Learning Cross-Architecture Instruction Embeddings for Binary Code Analysis in *Low-Resource Architectures*

Junzhe Wang
George Mason University
jwang69@gmu.edu

Qiang Zeng
George Mason University
zeng@gmu.edu

Lannan Luo ✉
George Mason University
lluo4@gmu.edu

Abstract

Binary code analysis is indispensable for a variety of software security tasks. Applying deep learning to binary code analysis has drawn great attention because of its notable performance. Today, source code is frequently compiled for various Instruction Set Architectures (ISAs). It is thus critical to expand binary analysis capabilities to multiple ISAs. Given a binary analysis task, the scale of available data on different ISAs varies. As a result, the rich datasets (e.g., malware) for certain ISAs, such as x86, lead to a disproportionate focus on these ISAs and a negligence of other ISAs, such as PowerPC, which suffer from the "*data scarcity*" problem. To address the problem, we propose to learn *cross-architecture instruction embeddings* (CAIE), where semantically-similar instructions, *regardless of their ISAs*, have close embeddings in a shared space. Consequently, we can transfer a model trained on a data-rich ISA to another ISA with less available data. We consider four ISAs (x86, ARM, MIPS, and PowerPC) and conduct both intrinsic and extrinsic evaluations (including malware detection and function similarity comparison). The results demonstrate the effectiveness of our approach to generate high-quality CAIE with good transferability.

1 Introduction

Binary code analysis, which allows one to analyze binary code without access to the corresponding source code, plays a critical role in a wide range of different tasks, including code plagiarism detection (Luo et al., 2014; Jhi et al., 2015), malware classification (Zhang et al., 2014; Sebastio et al., 2020), function similarity detection (Li et al., 2022; Ding et al., 2019), and vulnerability discovery (Pewny et al., 2015; Eschweiler et al., 2016).

Today, software is frequently cross-compiled for various Instruction Set Architectures (ISAs). For example, hardware vendors often use the same code base to compile firmware for different devices that operate on varying ISAs (e.g., x86 and ARM), which causes a single vulnerability at source-code level to spread across binaries of diverse devices. As a result, *cross-architecture binary code analysis* has become an emerging problem that draws great attention (Pewny et al., 2015; Feng et al., 2016; Xu et al., 2017; Zuo et al., 2019). Analysis of binaries across ISAs, however, is non-trivial: such binaries differ greatly in instruction sets, calling conventions, general- and special-purpose CPU register usages, memory addressing modes, and more.

Recently, we have witnessed a surge of research efforts that leverage deep learning to tackle various binary code analysis tasks. Deep learning has demonstrated its strengths on code analysis, and shown noticeably better performances over traditional program analysis-based methods in terms of both accuracy and scalability. However, training a deep learning model usually requires massive amount of data. As a result, most deep learning-based binary analysis models have been dedicated to a *high-resource* ISA, such as x86, where large-scale labeled datasets exist for training their models. But for many other ISAs, such as PowerPC, there are few or even no labeled dataset, resulting in a negligence focus on those *low-resource* ISA. Moreover, it is labor intensive and time-consuming to collect data samples and manually label them to build datasets for such low-resource ISAs.

Our Approach. A binary, after being disassembled, is expressed in an assembly language. Given this insight, `InnerEye` (Zuo et al., 2019) proposed to adapt deep learning techniques developed for natural language processing (NLP) to binary code analysis. Since then a surge of NLP-inspired binary analysis approaches have been proposed (Li et al., 2022, 2023; Zan et al., 2022; Chen et al., 2021; Redmond et al., 2019; Duan et al., 2020). In many NLP tasks, words are often converted

Findings of the Association for Computational Linguistics: NAACL 2024, pages 1320–1332

into *word embeddings*, which capture the semantic meaning of words, to facilitate further processing (Mikolov et al., 2013a; Wieting et al., 2015; Tang et al., 2014). To analyze binary code, we regard *instructions as words* and *basic blocks as sentences*.

Inspired by cross-lingual word embeddings in NLP (Mikolov et al., 2013b), we propose a novel approach to *tackle the challenge of data scarcity in binary code analysis*. Our approach learns *cross-architecture instruction embeddings* (CAIE), where semantically-similar instructions, regardless of their ISAs, have close embeddings in a shared space. Equipped with such a shared space, we can transfer knowledge from one ISA to another, especially in low-resource scenarios. Specifically, by projecting instructions disassembled from binaries in different ISAs into the shared space, we can train a model using only the data in a high-resource ISA, and transfer it to a low-resource ISA.

Unlike NLP, where obtaining cross-lingual signals can be difficult, obtaining cross-architecture signal for binary code analysis across ISAs is not a challenging task (see Section 4.1). We thus design a supervised model by borrowing the idea and technique from the `Bi-SENT2VEC` algorithm (Sabet et al., 2019) in NLP to learn CAIE. We first build a dataset consisting of a large number of semantically-equivalent basic block pairs and use them to train a joint model which learns to predict both the instruction and context in the source and target basic blocks. The model explores rich semantic relationships among instructions within its own ISA as well as across a different ISA. It is also fully self-supervised, which helps overcome the limitation of unsupervised linear transformation.

