M) + 203(B)	0.9021
x86	304(M) + 304(B)	PPC	203(M) + 203(B)	0.7837
	630(M) + 630(B)			0.8772
	960(M) + 960(B)			0.9112
	1294(M) + 1294(B)			0.9228

the original design, the LSTM model analyzes the opcode sequence of a program to detect whether it is malware or not.

We design the LSTM model as two layers. Figure 9 shows how we integrate the learned MAIE into the LSTM model to derive a multi-architecture model. Specifically, the feature extraction layer (highlighted in blue) is replaced with our multi-architecture instruction embedding (MAIE) layer, which represents each instruction using its MAIE.

Baseline Methods. We consider two baseline methods. (1) The first one is the LSTM model trained and tested on the *same* ISA. That is, we train and test LSTM using OAIE, rather than MAIE. (2) The second baseline is the prior work [64]. As the prior work only focuses on x86 and ARM, in the comparison, we also focus on this pair of ISAs.

Task-Specific Datasets. We first collect malware samples. We totally collect 2156, 1159, 1004, and 507 malware samples in x86, ARM, MIPS, and PPC from VirusShare.com [69].

Since the first baseline method trains and tests a malware detection model on the *same* ISA, it requires task-specific data in *each* ISA. We spend a lot of efforts in collecting malware samples, especially for MIPS and PPC. It is worth noting that our retargeted-architecture approach only needs malware in a high-resource ISAs for training a model, and thus can greatly save the efforts in data collection, especially for low-resource ISAs. That means, *in an extreme case, if only one PPC binary is available, we can still detect whether it is malware using the model trained on x86.*

For each ISA, we build its training and testing dataset. We divide the malware samples in each ISA into two parts: 60% are used for training and 40% for testing. In each training and testing dataset, we also include the same number of benign programs. In the training dataset, the benign programs are randomly selected from the four packages used to build the general dataset for training MAIE (Section 7.2), while the testing dataset contains benign programs randomly selected from completely different packages, including *Coreutils-9.0* and *Diffutils-3.7*, which are *not used to build the general dataset*. Through this, we can assess the adequacy of the general dataset in handling real-world scenarios, where the testing samples have never been seen during training.

Our Results. We first train LSTM on x86, and test it on x86, ARM, MIPS, and PPC, respectively. Through this, we can learn how much the model's accuracy decreases when reused for other ISAs. The less it decreases, the better the transferability of MAIE. Figure 10 shows the ROC curves.

We can see that (1) when the model is trained and tested on x86, it can achieve AUC = 0.97, and (2) when the model trained on x86 is reused for ARM, PPC, and MIPS, it can achieve AUC = 0.95 (with only 2% decreases), AUC = 0.93 (with 4% decreases), and AUC = 0.92 (with 5% decreases), respectively. The results show the good transferability of MAIE that can support transfer knowledge from one ISA to others.

Comparison with Baseline Methods. For ***the first baseline***, we train and test LSTM on the *same* ISA using OAIE. The results are shown in Table 3: when a model is trained and tested on ARM, MIPS, and PPC, it achieves AUC = 0.99, AUC = 0.95, and AUC = 0.90, respectively.

By comparing the first baseline and our approach, we can

see that our model's performance is slightly lower for ARM and MIPS, but still competitive (e.g., the baseline model trained and tested on MIPS achieves AUC = 0.95, while ours trained on x86 and tested on MIPS achieves AUC = 0.93).

For PPC, it is surprising that our model—trained on x86 and thus has not seen any PPC malware during training—achieves a higher AUC than the baseline model (trained on PPC). One reason may be that the PPC malware used to train the baseline model is not sufficient. While increasing the training dataset size can improve the baseline model performance, it can be challenging to collect data in certain tasks.

Next, we seek to understand how performance changes as the training dataset size varies. Specifically, we conduct experiments starting from the same size of the x86 and PPC training datasets, gradually increasing the x86 dataset size until reaching the maximum size. The results are shown in Table 4. We can see that when the model is trained on an x86 training dataset containing more than 960 malware samples and then reused for PPC, it outperforms the baseline model trained and tested on PPC with less available data. This demonstrates the critical role of a sufficiently large training dataset in order to achieve desirable performance.

