A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis

Kimberly Redmond Lannan Luo Qiang Zeng
University of South Carolina
redmonkm@email.sc.edu, {lluo, zeng1}@cse.sc.edu

***Abstract*—Given a closed-source program, such as most of proprietary software and viruses, binary code analysis is indispensable for many tasks, such as code plagiarism detection and malware analysis. Today, source code is very often *compiled for various architectures*, making cross-architecture binary code analysis increasingly important. A binary, after being disassembled, is expressed in an assembly language. Thus, recent work starts exploring Natural Language Processing (NLP) inspired binary code analysis. In NLP, words are usually represented in high-dimensional vectors (i.e., embeddings) to facilitate further processing, which is one of the most common and critical steps in many NLP tasks. We regard *instructions as words* in NLP-inspired binary code analysis, and aim to represent instructions as embeddings as well.**

To facilitate cross-architecture binary code analysis, our goal is that similar instructions, *regardless of their architectures*, have embeddings close to each other. To this end, we propose a *joint learning* approach to generating instruction embeddings that capture not only the semantics of instructions within an architecture, but also their *semantic relationships across architectures*. To the best of our knowledge, this is the first work on building cross-architecture instruction embedding model. As a showcase, we apply the model to resolving one of the most fundamental problems for binary code similarity comparison—semantics-based basic block comparison, and the solution outperforms the code statistics based approach. It demonstrates that it is promising to apply the model to other cross-architecture binary code analysis tasks.

I. Introduction

When the source code of programs is not available, binary code analysis becomes indispensable for a variety of important tasks, such as plagiarism detection [41], [31], malware classification [66], [30], and vulnerability discovery [53], [62]. Increasingly, software is *cross-compiled for various architectures*. For example, hardware vendors often use the same code base to compile firmware for different devices that operate on varying architectures (e.g., x86 and ARM): this could cause a single vulnerability at source-code level to spread across binaries across diverse devices. As a result, *cross-architecture binary code analysis* has become an emerging problem that draws great attention [53], [19], [21], [62], [67]. Analysis of binaries across instruction set architectures (ISAs), however, is non-trivial: binaries of varying ISAs differ greatly in instruction sets; calling conventions; general- and special- purpose CPU register usages; and memory addressing modes.

A binary, after being disassembled, is expressed in an assembly language. Given this insight, binary code analysis can be approached by borrowing ideas and techniques of Natural Language Processing (NLP)—a rich area focused on processing texts from various natural languages [9]. In many NLP tasks, words are first often converted into *word embeddings* (i.e., high-dimensional vectors) to facilitate further processing [60], [56]. A word's embedding is able to capture the contextual semantic meaning of the word; thus, words that have similar contexts have embeddings that appear close together in the high-dimensional space [46].

We regard *instructions as words* in NLP-inspired binary code analysis, and thus aim to represent instructions as embeddings as well. To facilitate cross-architecture binary code analysis, our goal is that similar instructions, *regardless of their architectures*, have embeddings that are close in the high dimensional space. Specifically, we aim to learn semantic features for each instruction, such that instructions *in one architecture* with similar semantics are assigned similar vector representations (the **mono-architecture** objective); and additionally, instructions *across different architectures* with similar semantics have similar vector representations (the **cross-architecture** objective). We call such vector representations *cross-architecture instruction embeddings*.

Why Cross-Architecture Instruction Embeddings? The cross-architecture instruction embeddings capture semantic relations of instructions across architectures, and keep invariant among tasks. Thus, it has many potential applications to cross-architecture binary code analysis. Take the search of semantically equivalent functions as an example. Given a function in x86 that contains, for instance, the *Heartbleed* function, by searching for functions similar to it from a large database of functions of varying architectures, more vulnerability instances may be found. This question has gained intense research interest [53], [19], [21], [62]. A core subtask involved is the comparison of basic blocks across architectures. We will show how the proposed technique can be applied to resolving this subtask.