We have implemented our model, called `CrossIns2Vec`, and evaluated it on four ISAs: x86, ARM, MIPS and PowerPC (PPC). (1) In intrinsic evaluation, we conduct the instruction similarity task to evaluate the *quality* of CAIE—whether they capture the syntax and semantic information of instructions across ISAs. (2) In extrinsic evaluation, we conduct two downstream binary analysis tasks to evaluate the *transferability* of CAIE, including function similarity comparison and malware detection. We also compare `CrossIns2Vec` to the baseline methods. The results show that `CrossIns2Vec` can effectively generate high-quality CAIE with better transferability. Below summarizes our contributions:

- To address the data scarcity issue in binary code analysis, we propose to learn *cross-architecture instruction embeddings* (CAIE), where semantically-similar instructions, regardless of their ISAs, have close embeddings in a shared space. Equipped with such a shared space, given a binary analysis task, we can transfer a model trained on a data-rich ISA to another ISA with less available data.
- We have implemented a supervised model for learning CAIE and conducted both intrinsic and extrinsic evaluations on four ISAs: x86, ARM, MIPS and PPC. The results demonstrate the effectiveness of our approach.
- NLP-inspired binary code analysis is a promising research direction, but not all NLP techniques are applicable to binary code analysis. Thus, studies like ours that identify and examine effective NLP techniques for binary code analysis are valuable in advancing exploration along this direction. Our evaluation shows how the adaptation works and why it is useful through two critical binary analysis tasks.
- We release the source code, datasets, trained models, and learned CAIE to facilitate the follow-up research in this direction[1].

2 Related Work

2.1 Traditional Code Analysis

Mono-architecture. Most traditional approaches work on a *single* ISA. Some analyze source code (Kamiya et al., 2002; Wang and Luo, 2022; Luo and Zeng, 2016). Others analyze binary code (Luo, 2020; Zeng et al., 2019b; Luo et al., 2019a; Zeng et al., 2019a, 2018; Luo et al., 2016), e.g., using symbolic execution (Luo et al., 2014, 2021, 2017), but are expensive, and inapplicable for large codebases. Dynamic approaches include API birthmark (Tamada et al., 2004; Chae et al., 2013), system call birthmark (Wang et al., 2009), and instruction birthmark (Tian et al., 2013; Park et al., 2008). *Extending them to other ISAs would be hard*. Plus, code coverage is another challenge.

Cross-architecture. Recent works have applied traditional approaches to the *cross-architecture* scenario (Pewny et al., 2015; Eschweiler et al., 2016; Chandramohan et al., 2016; Feng et al., 2017;

[1] https://github.com/lannan/CrossIns2Vec

David et al., 2017, 2018, 2016). Multi-MH and Multi-k-MH (Pewny et al., 2015) are the first two for comparing functions in different ISAs, but their fuzzing-based basic-block similarity comparison and graph (i.e., CFG) matching algorithms are expensive. discovRE (Eschweiler et al., 2016) uses pre-filtering to boost the matching process, but is unreliable and has many false negatives. Esh (David et al., 2016) compares basic blocks using a SMT solver, which is unscalable.

2.2 Machine/Deep Learning-based Analysis

Mono-architecture. Recent research has applied machine/deep learning to code analysis (Li et al., 2022; Ahmad et al., 2020; Allamanis et al., 2016; Wei et al., 2019; Hu et al., 2018; Shido et al., 2019; Chen et al., 2021; Nguyen et al., 2017; Van Nguyen et al., 2017; Ahmad and Luo, 2023; Han et al., 2017). Asm2Vec (Ding et al., 2019) considers functions as documents and uses a PV-DM model to generate function embeddings. PalmTree (Li et al., 2021) generates token embeddings based on BERT (Devlin et al., 2018). However, these works only focus on a single ISA.

Cross-architecture. Most existing models are trained and tested on a pair of ISAs and the training needs the task-specific data for each ISA of a given pair (Feng et al., 2016; Xu et al., 2017; Chandramohan et al., 2016; Zuo et al., 2019; Massarelli et al., 2019). InnerEye (Zuo et al., 2019) adopts Neural Machine Translation techniques to measure the similarity of binary code across ISAs. VulHawk (Luo et al., 2023) lifts binary code into IR and uses NLP techniques to generate function embeddings. These approaches require the task-specific data for each ISA, and cannot resolve the data scarcity problem. Our approach differs significantly from them: we aim at model reuse, that is, *transferring a model trained on one ISA to another*, thereby eliminating the need for data from another ISA, especially for low-resource ISAs.

The Most Related Work. To the best of our knowledge, UniMap (Wang et al., 2023) is the *only work* that shares the same goal as ours: focusing on learning cross-architecture instruction embeddings (CAIE) to tackle the data scarcity issue. Their approach relies on unsupervised learning, eliminating the requirement for parallel data. However, in the context of binary code analysis, due to the prevalence of *cross-compilation*, obtaining cross-architecture signals is not challenging compared to that in NLP when seeking cross-lingual signals (see Section 4.1). Moreover, in NLP, studies have shown that cross-lingual word embeddings, learned through supervised learning, have superior transferability compared to those learned by unsupervised learning (Upadhyay et al., 2016; Ruder et al., 2019). Based on these, our approach takes a different direction by designing a supervised model for learning CAIE. The evaluation results demonstrate that our supervised learning-based CAIE exhibit superior quality and enhanced transferability when compared to the most related work.

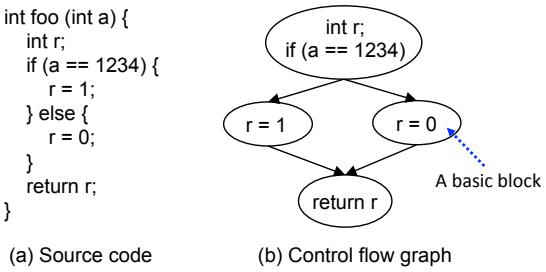

Figure 1: Control flow graph and basic block.

3 Background and Motivation

3.1 Control Flow Graph and Basic Block

A control flow graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications. A CFG is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks.

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or branching except at the end. Figure 1 shows an example of a piece of source code and its CFG, where each node is a basic block. Similarly, we can generate the CFG for a piece of binary code. We here use the source code as an example for simplicity.