These results demonstrate the great significance of our retargeted-architecture approach. (1) It only needs the training data in a high-resource ISA, which can greatly save the effort in data collection. (2) It trains one single model and reuses it for other ISAs, which can save computing resources and alleviate the *per*-ISA tuning effort. More importantly, the model predication accuracies keep high on other ISAs.

The second baseline is the prior work [64]. As it is based on supervised learning, we use its open-sourced trained model for comparison. It learns cross-architecture instruction embeddings (CAIE) for a pair of ISAs: x86 and ARM.

In Figure 9(b), we first replace the MAIE module with CAIE, so that the instructions are encoded using CAIE. We then train LSTM on x86 and test it on ARM. It achieves AUC = 0.8654. When MAIE are applied, a model trained on x86 and tested on ARM achieves AUC = 0.9473 (Figure 10). It shows that MAIE have much better transferability.

7.5 Function Similarity Detection Task

Model. `funcGNN` is a graph neural network-based model that analyzes CFGs to measure function similarity [58]. Fig. 11 shows how we integrate MAIE into `funcGNN` for retargeted-architecture binary analysis. The one-hot embedding layer (highlighted in blue) is replaced with multi-architecture instruction embedding layer, which represents each instruction as its MAIE. Each node is a basic block. We use the sum of MAIE of all instructions in a block to represent this block.

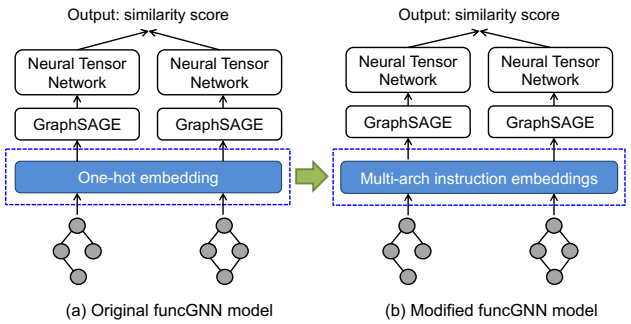
Baseline Methods. We also consider two baseline methods. (1) The first one is the `funcGNN` model trained and tested on the *same* ISA using OAIE. (2) The second baseline is the prior work [64] that only focuses on x86 and ARM.

Task-Specific Datasets. For this task, it is easy to collect task-specific datasets for different ISAs using cross-compilation. The significance of our retargeted-architecture approach is that we only need to train *one* model on one ISA and *reuse* it for all other ISAs. (1) We first build the training dataset for each ISA, where each contains 50,000 similar and 50,000 dissimilar function pairs in the corresponding ISA. (2) We then build the testing dataset for each ISA, containing 5,000 similar and 5,000 dissimilar functions pairs.

To make sure there is no overlap between training and testing datasets, we select completely different programs to build them. Specifically, the functions in the training datasets are randomly selected from the four packages used to build the general dataset for training MAIE (Section 7.2), while the functions in the testing dataset are randomly selected from completely different packages, including *Coreutils-9.0* and *Diffutils-3.7*, which are *not used to build the general dataset*. Following the dataset building method in InnerEye [80], we consider two functions *similar* if they are compiled from the same piece of source code, and dissimilar if their source code is rather different. Each program is compiled using four different optimization levels. Thus, given two similar or dissimilar functions, by applying different optimization levels to the two functions, we build 6 (= 4*3÷2) pairs in the datasets.

Our Results. We first train `funcGNN` on x86 and then test the trained model on x86. After that, we reuse the trained model for the other three ISAs. The results in Figure 12 demonstrate the good transferability of MAIE. The model (trained on x86) tested on x86 achieves AUC = 0.98. When the model is reused for ARM, MIPS, and PPC, it achieves AUC = 0.95 (with only 3% decreases), AUC = 0.94 (with only 4% decreases), and AUC = 0.95 (with only 3% decreases), respectively.