Moreover, many deep learning based NLP techniques take word embedding as inputs. Following the idea of NLP-inspired binary code analysis, the proposed instruction embedding model can be applied to, e.g., classifying binaries across architectures by feeding the instruction embeddings of the binaries into classic neural network structures that are used for classifying texts in NLP [37], [64], [58].

Our Approach. We propose to learn the cross-architecture instruction embedding through a *joint learning* approach. Specifically, our joint model utilizes both the context concurrence

information present in the instruction sequences from the same architecture, and the semantically-equivalent signals exhibited in the instruction sequence pairs from different architectures. By jointly learning these two types of information, our model can achieve both the *mono-architecture* and *cross-architecture* objectives, and generate high-quality cross-architecture instruction embeddings that capture not only the semantics of instructions within an architecture, but also their *semantic relationships across architectures*.

We have implemented the novel cross-architecture instruction embedding model, and conducted a series of experiments to evaluate the quality of the learned instruction embeddings. Moreover, as a showcase, we apply the model to resolving one of the most fundamental problems for binary code similarity comparison, that is, semantics-based basic block similarity comparison. Our solution achieves AUC = 0.90. Recent work [19], [21], [62] uses several manually selected statistic features (such as the number of instructions and the number of constants) of a basic block to represent it. However, a SVM classifier based on such features only achieves AUC = 0.85 for the same task. The trained models, datasets, and evaluation results are publicly available.[1]

We summarize our contributions as follows:

- To the best of our knowledge, this is the *first* work on building a uniform cross-architecture instruction embedding model that tolerates the significant syntactic differences across architectures.
- We propose an effective joint learning approach to training the model, which makes use of both the information in the instruction sequences from the same architecture, and the semantically-equivalent signals exhibited in the instruction sequence pairs from different architectures.
- We implement model, and the evaluation demonstrates the good quality of the learned instruction embeddings. Moreover, we apply the model to cross-architecture basic block similarity comparison and the solution outperforms the statistic feature based approach.
- This research successfully demonstrates that it is promising to adapt NLP ideas and techniques to binary code analysis tasks. Just like word embeddings are critical for many NLP tasks, the proposed instruction embedding model can substantially facilitate NLP-inspired cross-architecture binary code analysis.

II. Related Work

We expect the proposed model can be naturally applied to binary code similarity comparison. Existing binary code analysis techniques for code similarity comparison can be roughly divided into two classes: traditional approaches and machine learning-based ones.

Traditional approaches. Most traditional approaches work on a *single* architecture. First, static plagiarism detection or clone detection includes string-based [2], [5], [15], AST-based [32], [57], [63], [36], token-based [33], [55], [54], and PDG-based approaches [22], [40], [11], [39]. *Source code-based* approaches are inapplicable for closed-source software. Symbolic execution of binary code has been enabled by tools such as BitBlaze [6] and BAP [3]; it is accurate in extracting code semantics [42], but is very *computationally expensive* and *unscalable*.

Recent works have applied traditional approaches to addressing the *cross-architecture* scenario [53], [19], [8], [20], [13], [14], [12]. Multi-MH and Multi-k-MH [53] are the first two methods for comparing functions of different ISAs. But their fuzzing-based basic-block similarity comparison and graph (i.e., CFG) matching-based algorithms are very expensive. discovRE [19] boosts CFG-based matching process, but is still expensive. Both `Esh` [12] and its successor [13] use data-flow slices of basic blocks as the basic comparable unit. `Esh` uses SMT solver to verify function similarity, which makes it unscalable. In [13], binaries are lifted to IR for creating function-centric signatures.

Machine learning based approaches. Machine learning, including deep learning, has been applied to code analysis [10], [38], [21], [62], [50], [43], [51], [29], [59], [28], [52], [27], [17]. Lee et al. propose `Instruction2vec` for converting assembly instructions to vector representations [38]; but their instruction embedding model can only work on a *single* architecture. `Asm2Vec` [17] produces a numeric vector for each function, but can only work on a *single* architecture. We instead build a *cross-architecture instruction embedding model* which works for varying architectures.