3.2 Motivation

Let us consider the malware detection task as an example. Recently, applying deep learning to detect malware has garnered significant attention due to its remarkable performance capabilities. However, training a deep learning model usually requires a large amount of data. As a result, the rich datasets (e.g., malware) for certain ISAs, such as x86, lead to a disproportionate focus on these ISAs and a negligence of other ISAs, such as PowerPC, which suffer from the "*data scarcity*" problem (i.e., few

or even no labeled datasets exist). Moreover, it is labor-intensive and time-consuming to collect data samples and manually label them to build datasets for such low-resource ISAs. Dealing with the data scarcity issue is an unresolved challenge.

With some ISAs, like x86, being widely used, it becomes more feasible to collect sufficient data for these ISAs. Thus, it would be a great advantage if the abundance of training data for widely-used ISAs could facilitate the automated analysis of binaries in other ISAs where such data is scarce. For example, suppose a large training dataset exists for ISA **X**, but we need to analyze a binary ***b*** in ISA **Y**, for which the available training data is insufficient. As a result, it is difficult to train a model on **Y** for analyzing ***b***. To address this issue, our idea is to *transfer knowledge from ISA **X** to **Y***, such that we can train a model on **X** and transfer the trained model to perform prediction on ***b*** in **Y**.

To achieve this objective, it is essential to address the syntactic variations among different ISAs. Drawn inspiration from cross-lingual word embeddings in NLP, we propose to learn *cross-architecture instruction embeddings* (CAIE), where semantically-similar instructions, regardless of their ISAs, have close embeddings in a shared space. Equipped with such a shared space, we can transfer knowledge from one ISA to another, especially in low-resource scenarios. As a result, we can *train a model using only the data in a high-resource ISA, and transfer it to a low-resource ISA.*

3.3 Why not IR?

Intermediate representation (IR) can be used to represent code of different ISAs. For example, VEX IR is an architecture-agnostic and side-effect-free representation that can represent instruction sets of different ISAs in a uniform style. Thus, this raises the question of whether IR can serve as the bridge for achieving model reuse.

After conducting thorough investigations and experiments, we found that: given two binaries with different ISAs that are compiled from the ***same*** piece of source code, after we lift them into a common IR, the resulting IR code differs greatly. Specifically, the lengths and types of the IR statements can vary significantly from one another. Figure 2 in shows an example, where the source code is highlighted in blue at the top and the corresponding assembly code and VEX IR in x86 and MIPS are shown below. In VEX IR, the `IMark` statement indicates the address and length of its 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)`. This is similar for the IR code in ARM and MIPS, where the resulting IR code is significantly distinct. (Additional examples can be found in Figure 1 and Figure 3 of (Pewny et al., 2015)).

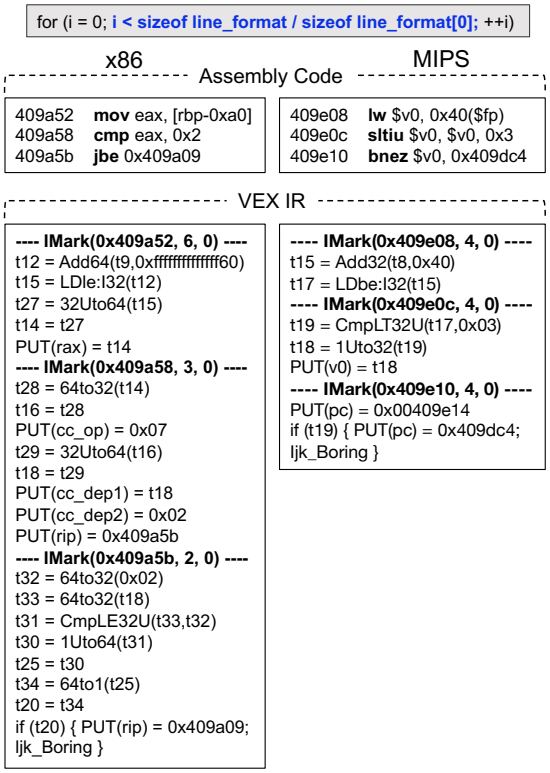

Figure 2: A example of C source code (highlighted in blue) and the corresponding assembly code and VEX IR code in different ISAs.

Therefore, existing works that utilize IR for analyzing binaries across ISAs have to perform further *advanced* analysis on the IR code. For example, (1) `Multi-MH` (Pewny et al., 2015) uses fuzzing to detect whether two pieces of VEX IR code (after lifting) are semantically similar. (2) `GitZ` (David et al., 2017) conducts complex re-optimization on IR code to compare function similarity. (3) `GeneDiff` (Luo et al., 2019b) applies deep learning analysis to VEX IR code for cross-architecture binary clone detection. Thus, IR is not "magic" that can directly serve as the "bridge" for facilitating model reuse. This work, therefore, focuses on building the "bridge"—i.e., cross-architecture instruction embeddings (CAIE)—for enabling a model trained for one ISA to be reused for other ISAs.

4 Model Design

In contrast to the challenges faced in NLP, where obtaining cross-lingual signals can be a difficult task, acquiring cross-architecture signals for binary analysis across ISAs is straightforward (Section 4.1). Moreover, studies have shown that supervised methods typically exhibit superior performance compared to unsupervised ones (Upadhyay et al., 2016; Ruder et al., 2019). We thus design a supervised model, called `CrossIns2Vec`, to learn cross-architecture instruction embeddings (CAIE). Figure 3 shows the model architecture. As we consider *instructions as words* and *basic blocks as sentences*, the input is a pair of semantically-equivalent basic blocks, B_1 and B_2, in different ISAs. Initially, each instruction is assigned a random vector. During the joint learning process, `CrossIns2Vec` effectively learns CAIE for each instruction. Below we present the detailed process.

4.1 Collecting Semantically-Equivalent Basic Block Pairs

We first need to collect the semantically-equivalent basic block pairs from different ISAs. We consider basic blocks of different ISAs that are *compiled from the same piece of source code as semantically-equivalent*. To determine the ground truth regarding the similarity of basic blocks, we rely on the source code line number. Specifically, if two basic blocks from different ISAs have the *same starting* and *end* source code line numbers, they are considered to be semantically-equivalent.