We next investigate the impact of optimization levels on the transferability of MAIE. Instead of using the assembly code generated at all optimization levels (O0-O3) to train MAIE (Section 7.2), we build four separate general datasets using code generated at each optimization level to train four sets of MAIE, referred to as MAIE_{O0}, MAIE_{O1}, MAIE_{O2}, and MAIE_{O3}. We then integrate each set of MAIE into `funcGNN` and follow the aforementioned procedure to evaluate the transferability. The task-specific training and testing datasets *remain the same* when each set of MAIE is integrated. Table 5 shows the results, while the results based on MAIE trained using code generated at all optimization levels (referred to as MAIE_{all}) are shown in Figure 12. We have the following observations. (1) MAIE_{Oi} (i = 0~3) all demonstrate relatively high transferability, in the sense that `funcGNN` achieves acceptable or good performance when reused for other ISAs. (2) The transferability of MAIE at a higher optimization level does not necessarily outperform that at a lower level. This highlights the drawback of training MAIE using code generated at individual optimization levels. (3) When MAIE_{all} are used, `funcGNN` has the highest AUC values, indicating that MAIE_{all} has the best transferability. This can be attributed to



(a) Original funcGNN model (b) Modified funcGNN model

Figure 11: Integrating MAIE into the `funcGNN` model for retargeted-architecture analysis.

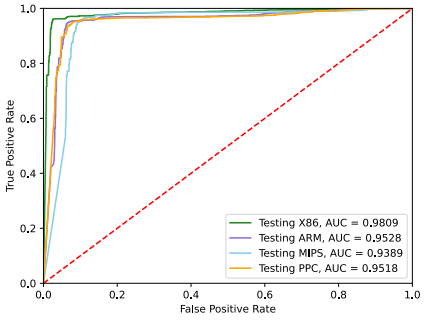


Figure 12: ROC for retargeted-architecture function similarity detection task.

Table 5: Impact of optimization levels on the transferability of MAIE. MAIE_{Oi} are learned using code generated at optimization level Oi. The results when MAIE are learned using code generated at all optimization levels are also shown in Figure 12 and Table 6.

MAIE	Train	Test	AUC	MAIE	Train	Test	AUC
MAIE_{O0}	x86	ARM MIPS PPC	0.8726 0.8434 0.7309	MAIE_{O1}	x86	ARM MIPS PPC	0.9049 0.9249 0.9287
MAIE_{O2}	x86	ARM MIPS PPC	0.8433 0.8838 0.9334	MAIE_{O3}	x86	ARM MIPS PPC	0.9484 0.8419 0.9172

Table 6: Comparing results of our approach and the first baseline for function similarity detection. The results of our approach are also shown in Figure 12.

	Train	Test	AUC
Train and test on the same ISA (*baseline*)	x86	x86	0.9809
	ARM	ARM	0.9896
	MIPS	MIPS	0.9718
	PPC	PPC	0.9847
Train on x86 and test on other ISAs (***ours***)	x86	ARM	0.9528
		MIPS	0.9389
		PPC	0.9518

the fact that MAIE_{all} are trained using a larger and more comprehensive dataset that covers a wider range of instructions, supporting more effective learning of embeddings.

Comparison with Baseline Methods. For ***the first baseline***, we train and test `funcGNN` on the *same* ISA using OAIE: it can achieve AUC = 0.98, 0.97, and 0.98, for ARM, MIPS, and PPC, respectively. The results are shown in Table 6.

Comparing the baseline and our approach, we can see that although the AUC values of our model are slightly lower but still keep high, considering that we do not need to spend a lot of efforts in data collection for each ISA as well as computing resources in training a large number of models which involves complicated parameter tuning, our approach is of great use.

The second baseline is the prior work [64]. As it focuses on x86 and ARM, we focus on this pair. We use its trained cross-architecture instruction embeddings (CAIE) for comparison. In Fig. 11(b), we first replace the MAIE layer (highlighted in blue) with the CAIE layer. We then train `funcGNN` on x86 and test it on ARM. It achieves AUC = 0.8832. When our MAIE is applied, a model trained on x86 and tested on ARM achieves AUC = 0.9528. Thus, our MAIE have better transferability.

7.6 Efficiency

We evaluate the training time of UniMap, which first learns OAIE of each ISAs using `fastText` (Part I) and then maps OAIE of the source ISAs (i.e., ARM, MIPS and PPC) to the vector space of x86 to learn MAIE (Part II). The training time of Part I and II is linear to the corpus size, and the total training time is the sum of the two parts.