A few works target *cross-architecture* binary code analysis [21], [62], [8], [67]. Some exploit the *statistical* aspects of code, rather than its semantics. For example, Genius [21] and Gemini [62] use some *manually* selected statistical features (e.g., the number of constants) to represent basic blocks, but they ignore *the meaning of instructions and the dependency between them*, resulting in significant loss of semantic information. `INNEREYE-BB` [67] uses LSTM to encode each basic block into an embedding, but it needs to train a *separate* instruction embedding model for each architecture. Instead, we build a uniform cross-architecture instruction embedding model that tolerates the syntactic differences across architectures.

Summary. To the best of our knowledge, ours is the *first work* to learn cross-architecture instruction embeddings that capture semantic features *invariant to specific tasks*. Such instruction embeddings can be adopted to a variety of important code analysis tasks, and help us scale to more architectures.

III. Background

A. Word Embeddings

Many NLP models applying deep learning techniques have been proposed to learn high-quality word embeddings, with Mikolov's *skip-gram* (SG) and *Continuous Bag Of Words* (CBOW) [46] gaining a lot of traction due to their relatively low memory use, and overall increased efficiency.

The SG model takes each word as the input and predicts the context corresponding to the word, while the CBOW model takes the context of each word as the input and predicts the word corresponding to the context. During training, a sliding window is applied on a text. Each model starts with a random vector for each word, and then gets trained when going over each sliding window. After the model is trained, the embeddings

[1] https://github.com/nlp-code-analysis/cross-arch-instr-model

of each word become meaningful, yielding similar vectors for similar words. Due to their simplicity, both models can achieve very good performances on various semantic tasks, and can be trained on a desktop computer at billions of words per hour.

B. Multilingual Word Embeddings

A wide variety of multilingual NLP tasks, including machine translation [7], entity clustering [26], and multilingual document classification [4], have motivated recent work in training *multilingual word representations* where similar-meaning words in different human languages are embedded close together in the same high-dimensional space.

Different approaches have been proposed to training multilingual word embeddings. For example, one category of approaches is based on *multilingual mapping*, where word embeddings are first trained on each language independently and a mapping is then learned to transform word embeddings from one language to another [47]. Another category attempts to *jointly learn* multilingual word embeddings from scratch [34], [35], [25], [44]. Our cross-architecture instruction embedding model adapts the technique proposed in [44].

IV. Cross-Architecture Instruction Embedding Model

In light of the idea of NLP-inspired binary code analysis, we regard *instructions as words*. An instruction includes an opcode (specifying the instruction operation) and zero or more operands (specifying registers, memory locations, or literal data). For example, `mov ebp, esp` is an instruction where `mov` is an opcode and both `ebp` and `esp` are operands. Note that the assembly code in this paper adopts the Intel syntax.

A. Design Goal

Our goal in building the instructions model is to achieve both the *mono-architecture* and *cross-architecture* objectives. That is, we want the learned cross-architecture instruction embeddings not only to preserve the clustering properties mono-architecturally (instructions in one architecture with similar semantics are close together in the vector space), but also exhibit the semantic relationships across different architectures (instructions across architectures with similar semantics are close together).

B. System Overview

Our proposed cross-architecture instruction embedding model adapts the *joint learning* approach in [44], consisting of a *mono-architecture* component and a *multi-architecture* component. The *mono-architecture* component utilizes the context concurrence information present in the input instruction sequences from the same architecture (Section IV-C); and the *multi-architecture* component learns the semantically-equivalent signals exhibited in the equivalent instruction sequence pairs from varying architectures (Section IV-D).

An *instruction sequence* in our work is a basic block, as we regard instructions as words and basic blocks as sentences. Note that we do not consider a function as a sentence, as a function cannot be treated as a straight-line sequence: when a function is invoked, its instructions are not executed sequentially.

Handling out-of-vocabulary (OOV) instructions. The issue of OOV words is a well-known problem in NLP, and it exacerbates significantly in our case as constants, address offsets, labels, and strings are frequently used in instructions. To address it, instructions are preprocessed using the following rules: (1) Numerical constant values are replaced with 0, and the minus signs are preserved. (2) String literals are replaced with `<STR>`. (3) Function names are replaced with `FOO`. (4) Other symbol constants are replaced with `<TAG>`.