To this end, we first collect the source code of various programs and compile each one for different ISAs by cross-compilation. This process proves to be both convenient and feasible for handling various ISAs, thanks to the availability of tools such as QEMU (QEMU, 2023) and LLVM (LLVM, 2023). Consequently, the task of *acquiring cross-architecture signals across ISAs poses no significant challenge in binary code analysis*.

During the cross-compilation process, we include the "`-g`" compiling option. This ensures that the compiled binary file contains the DWARF debug information, including valuable details like the source code line number for each assembly instruction. After getting the binaries, we use IDA Pro (IDA, 2023) to disassemble each binary and generate control flow graphs (CFGs), where each node represents a basic block. During disassembly, we take advantage of IDA disassembly options, which can display the source code line number for each basic block. By leveraging these line numbers, we can identify semantically-equivalent basic block pairs. Specifically, for each basic block in one ISA, we search for its counterpart in another ISA if they have the *same starting* and *end* source code line number, indicating that they are compiled from the same piece of source code.

4.2 Learning Cross-architecture Instruction Embeddings

Our goal is to learn CAIE, where semantically-similar instructions, *regardless of their ISAs*, have embeddings that are close in a shared space.

In NLP, if a trained model is used to convert a word that has never appeared during training, the word is called an out-of-vocabulary (OOV) word and the embedding generation for them will fail. To mitigate the OOV issue, similar to `UniMap` (Wang et al., 2023), we normalize instructions by applying the following rules: (C1) replacing number constants with 0, while preserving minus signs; (C2) replacing string literals with `<STR>`; and (C3) replacing function names with `<FOO>`; (C4) other symbols are replaced with `<TAG>`.

Drawing inspiration from `Bi-SENT2VEC` (Sabet et al., 2019) in NLP, we design our model to learn CAIE based on two objectives: (1) *mono-architecture* objective: similar instructions in the *same* ISA are assigned close embeddings; (2) *cross-architecture objective*: similar instructions across *different* ISAs are assigned close embeddings.

Mono-Architecture Objective. The training objective is to predict a masked instruction e_t in a basic block B using the representation of the rest instructions in B, denoted as $\mathbf{v}_{B\setminus\{e_t\}}$. We use logistic loss $l : x \rightarrow log(1 + e^{-x})$ in conjunction with negative sampling to formulate the training objective. The training objective is computed as:

$$\min \sum_{B\in C}\sum_{e_t\in B}(l(\mathbf{u}_{e_t}^{\mathsf{T}}\mathbf{v}_{B\setminus\{e_t\}}) + \sum_{e'\in N_{e_t}} l(-\mathbf{u}_{e'}^{\mathsf{T}}\mathbf{v}_{B\setminus\{e_t\}})) \tag{1}$$

where e_t the masked instruction in B, and N_{e_t} the set of words sampled negatively for the masked instruction e_t. The set of negative instructions N_{e_t} are sampled following a multinomial distribution where each instruction e is associated with a probability: $p = \sqrt{f_e}/\sum_{e_i\in C}\sqrt{f_{e_i}}$, where f_e is the normalized frequency of e in the corpus.

Cross-Architecture Objective. To capture semantic relations of instructions across ISAs, we include

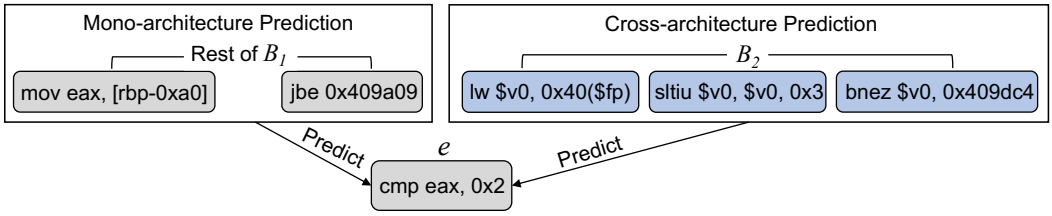

Figure 3: The `CrossIns2Vec` model.

a cross-architecture training objective, where given two semantically-equivalent basic blocks (B_1, B_2), a masked instruction e_t in B_1 is predicted using all instructions in B_2, denoted as $\mathbf{v}_{B_2}$. The training objective is computed as:

$$\min \sum_{(B_1,B_2)\in C} \sum_{e_t \in B_1} (l(\mathbf{u}_{e_t}^{\mathsf{T}} \mathbf{v}_{B_2}) + \sum_{e' \in N_{e_t}} l(-\mathbf{u}_{e'_t}^{\mathsf{T}} \mathbf{v}_{B_2})) \tag{2}$$

where e_t is the masked instruction in B_1, and N_{e_t} is the set of words sampled negatively for $e_t \in B_1$, following the same strategy as Equation 1.

Model Final Objective. By combining the mono-architecture and cross-architecture objectives, the objective function of our model is formulated as:

$$\min \sum_{e_t \in B_1} \underbrace{(l(\mathbf{u}_{e_t}^{\mathsf{T}} \mathbf{v}_{B_1 \setminus \{e_t\}}) + \sum_{e' \in N_{e_t}} l(-\mathbf{u}_{e'}^{\mathsf{T}} \mathbf{v}_{B_1 \setminus \{e_t\}}) +}_{\text{Mono-architecture loss}}$$
$$\underbrace{l(\mathbf{u}_{e_t}^{\mathsf{T}} \mathbf{v}_{B_2}) + \sum_{e' \in N_{e_t}} l(-\mathbf{u}_{e'_t}^{\mathsf{T}} \mathbf{v}_{B_2}))}_{\text{Cross-architecture loss}} \tag{3}$$

In summary, for a masked instruction e_t in B_1, we use the rest instructions in B_1 as well as all the instructions in B_2 to predict e_t and vice-versa, as shown in Figure 3. In this example, the masked instruction e_t is `cmp eax, 0x2` in B_1. For the mono-architecture objective, we use the rest instructions in B_1 (colored in grey) to predict e_t. For the cross-architecture objective, we use all instructions in B_2 (colored in blue) to predict e_t. By combining the two objectives, we train `CrossIns2Vec` to learn CAIE, such that similar instructions, regardless of their ISAs, tend to have close embeddings in a shared vector space.