Table 7: Training time (*minute*) of UniMap with respect to different embedding dimensions.

Embedding Dimension		100	200	300
Learning OAIE using CPU (Part I)	x86	4	7	10
	ARM	3	6	9
	MIPS	5	8	11
	PPC	4	6	8
Mapping OAIE to the vector space of x86 using GPU (Part II)	ARM → x86	13	21	26
	MIPS → x86	12	14	18
	PPC → x86	12	15	20

The results are shown in Table 7. Part I is performed using CPU, and Part II is performed using GPU. We can see that UniMap is very efficient. For example, if the embedding dimension is 200, the total training time for generating MAIE for x86 and ARM is $34 (= 7+6+21)$ minutes. Note that *MAIE once trained can be reused in various downstream tasks*: e.g., we use the same MAIE for both malware detection and function similarity detection tasks.

7.7 Hyperparameter Study

We vary the instruction embedding dimension and evaluate its impacts on the training time and model performance. We observe that increasing the dimensions yields better perfor-

mance. For example, (1) with 100 dimensions, the LSTM model achieves AUC = 95%, 90%, 90%, and 89%, when tested on x86, ARM, MIPS, and PPC, respectively. Thus, it has 5%, 5%, and 6% decreases when the model (trained on 86) is reused for ARM, MIPS and PPC, respectively. (2) With 300 dimensions, the model achieves AUC = 97%, 95%, 94%, and 92%, when tested on x86, ARM, MIPS, and PPC, respectively, with less than 2%, 3%, and 5% decreases when reused for ARM, MIPS and PPC, respectively.

However, a higher embedding dimension leads to higher computational costs (requiring longer training time, as shown in Table 7). Thus, a moderate dimension of 200 is a good trade-off between accuracy and efficiency.

8 Discussion

Vocabulary Sizes. Our approach learns a linear transformation that maps the instruction embeddings of the source ISA to the vector space of the target. For it to work, different ISAs need to have relatively similar vocabulary sizes. Through our instruction normalization, the gap of the vocabulary sizes are significantly reduced (see Section 5.1), and our evaluation demonstrate that MAIE lead to good performance. However, there may exist an ISA whose vocabulary size is much larger (or smaller) than that of another. In this case, we may select a subset of instructions in this (or another) ISA for learning the transformation. Moreover, finding the maximum gap of vocabulary sizes that allows our approach to work is an interesting question. We leave it for future work.

Alternative Solutions. It would be intriguing to explore approaches alternative to a linear transformation for learning MAIE. For example, one can train a transformer [60] using the general data from multiple ISAs to learn MAIE. However, to achieve retargeted-architecture binary analysis, there is a crucial requirement for MAIE—i.e., semantically-similar instructions across ISAs should have close embeddings in a shared space. Whether this approach satisfies this requirement remains unclear and needs investigation. Plus, the fine-tuning step in [60] is impeded by the data scarcity issue. How to avoid or mitigate the issue also needs dedicated work.

In this work, we consider *instructions* as *words*, which is a natural choice and works well according to our evaluation. However, there exist other choices. For example, we may also consider *raw bytes* [28,42,63] or *opcode/operands* [20,40,60] as words. An interesting research is to study which choice works the best and whether it depends on ISAs and downstream tasks. We use the mature Skip-gram model to generate OAIE (and then map OAIE to a shared vector space for learning MAIE). There exist other models for learning OAIE, such as BERT [40] and transformer-based models [60]. It is worth highlighting that our retargeted-architecture analysis does not depend a particular word embedding approach; however, its performance may benefit from progress in this topic.

Generalizability. Considering that x86 is the most data-rich ISA, we have developed techniques to transfer knowledge from x86 to other ISAs. To validate the effectiveness of the techniques, we conducted an evaluation that tested the transferability from x86 to three other ISAs for two critical downstream tasks. In theory, the techniques could apply generally. However it is important to note that our results are specific to the evaluated scenarios. In order to ascertain the effectiveness of this approach across other downstream tasks and different source/target ISAs, further investigation and comprehensive testing are needed.