Joint objective function. Below is our *joint objective* function:

$$J = \gamma \sum_{i=1}^{N} J(\texttt{Mono}_{a_i}) + \beta \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} J(\texttt{Multi}_{<a_i,a_j>}) \quad (1)$$

In this equation, each mono-architecture component, $\texttt{Mono}_{a_i}$ ($\forall i \in \{1, ...N\}$) aims to capture the clustering property of the corresponding architecture a_i, where $J(\texttt{Mono}_{a_i})$ is the objective function of $\texttt{Mono}_{a_i}$. Each multi-architecture component, $\texttt{Multi}_{<a_i,a_j>}$ ($\forall i, j \in \{1, ...N\}$, $i \neq j$), is used to learn the semantic relationships across architectures, where $J(\texttt{Multi}_{<a_i,a_j>})$ is the objective function of $\texttt{Multi}_{<a_i,a_j>}$. The γ and β hyperparameters balance out the influence of the mono-architecture components over the multi-architecture one.

Specifically, if there are only two architectures, e.g., x86 and ARM, the *joint objective* function becomes:

$$J = \gamma(J(\texttt{Mono}_{x86}) + J(\texttt{Mono}_{ARM})) + \beta J(\texttt{Multi}_{<x86,ARM>}) \quad (2)$$

C. Mono-Architecture Component

Any word embedding model can be a candidate to be selected to build the mono-architecture component [46], [34], [25], [48], [49], [16]. Based on our experiment, we adopt the CBOW model as implemented in `word2vec` [46], which achieves better performance than the skip-gram model.

The CBOW model predicts a current instruction based on its context. During training, a sliding window with size n is employed on an instruction sequence. The context of a current instruction e_t is defined as n instructions before and after e_t within the corresponding sliding window. The CBOW model contains three layers. The input layer corresponds to the context. The hidden layer corresponds to the projection of each instruction from the input layer into the weight matrix, which is then projected into the third output layer. The final step is the comparison between the output and the current instruction in order to correct its vector representation based on the back propagation of the error gradient. Thus, the *objective* of the CBOW model is to maximize the following equation:

$$J = \frac{1}{T} \sum_{t=1}^{T} \log P(e_t | e_{t-n}, ...e_{t-1}, e_{t+1}, ..., e_{t+n}) \quad (3)$$

where T is the length of the instruction sequence, and n the size of the sliding window.

After the model is trained on many sliding windows, similar instructions tend to have embeddings that appear close together in the high-dimensional vector space.

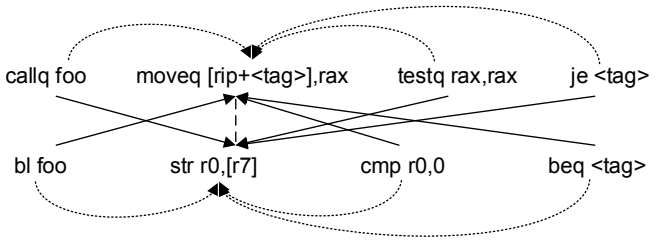

Fig. 1: A cross-architecture instruction embedding model.

D. Multi-Architecture Component

We adopt the CBOW model to build our multi-architecture component. Our cross-architecture instruction embedding model is extended from the CBOW model as implemented in `word2vec`, and is effective in learning instruction representations both mono-architecturally and multi-architecturally.

Figure 1 shows how our cross-architecture instruction embedding model works. The input is a pair of semantically-equivalent basic blocks, each of which is a sequence of instructions: the instruction sequence of the basic block compiled for x86 is {`callq foo`; `moveq [rip+<tag>],rax`; `testq rax,rax`; `je <tag>`}, and the instruction sequence of the block compiled for ARM is {`bl foo`; `str r0,[r7]`; `cmp r0,0`; `beq <tag>`}. Note that the instruction sequences may be of different lengths, and the lengths can vary from example to example; both can be easily handled by the model.