5 Evaluation

We evaluate our model in terms of the quality and transferability of CAIE. We have two questions: (***Q1***) ***Quality:*** how well can CAIE tolerate architectural differences and capture code semantics across ISAs? (***Q2***) ***Transferability:*** whether CAIE can transfer knowledge from one ISA to another? To answer ***Q1***, we conduct the intrinsic evaluation, including the instruction similarity task. To answer ***Q2***, we conduct the extrinsic evaluation, including two critical binary analysis tasks: function similarity comparison and malware detection.

5.1 Experimental Settings

Building Datasets for Learning CAIE. We consider four ISAs: x86, ARM, MIPS, and PowerPC (PPC). We consider x86 as the high-resource ISA, and the other ISAs as the low-resource ISAs.[2] We first collect various programs, including *OpenSSL-1.1.1*, *Binutils-2.34*, *Curl-7.87*, *Findutils-4.8.0*, *gmp-6.2.0*, *Libgpg-error-1.45*, and *Zlib-1.2.11*. These programs are widely used in prior NLP-based binary code analysis works (Luo et al., 2023; Ding et al., 2019; Marcelli et al., 2022; Massarelli et al., 2019; Li et al., 2021). For each program, we compile it on the four ISAs using different optimization levels (O0-O3).

Given a pair of ISAs (one is x86 and another a low-resource ISA), we build the dataset comprising semantically-equivalent basic block pairs for learning CAIE (the details of how to collect such pairs are discussed in Section 4.1). Through this, we have three datasets: $D_{x86 \leftrightarrow ARM}$ contains 2,058,484 semantically-equivalent basic block pairs between x86 and ARM; $D_{x86 \leftrightarrow MIPS}$ contains 2,121,125 semantically-equivalent basic block pairs between x86 and MIPS; and $D_{x86 \leftrightarrow PPC}$ contains 2,189,139 semantically-equivalent basic block pairs between x86 and PPC.

Subsequently, we use each of the three datasets to train `CrossIns2Vec` to learn CAIE for the instructions across the respective pair of ISAs.

Note that in our evaluation, the datasets used for learning CAIE have *no overlap* with the testing datasets used in the downstream tasks, the details

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

of which are introduced in Sections 5.3 and 5.4.

Baseline Method. To the best of our knowledge, `UniMap` (Wang et al., 2023) is the *only work* that shares the same goal as ours: focusing on learning CAIE to tackle the data scarcity issue. We thus consider it as the baseline method. `UniMap` employs an unsupervised learning approach, while our approach relies on supervised learning. As we discussed in Section 2.2, in the context of binary code analysis, obtaining cross-architecture signals is not challenging due to the prevalence of cross-compilation. Although the collection of parallel data demands increased engineering efforts, the additional efforts prove their worth if the learned CAIE exhibits superior transferability.

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.

5.2 Instruction Similarity Task

This task is to evaluate whether CAIE can tolerate the syntactic differences and capture the semantic information of instructions across ISAs. To evaluate this, we measure whether two semantically-similar instructions, *regardless of their ISAs*, have close embeddings. Unlike word embeddings, which have many existing corpora for evaluation, we do not have such data. We thus create the datasets ourselves, which contain manually-labeled instruction pairs. We rely on the assembly language references (x86, 2023; ARM, 2023; MIPS, 2023; PowerPC, 2023) to create our datasets.

Similar to `UniMap`, we categorize instructions into 6 categories, including data transfer, arithmetic, logical, shift/rotate, bit/byte, and control transfer. Note that this categorization serves the purpose for conducting the instruction similarity task. Thus, when preparing the datasets, we encompass instructions from all categories that are shared across the four ISAs (x86, ARM, MIPS, and PPC). For each category, we randomly select 40 x86 instructions. 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., performing the same operation). Finally, we create three datasets: D_1 contains 240 similar and 240 dissimilar pairs of x86↔ARM instructions; D_2 contains the same number of pairs of x86↔MIPS instructions; and D_3 contains the same number of pairs of x86↔PPC instructions. Given a pair of instructions, we calculate the cosine similarity of their CAIE to measure their similarity. For D_1, D_2, and D_3, we achieve AUC = 0.78, 0.73, and 0.74, respectively.

Comparison with Baseline Method. To compare with the baseline `UniMap`, we use the same datasets created by `UniMap`, which contain 120 similar and 120 dissimilar pairs of x86↔ARM instructions, as well as the same number of similar and dissimilar pairs of x86↔MIPS and x86↔PPC instructions. We achieve higher AUCs of 0.79, 0.71, 0.72 for x86↔ARM, x86↔MIPS, and x86↔PPC, respectively, while the prior work yields 0.76, 0.66, and 0.68. Thus, our model exhibits better capabilities of learning CAIE that excel in capturing the semantic relations among various ISAs.

Nearest Neighbor Instructions. We next examine, for a given x86 instruction, its top-K similar instructions in the other ISAs. We first select eight high-frequency x86 instructions from all the six categories. For each x86 instruction, we search for the top-two similar instructions in ARM, MIPS, and PPC, respectively, based on the cosine similarity of their CAIE. The results are shown in Table 1. We can see that for a given x86 instruction, its top-two similar instructions in the other ISAs share similar semantics, as predicted. For example, for the x86 instruction `ADC RDX,R11`, we find the relevant ARM instructions `ADC R11,R7,R3` and `ADDS R10,R6,R11`, MIPS instructions `ADDU R6,R3,R4` and `ADDU R2,R3,R16`, and PPC instruction `ADDE R23,R8,R10` and `ADDC R7,R7,R28`, where all of them add the values in two operands and store the result back in the destination operand.