9 Conclusion

We have proposed a new direction, *retargeted-architecture binary code analysis*, where a deep learning model trained for one ISA can be reused for others. It has great significance for coping with the data scarcity issue and saving the dataset building effort and computing resources, compared to training per-ISA models. For NLP-based binary analysis approaches, our approach enables the reuse of their models for various ISAs. To that end, an unsupervised learning method, UniMap, has been developed to learn multi-architecture instruction embeddings (MAIE), where semantically similar instructions, regardless of their ISAs, have close embeddings. We evaluated the quality and transferability of MAIE through two important binary analysis tasks: malware detection and function similarity measurement. Our approach significantly outperforms prior work. We thus advocate that retargeted-architecture binary analysis is a promising direction for multi-ISA binary analysis.

Acknowledgements

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-2309550, CNS-2310322, CNS-2309477, and CNS-2304720. The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] ARM assembly language. https://downloads.ti.com/docs/esd/SPNU118O/Content/SPNU118O_HTML/assembler_description.html.

[2] ARM registers. https://bluetechs.wordpress.com/zothers/x/registers/.

[3] MIPS assembly/MIPS details. https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Details.

[4] angr Documentation. Intermediate representation. https://docs.angr.io/advanced-topics/ir.

[5] ARM. Arm instruction reference. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHEDHIF.html.

[6] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. A robust self-learning method for fully unsupervised cross-lingual mappings of word embeddings. *arXiv preprint arXiv:1805.06297*, 2018.

[7] BAP. Bap: Binary analysis platform. https://github.com/BinaryAnalysisPlatform/bap/.

[8] Nuria Bel, Cornelis HA Koster, and Marta Villegas. Cross-lingual text categorization. In *International Conference on Theory and Practice of Digital Libraries*, 2003.

[9] BitBlaze. Bitblaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu.

[10] Peter F Brown, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 1993.

[11] Dong-Kyu Chae, Sang-Wook Kim, Jiwoon Ha, Sang-Chul Lee, and Gyun Woo. Software plagiarism detection via the static api call frequency birthmark. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. In *FSE*, 2016.

[13] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. Pfirewall: Semantics-aware customizable data flow control for home automation systems. In *NDSS*, 2019.

[14] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *USENIX Security*, 2017.

[15] Computer Wiki. List of cpu architectures. https://computer.fandom.com/wiki/List_of_CPU_Architectures.

[16] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *arXiv preprint arXiv:1710.04087*, 2017.

[17] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *PLDI*, 2016.

[18] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *ACM SIGPLAN Notices*, 2017.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *S&P*, 2019.

[21] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *NDSS*, 2020.

[22] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[23] EverybodyWilk. List of instruction sets. https://en.everybodywiki.com/List_of_instruction_sets.

[24] fastText. Word representations. https://fasttext.cc/docs/en/unsupervised-tutorial.html.

[25] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *CCS*, 2016.

[26] Stephan Gouws, Yoshua Bengio, and Greg Corrado. Bilbowa: Fast bilingual distributed representations without word alignments. In *International Conference on Machine Learning*, 2015.

[27] Spence Green, Nicholas Andrews, Matthew R Gormley, Mark Dredze, and Christopher D Manning. Entity clustering across languages. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2012.

[28] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *USENIX Security*, 2019.

[29] Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, 2018.

[30] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *IEEE International Conference on Software Maintenance and Evolution*, 2017.

[31] IDA. Dummy name prefixes. https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names/.

[32] Yoon-Chan Jhi, Xiaoqi Jia, Xinran Wang, Sencun Zhu, Peng Liu, and Dinghao Wu. Program characterization using runtime values and its application to software plagiarism detection. *IEEE Transactions on Software Engineering*, 2015.

[33] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. Value-based program characterization and its application to software plagiarism detection. In *ICSE*, 2011.

[34] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

[35] Nikhil Ketkar. Stochastic gradient descent. In *Deep learning with Python*. Springer, 2017.

[36] Alexandre Klementiev, Ivan Titov, and Binod Bhattarai. Inducing crosslingual distributed representations of words. *COLING*, 2012.

[37] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, 2006.

[38] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 2014.

[39] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, and Ki-Woong Park. Learning binary code with deep learning to detect software weakness. In *KSII the 9th International Conference on Internet*, 2017.