To predict instructions cross-architecturally rather than *only* mono-architecturally as in the standard CBOW model (Section IV-C), we use the contexts in one architecture to predict the instructions in another architecture. For example, if we know that the instruction `moveq [rip+<tag>],rax` in x86 has the same meaning as the instruction `str r0,[r7]` in ARM, we can simply substitute `str r0,[r7]` and use the surrounding instructions—such as `bl foo` and `cmp r0,0`—to predict `moveq [rip+<tag>],rax`. Therefore, given an alignment link between an instruction e_1 in one architecture and an instruction e_2 in another architecture, our cross-architecture model uses the neighbors of the instruction e_2 to predict the instruction e_1, and vice versa. Then, $J(\texttt{Multi}_{<a_i,a_j>})$ in Equation 1, where a_i and a_j represent two different architectures, is also the *objective* function in Equation 3.

The challenge here is how to find the alignment links between instructions. There are two solutions. (1) A simple way is to assume linear alignments between instructions across architectures. That is, each instruction in one sequence M at position i is aligned to the instruction in another sequence N at position $i \times |N|/|M|$, where $|M|$ and $|N|$ are the length of the corresponding sequences. (2) Another way is to determine the alignment links based on the *opcode* contained in each instruction. For example, from the opcode references of x86 [61] and ARM [1], we can find that `moveq` from x86 and `str` from ARM can be used to store data in registers; thus, it is reasonable to align an instruction containing `moveq` with another instruction containing `str`. Then, a dynamic programming algorithm similar to the solution to finding the Longest Common Subsequence can be used to determine the best alignment between two sequences.

We adopt the first solution in our current implementation. Our preliminary results show that the model has good performance. We plan to explore the second solution to further improve the model as future work.

V. Evaluation

This section presents our evaluation results. We first describe the dataset used in our evaluation (Section V-A) and discuss how the model is trained (Section V-B). We then conduct three different tasks to evaluate the quality of the learned model: (1) the mono-architecture instruction similarity task (Section V-C); (2) the cross-architecture instruction similarity task (Section V-D); and (3) as a concrete application, the cross-architecture basic-block similarity comparison task (Section V-E).

A. Dataset

We train our model using basic blocks that are open-sourced by our prior work[2] [67], consisting of 202,252 semantically similar basic-block pairs. This dataset is prepared using `OpenSSL` (v1.1.1-pre1) and four popular Linux packages, including `coreutils` (v8.29), `findutils` (v4.6.0), `diffutils` (v3.6), and `binutils` (v2.30). Each program is compiled by two architectures (x86-64 and ARM) and `clang` (v6.0.0) with three different compiler optimization levels (O1-O3).

Two basic blocks of different ISAs compiled from the same piece of source code are considered as equivalent. To collect such ground truth, we modify the *backends* of various architectures in the LLVM compiler to add the basic-block boundary annotator, which annotates a unique ID for each block so that *all blocks compiled from the same piece of source code, regardless of architecture, will obtain the same ID*.

B. Model Training

We use the following settings to train our cross-architecture instruction model[3]: the instruction embedding dimension of 200, the sliding window size of 5, a subsampling rate of $1e$-5, negative sampling with 30 samples, and the learning rate of 0.05. The model is trained for 10 epochs and the learning rate is decayed to 0 once training is done. We set the hyperparameters in Equation 1 to 1 for γ and 4 for β.

C. Mono-Architecture Instruction Similarity Task

Instruction Similarity Test. Unlike the case of word embedding models—which have many existing word-aligned corpora to evaluate the quality of word embeddings—we do not have such data. We thus create a set of manually-labeled instruction pairs from the same architecture to test the instruction embeddings. We consider a pair of instructions to be similar if they contain the same opcode, and a pair of instructions with different opcodes to be dissimilar; but a few exceptions exist—for example, in x86, `cmp` and `test` are different opcodes but are semantically similar, and thus instructions containing them tend to have similar embeddings. We randomly select 50 similar instruction pairs and 50 dissimilar ones, which are assigned with labels 1 and -1, respectively.