5.3 Function Similarity Detection Task

The extrinsic evaluation is to evaluate the transferability of CAIE. We conduct two binary analysis tasks: function similarity detection and malware detection. This section presents the result of the first task. For each task, we train a model using the task-specific data on x86, and transfer the trained model on another ISA (e.g., ARM, MIPS, and PPC).

FunGnn Model. `FuncGNN` (Nair et al., 2020) is a graph neural network trained on labeled control flow graph (CFG) pairs to measure the function similarity. To evaluate the transferability of CAIE, we modify the input layer of `FuncGNN` to encode each instruction as its CAIE. We then train `FuncGNN`

Table 1: Nearest neighbor instructions *cross-architecturally* as measured by cosine similarity of CAIE. The top two similar ARM, MIPS, and PPC are shown for each of the eight x86 instructions randomly selected from the six categories of instructions.

	MOV R15D,[R9+0]	*Score*	**ADC RDX,R11**	*Score*	**SAR EDX,CL**	*Score*	**LEA R14,[RSP+0+<TAG>]**	*Score*
ARM	MOVNE R9,R3	0.59	ADC R11,R7,R3	0.80	ASRS R3,R5	0.69	ADD R9,SP,0	0.53
	MOVLT R0,R4	0.58	ADDS R10,R6,R11	0.80	ASRS R0,R1	0.64	MOV R8,SP	0.51
MIPS	MOVN R16,R5,R16	0.56	ADDU R6,R3,R4	0.59	SRAV R3,R4	0.61	ADDIU R19,R29,0	0.62
	MOVE R15,R16	0.55	ADDU R2,R3,R16	0.52	SRL R17,R16,0	0.56	ADDIU R21,R29,0	0.58
PPC	MR R21,R4	0.76	ADDE R23,R8,R10	0.78	SRAW R9,R8,R9	0.71	ADDI R29,R1,0	0.55
	MR R23,R4	0.72	ADDC R7,R7,R28	0.78	SRAW R8,R21,R9	0.63	LWZ R29,<OFF>R24	0.52
	AND ECX,EAX	*Score*	**XOR ECX,EAX**	*Score*	**JMP DEF_<TAG>**	*Score*	**SHL EAX,CL**	*Score*
ARM	ANDS R2,R1	0.69	EORS R2,R3	0.51	B DEF_<TAG>	0.49	LSL R3,R3,LR	0.62
	AND R3,R9,R3	0.63	EOR LR,R5,R6	0.48	LDR R3,[R8,0]	0.49	ASR R9,R9,R4	0.61
MIPS	AND R6,R3	0.69	XOR R3,R2	0.69	B DEF_<TAG>	0.50	SLL R3,R16,R5	0.60
	AND R10,R6	0.69	XOR R4,R3	0.66	SUBU R5,R17,R30	0.43	SRA R20,R7	0.59
PPC	AND R3,R10,R3	0.69	XOR R6,R6,R9	0.63	B DEF_<TAG>	0.60	SLW R9,R28,R9	0.59
	AND R8,R9,R8	0.70	XOR R10,R10,R9	0.62	LWZ R12,<OFF>R23	0.46	SLW R8,R8,R5	0.58

Table 2: Results of function similarity detection task.

Train	Test	CrossIns2Vec (***Ours***)			UniMap (***Baseline***)		
		AUC	Prec.	Recall	AUC	Prec.	Recall
x86	ARM	0.99	0.95	0.98	0.95	0.93	0.95
	MIPS	0.99	0.97	0.98	0.93	0.90	0.93
	PPC	0.98	0.93	0.97	0.94	0.90	0.91

on x86 and transfer the trained model to test data in ARM, MIPS, and PPC, respectively.

Task-Specific Datasets. We first build the task-specific training dataset on x86, containing 50,000 similar and 50,000 dissimilar x86 function pairs. We then build the testing datasets for ARM, MIPS and PPC, each containing 5,000 similar and 5,000 dissimilar function pairs in the corresponding ISA. To ensure *no overlap* between the training and testing datasets, we select different programs to build them: (1) *OpenSSL-1.1.1*, *Binutils-2.34*, *Curl-7.87*, *Findutils-4.8.0*, *gmp-6.2.0*, *Libgpg-error-1.45*, and *Zlib-1.2.11* are used to build the training dataset; (2) *Coreutils-9.0* and *Diffutils-3.7* are used to build the testing datasets.

Following the dataset building method in InnerEye (Zuo et al., 2019), we consider two functions similar if they are compiled from the same piece of source code, and dissimilar if their source code is different. Each program is compiled using four optimization levels (O0-O3). For a given piece of source code, by applying different optimization levels, we can find six similar pairs. Then, the similar and dissimilar function pairs in the training and testing datasets are evenly divided among the six possible pairs of optimization levels.

Results. Table 2 shows the performance results, including AUC, precision, and recall. We can observe that when the model trained on x86 is transferred to ARM, MIPS, and PPC, it achieves AUC values of 0.99, 0.99, and 0.98, respectively. The results show that the model achieves exceptional performance when transferred from x86 to the other ISAs, demonstrating the superior transferability of CAIE.

Comparison with Baseline Method. To compare with the baseline UniMap, we first modify the input layer of FuncGNN, such that the CAIE generated by UniMap are used to encode each instruction. We then use the same training dataset to train FuncGNN on x86. Finally, we transfer the trained model to perform testing on ARM, MIPS, and PPC, respectively. The testing datasets are the same as those used for evaluating the transferability of CAIE generated by CrossIns2Vec.