[40] Xuezixiang Li, Qu Yu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *CCS*, 2021.

[41] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization. In *Information and Communications Security*, 2018.

[42] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *ASE*, 2018.

[43] Lannan Luo. Heap memory snapshot assisted program analysis for android permission specification. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, 2020.

[44] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. Repackage-proofing android apps. In *DSN*, 2016.

[45] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[46] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 2017.

[47] Lannan Luo and Qiang Zeng. Solminer: mining distinct solutions in programs. In *International Conference on Software Engineering Companion*, 2016.

[48] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *MobiSys*, 2017.

[49] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. Tainting-assisted and context-migrated symbolic execution of android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing*, 2019.

[50] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on iot cloud platforms. In *ACSAC*, 2021.

[51] Zhenhao Luo, Baosheng Wang, Yong Tang, and Wei Xie. Semantic-based representation binary clone detection for cross-architectures in the internet of things. *Applied Sciences*, 2019.

[52] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *NDSS*, 2023.

[53] Thang Luong, Hieu Pham, and Christopher D Manning. Bilingual word representations with monolingual quality in mind. In *the 1st Workshop on Vector Space Modeling for Natural Language Processing*, 2015.

[54] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 2008.

[55] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedsings of Workshop at ICLR*, 2013.

[56] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013.

[57] MIPS. Mips opcode and instruction reference home. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf.

[58] Aravind Nair, Avijit Roy, and Karl Meinke. funcgnn: A graph neural network approach to program similarity. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020.

[59] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *ICSE*, 2017.

[60] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[61] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *S&P*, 2015.

[62] PowerPC. Powerpc opcode and instruction reference home. http://math-atlas.sourceforge.net/devel/assembly/ppc_isa.pdf.

[63] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the 32 AAAI Conference on Artificial Intelligence*, 2018.

[64] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. In *NDSS Workshop on Binary Analysis Research*, 2019.

[65] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

[66] Stefano Sebastio, Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Optimizing symbolic execution for malware behavior classification. *Computers & Security*, 2020.

[67] The IDA Pro Disassembler and Debugger. http://www.datarescue.com/idabase/.

[68] Thanh Van Nguyen, Anh Tuan Nguyen, Hung Dang Phan, Trong Duc Nguyen, and Tien N Nguyen. Combining word2vec with revised vector space model for better code retrieval. In *the 39th International Conference on Software Engineering Companion*, 2017.

[69] VirusShare. Open repository of malware samples. https://virusshare.com/.

[70] Junzhe Wang and Lannan Luo. Privacy leakage analysis for colluding smart apps. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2022.

[71] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *the 16th ACM conference on Computer and communications security*, 2009.

[72] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Detecting software theft via system call based birthmarks. In *Computer Security Applications Conference*, 2009.

[73] x86. x86 opcode and instruction reference home. http://ref.x86asm.net/coder32.html.

[74] Ruochen Xu, Yiming Yang, Naoki Otani, and Yuexin Wu. Unsupervised cross-lingual transfer of word embedding spaces. *arXiv preprint arXiv:1809.03633*, 2018.

[75] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *CCS*, 2017.

[76] Qiang Zeng, Golam Kayas, Emil Mohammed, Lannan Luo, Xiaojiang Du, and Junghwan Rhee. Heaptherapy+: Efficient handling of (almost) all heap vulnerabilities using targeted calling-context encoding. In *DSN*, 2019.

[77] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. Resilient decentralized android application repackaging detection using logic bombs. In *International Symposium on Code Generation and Optimization*, 2018.

[78] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[79] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.

[80] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint (NDSS'19)*, 2018.