We then measure the similarity of two instructions based on their cosine similarity. Figure 2 shows the ROC curves and

[2]https://nmt4binaries.github.io

[3]See [46] for more details on the parameters.

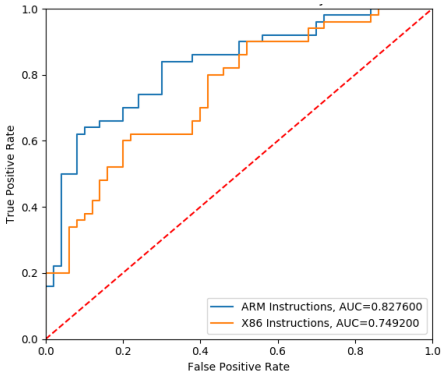

Fig. 2: The ROC for the mono-architecture instruction similarity test.

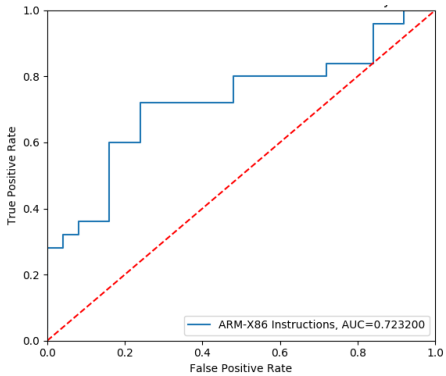

Fig. 3: The ROC for the cross-architecture instruction similarity test.

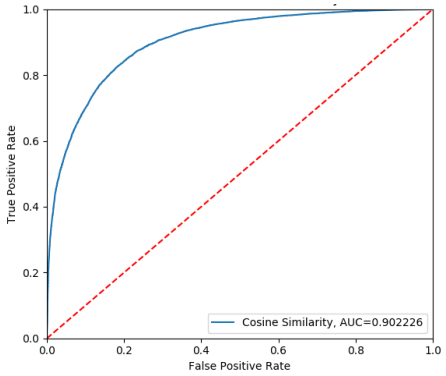

Fig. 4: The ROC for the cross-architecture basic-block similarity comparison test.

TABLE I: Nearest neighbor instructions *mono-architecturally*. It shows the top two similar ARM instructions for four randomly selected ARM instructions as measured by cosine similarity.

`ADD r1,r0,r7`		`SUB sp,sp,0`		`LDR r0,[r5+0]`		`MOV r0,r5`	
ARM	*Sim. score*	*ARM*	*Sim. score*	*ARM*	*Sim. score*	*ARM*	*Sim. score*
`ADD r1,r0,r5`	0.643148	`ADD r4,sp,0`	0.339248	`LDR r4,[r2+0]`	0.441962	`LDR r2,[sp+0]`	0.590213
`ADD r1,r0,r6`	0.630020	`STR r4,[r0],0`	0.339097	`STR r7,[sp+0]`	0.392250	`LDR r3,<tag>`	0.528087

TABLE II: Nearest neighbor instructions *cross-architecturally*. It shows the top two similar x86 instructions for six randomly selected ARM instructions as measured by cosine similarity.

`LDR r0,[r5+0]`		`LDRNE r4,[sp+0]`		`ADD r1,r0,r7`	
x86	*Sim. score*	*x86*	*Sim. score*	*x86*	*Sim. score*
`MOVL [rbp],eax`	0.437663	`CMOVLEL r14d,eax`	0.584823	`MOVQ [rax+0],r13`	0.538247
`MOVL [r14+0],eax`	0.432104	`CMOVEQ r12,r9`	0.584176	`ADDQ r13,rbx`	0.502640
`BLT <tag>`		`BEQ <tag>`		`MOV r8,r2`	
x86	*Sim. score*	*x86*	*Sim. score*	*x86*	*Sim. score*
`JL <tag>`	0.524643	`JE <tag>`	0.372829	`MOVQ r13,rdi`	0.453570
`JLE <tag>`	0.453281	`CMPB [rsi+0],0`	0.367084	`MOVQ r8,rbp`	0.413255

AUC values for ARM and x86: the AUC for ARM and x86 are around 0.828 and 0.749, respectively. It is worth noting that the accuracy of the monolingual word similarity test for the *multilingual* word embedding models are around 0.51 [18].