The results are shown in Table 2. We observe that when the model trained on x86 is transferred to ARM, MIPS, and PPC, it achieves lower AUC/precision/recall values than those obtained when employing the CAIE generated by our model CrossIns2Vec. This demonstrates that the CAIE learned by CrossIns2Vec exhibits better transferability compared to UniMap.

5.4 Malware Detection Task

LSTM Model. We use the Long Short Term Memory (LSTM) model (HaddadPajouh et al., 2018) to detect malware. We modify the input layer of LSTM to encode each instruction as its corresponding CAIE. We then train LSTM on x86 and transfer the model to ARM, MIPS, and PPC.

Task-Specific Datasets. We first collect malware samples from *VirusShare.com* (virusShare, 2023), and then deduplicate the collected samples to elim-

Table 3: Results of malware detection task.

Train	Test	CrossIns2Vec (***Ours***)			UniMap (***Baseline***)		
		AUC	Prec.	Recall	AUC	Prec.	Recall
x86	ARM	0.94	0.91	0.94	0.93	0.89	0.93
	MIPS	0.93	0.91	0.94	0.91	0.90	0.92
	PPC	0.95	0.90	0.98	0.91	0.89	0.92

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

Train	Training Size	Test	Testing Size	AUC
PPC	300(M) + 300(B)	PPC	200(M) + 200(B)	0.87
x86	300(M) + 300(B)	PPC	200(M) + 200(B)	0.78
	600(M) + 600(B)			0.85
	900(M) + 900(B)			0.90
	2000(M) + 2000(B)			0.91

inate redundant ones. As a result, we have 2000, 1100, 1000, and 500 samples in x86, ARM, MIPS, and PPC, respectively. It should be noted that we spent a lot of efforts in collecting malware samples in MIPS and PPC, which are considered as low-resource ISAs.

We then build the task-specific training and testing datasets. The x86 malware samples are used for training. For the other ISAs, the malware samples are used for testing. In each training and testing dataset, we include the same number of benign samples. In the training dataset, the benign samples are randomly selected from *OpenSSL-1.1.1*, *Binutils-2.34*, *Curl-7.87*, *Findutils-4.8.0*, *gmp-6.2.0*, *Libgpg-error-1.45*, and *Zlib-1.2.11*, while the testing dataset contains benign samples selected from different programs, including *Coreutils-9.0* and *Diffutils-3.7*. We ensure no overlap between the training and testing datasets.

Results. Table 3 shows the performance results, including AUC, precision, and recall. We can see that when the model trained on x86 is transferred to ARM, MIPS, and PPC, it achieves AUC values of 0.94, 0.92, and 0.91, respectively. The fact that the model's accuracies keep high demonstrates the efficacy of our learned CAIE in facilitating the transfer of knowledge across ISAs.

We then 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. The results are shown in Table 4. We can see that when the model is trained on an x86 training dataset containing more than 900 malware samples and then reused for PPC, it outperforms the 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. However, for low-resource ISAs like PPC, acquiring a large dataset proves to be challenging.

Comparison with Baseline Method. We first modify the input layer of LSTM, such that the CAIE generated by UniMap are used to encode each instruction. We then use the same training datasets to train LSTM on x86. Finally, we transfer the trained model to perform prediction on the same testing datasets on ARM, MIPS, and PPC, respectively. The results are shown in Table 3. When comparing the AUC/precision/recall values obtained when employing CAIE learned by UniMap to those leaned by CrossIns2Vec, it demonstrates that our learned CAIE have superior transferability compared to the baseline UniMap.

6 Conclusion

Applying deep learning to binary code analysis has drawn great attention. Limited availability of data on low-resource ISAs, however, hinders deep learning-based binary code analysis. In this work, we propose to learn *cross-architecture instruction embeddings (CAIE)*, where semantically-similar instructions, *regardless of their ISAs*, have close embeddings in a shared space. As a result, we can transfer a model trained on a data-rich ISA to another ISA with less available data. We conducted experiments to evaluate the quality and transferability of the learned CAIE. In the downstream tasks, when a model trained on x86 is transferred to ARM, MIPS and PPC, the prediction accuracies keep high. Our approach significantly outperforms the prior work. Therefore, our approach can generate CAIE with high quality and transferability, and resolve the data scarcity problem in low-resource ISAs for binary code analysis tasks.

Acknowledgments

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

Ethical Considerations

Datasets. To train our model `CrossIns2Vec`, we first need to collect semantically-equivalent basic block pairs from different ISAs. We first collect open-source programs, and compile them for different ISAs using cross compilers. Given the wide availability of open-source code, this requires little effort. To determine the ground truth regarding the similarity of basic blocks, we rely on the source code line number. Specifically, if two basic blocks from different ISAs have the *same starting* and *end* source code line numbers, they are considered to be semantically-equivalent. The detailed description of the process can be found in Section 4.1.

For training `CrossIns2Vec`, we use a dataset of semantically-equivalent basic block pairs. We acknowledge that aggressive optimizations, such as inlining in O3, have an impact for searching basic block pairs. However, we clarify that we skip including basic blocks that involve inlining into our datasets. Given the large number of basic blocks available, this does not impose a barrier for creating the datasets for training `CrossIns2Vec`.

For each downstream task, we collect the task-specific training and testing datasets, the details of which are introduced in Sections 5.3 and 5.4. A special note is about malware samples, which are collected from *VirusShare.com* (virusShare, 2023). *VirusShare.com* is a repository of malware samples that researchers use to study and develop cybersecurity solutions. While it can be a valuable resource, there are ethical considerations, including using the samples responsibly for legitimate research purposes, preventing the creation of new threats, and respecting privacy and legal boundaries.

In our efforts to support subsequent research, we plan to make the datasets available for public use. Specifically, datasets obtained using open-source programs will be openly released. However, in the case of malware samples, we will provide the file names and hash values sourced from *VirusShare.com*. This offers researchers the means to identify specific malware samples without directly sharing the potentially harmful code.