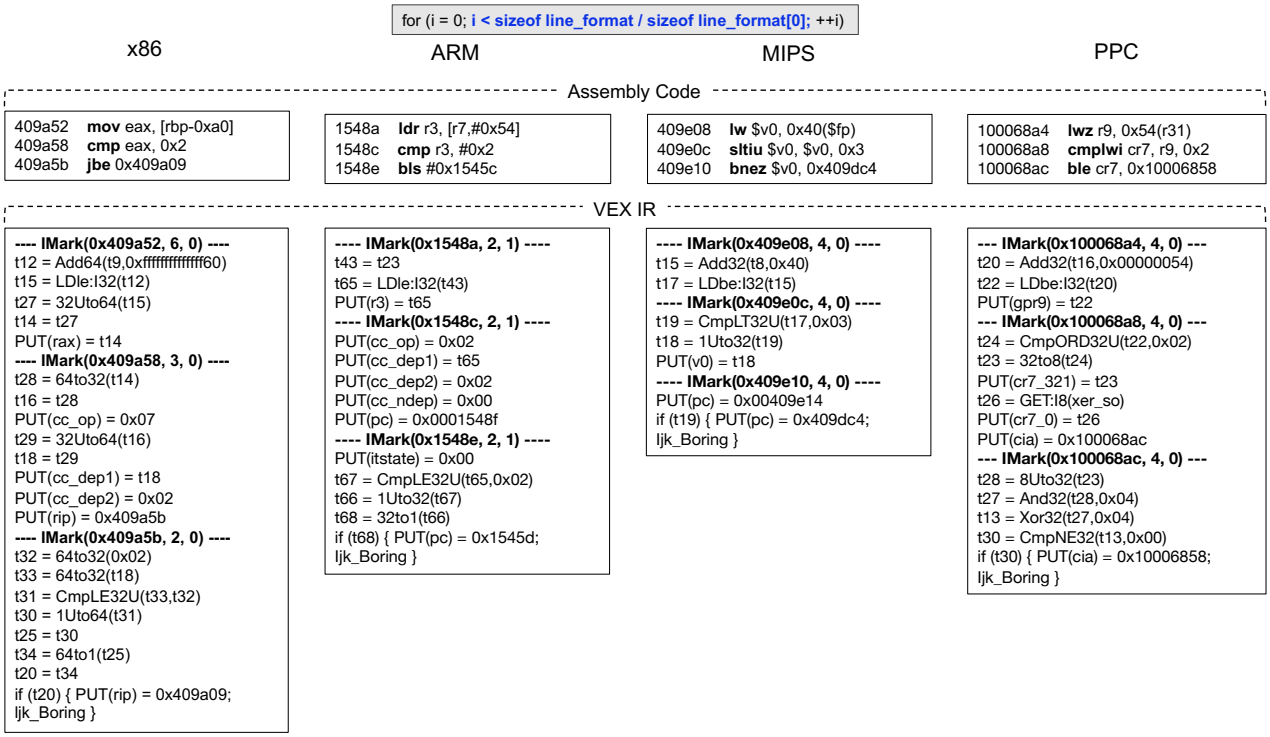


Figure 13: A example of C source code and the corresponding assembly code and VEX IR code in four different ISAs.

A Appendix

A.1 IR Code

Figure 13 shows the source code and the corresponding assembly code and VEX IR in four different ISAs, x86, ARM, MIPS and PPC. The source code we consider here is: `i < sizeof line_format / sizeof line_format[0]`, which is highlighted in blue at the top. Its corresponding assembly code and IR code are shown below. In VEX IR, the `IMark` is an IR statement that does not represent actual code. Instead it indicates the address and length of the corresponding assembly instruction. For example, in x86, the address of the first assembly instruction `mov eax, [rbp-0xa0]` is `0x409a52`, and it is translated to five IR statements belonging to the first `IMark` statement, `IMark(0x409a52, 6, 0)`.

From the example, we can see that given the same piece of source code, the assembly code in different ISAs is quite different. More importantly, even if we lift the binaries into a common IR (e.g., VEX IR), their IR code is also quite different (e.g., the lengths and types of IR statements vary).

Therefore, existing works that leverage IR for analyzing binaries across ISAs have to perform further advanced analysis on the IR code. For example, (1) Multi-MH [61] uses fuzzing to detect whether two pieces of VEX IR code (after lifting) are semantically similar. (2) GitZ [18] conducts complex re-optimization on IR code for function similarity comparison. (3) GeneDiff [51] trains a deep learning model using VEX IR code for cross-architecture binary clone detection. Thus, IR is not magic and a "bridge" (MAIE in our work) is needed for a model trained for one ISA to be reused for other ISAs.