Applications. To cope with the data scarcity issue and alleviate the per-ISA effort, this work proposes to learn cross-architecture instruction embeddings (CAIE), where semantically-similar instructions, regardless of their ISAs, have close embeddings in a shared vector space. Enabled by the technique, we can train *a single model* on a high-resource ISA and *reuse* it for low-resource ISAs, without any modification. Compared to existing methods, this work offers significant advantages by eliminating the need for data collection in multiple ISAs (particularly for low-resource ISAs where labeled data is limited or unavailable) as well as the per-ISA fine tuning efforts. It will not only advance binary code analysis by developing a bridge for enabling model reuse, but also have various security applications, including malware detection and function similarity comparison.

Limitations

NLP-inspired binary code analysis is a promising research direction, but not all NLP techniques are applicable to binary code analysis. Thus, studies like ours that identify and examine effective NLP techniques for binary code analysis are valuable in advancing exploration along this direction.

To validate the effectiveness of our approach, we conducted two downstream tasks to evaluate the transferability of the learned CAIE. We acknowledge that the learned CAIE may not be generalize to all types of code, such as Windows, iPhone, and Android applications. To ascertain this, further investigation and comprehensive testing are needed.

Due to the extensive range of binary analysis tasks and their inherent complexity, we do not claim that our approach can be applied to all tasks. However, the successful performance of our approach in two critical tasks highlights the significant value of CAIE, while demonstrating the application of CAIE for other tasks needs dedicated future work. Much research can be done for exploring and expanding the boundaries of the approach.

References

Iftakhar Ahmad and Lannan Luo. 2023. Unsupervised binary code translation with application to code clone detection and vulnerability discovery. In *Findings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning (ICML)*.

ARM. 2023. Arm instruction reference. http://infocenter.arm.com/help/

index.jsp?topic=/com.arm.doc.dui0068b/CIHEDHIF.html.

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

Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM.

Fuxiang Chen, Mijung Kim, and Jaegul Choo. 2021. Novel natural language summarization of program code via leveraging multiple input representations. In *Findings of the Association for Computational Linguistics: EMNLP 2021*.

Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

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

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.

Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium (NDSS)*.

Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. discovre: Efficient cross-architecture identification of bugs in binary code. In *Network and distributed system security symposium (NDSS)*.

Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM.

Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, 85:88–96.

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

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. In *The Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI)*.

IDA. 2023. IDA Pro. https://hex-rays.com/ida-pro/.

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

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)*.

Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu. 2023. Codeie: Large code generation models are better few-shot information extractors. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Xuezixiang Li, Qu Yu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceeding of the ACM Conference on Computer and Communications Security (CCS)*.

LLVM. 2023. The llvm compiler infrastructure. https://llvm.org.

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

Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. 2016. Repackage-proofing android apps. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE.

Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM.

Lannan Luo and Qiang Zeng. 2016. Solminer: mining distinct solutions in programs. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM.

Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM.

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

Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on iot cloud platforms. In *Annual Computer Security Applications Conference (ACSAC)*.

Zhenhao Luo, Baosheng Wang, Yong Tang, and Wei Xie. 2019b. Semantic-based representation binary clone detection for cross-architectures in the internet of things. *Applied Sciences*, 9(16):3283.

Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *The Network and Distributed System Security Symposium (NDSS)*.

Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *USENIX Security Symposium (USENIX Security)*.

Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

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

MIPS. 2023. Mips opcode and instruction reference home. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf.

Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcgnn: A graph neural network approach to program similarity. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring api embedding for api usages and applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE.

Heewan Park, Seokwoo Choi, Hyun-il Lim, and Taisook Han. 2008. Detecting code theft via a static instruction trace birthmark for java methods. In *IEEE International Conference on Industrial Informatics*. IEEE.

Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy (SP)*. IEEE.

PowerPC. 2023. Powerpc opcode and instruction reference home. http://math-atlas.sourceforge.net/devel/assembly/ppc_isa.pdf.

QEMU. 2023. Qemu: A generic and open source machine emulator and virtualizer. https://www.qemu.org.

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

Sebastian Ruder, Ivan Vulić, and Anders Søgaard. 2019. A survey of cross-lingual word embedding models. *Journal of Artificial Intelligence Research*.

Ali Sabet, Prakhar Gupta, Jean-Baptiste Cordonnier, Robert West, and Martin Jaggi. 2019. Robust cross-lingual embeddings from parallel sentences. *arXiv preprint arXiv:1912.12481*.

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

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE.

Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2004. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology*.

Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. 2014. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Zhenzhou Tian, Qinghua Zheng, Ting Liu, and Ming Fan. 2013. DKISB: Dynamic key instruction sequence birthmark for software plagiarism detection. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE.

Shyam Upadhyay, Manaal Faruqui, Chris Dyer, and Dan Roth. 2016. Cross-lingual models of word embeddings: An empirical comparison. *arXiv preprint arXiv:1604.00425*.

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

virusShare. 2023. open repository of malware samples. https://virusshare.com/.

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

Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. 2023. Can a deep learning model for one architecture be used for others? Retargeted-Architecture binary code analysis. In *32nd USENIX Security Symposium (USENIX Security)*.

Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*. ACM.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems (NeurIPS)*.

John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2015. Towards universal paraphrastic sentence embeddings. *arXiv preprint:1511.08198*.

x86. 2023. x86 opcode and instruction reference home. http://ref.x86asm.net/coder32.html.

Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (CCS)*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Qiang Zeng, Golam Kayas, Emil Mohammed, Lannan Luo, Xiaojiang Du, and Junghwan Rhee. 2019a. Heaptherapy+: Efficient handling of (almost) all heap vulnerabilities using targeted calling-context encoding. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. ACM.

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

Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Network and Distributed System Security Symposium (NDSS)*.