Nearest Neighbor Instructions. We then randomly select four ARM instructions, and search for the top two similar ARM instructions using cosine similarity. The result is shown in Table I. We omit the result for x86 due to space limits. It can be observed that the learned cross-architecture instruction embeddings still preserve the clustering property *mono-architecturally*. For example, our embeddings find very relevant neighbor instructions for the instruction `ADD r1,r0,r7`, such as `ADD r1,r0,r5` and `ADD r1,r0,r6`.

D. Cross-Architecture Instruction Similarity Task

Instruction Similarity Test. Similar to the previous experiment, we create a set of manually-labeled cross-architecture instruction pairs. We first determine the similar and dissimilar *opcode* pairs for different ISAs based on our prior knowledge and experience, and then select a set of similar and dissimilar *instruction* pairs based on whether their contained opcodes are (dis)similar or not. We then measure the similarity of two instructions using cosine similarity. Figure 3 shows the ROC result. Our model achieves AUC = 0.723 on this test.

Nearest Neighbor Instructions. We next randomly select six ARM instructions, and search for the top two similar x86 instructions for each ARM instruction based on their cosine similarity. The result is shown in Table II. We can see that similar instructions of different ISAs have embeddings close to each other, as predicted. For example, our embeddings find very relevant neighbor *x86* instructions for the *ARM* instruction `LDR r0,[r5+0]`, such as `MOVL [rbp],eax` and `MOVL [r14+0],eax`; both `LDR` from ARM and `MOV` from x86 can be used to load register from memory. Thus, the cross-architecture instruction embeddings successfully capture semantics of instructions across architectures.

Cross-Architecture Instruction Embedding Visualization. We next use t-SNE [45], a useful tool for visualizing high-dimensional vectors, to plot all the cross-architecture instruction embeddings in a two-dimensional space (due to space limits,

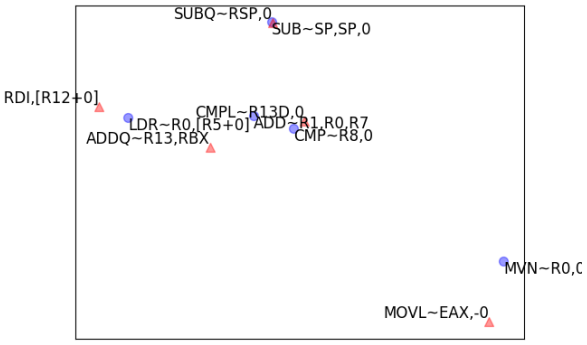

Fig. 5: Visualization of five ARM and x86 instruction pairs. A blue circle and red triangle represent an ARM and x86 instruction, respectively.

we omit it here). A quick inspection shows that the instructions of different ISAs overlap together. Our prior work [67] learns the *mono-architecture* instruction embeddings, where the instruction embeddings are architecture-specific (i.e., separate instruction embedding models need to be trained for each architecture), and the embeddings of different ISAs exist in different vector spaces (see Figure 9 in [67]). Instead, this work establishes a cross-architecture instruction embedding model, which learns embeddings in the same vector space.

We then visualize the embeddings of a set of similar instruction pairs. To this end, we randomly pick five x86 instructions; and for each x86 instruction, we select its similar counterpart from ARM based on our prior knowledge. We use t-SNE to plot their embeddings, as shown in Figure 5. It can be observed that most x86 and ARM instructions with similar meanings appear nearby: for example, the two similar x86 and ARM instructions, `SUBQ RSP,0` and `SUB SP,SP,0`, are close together in the vector space.

Therefore, our cross-architecture instruction embeddings capture not only instruction semantics, but also semantic relationships across different architectures.

E. Cross-Architecture Basic-Block Similarity Comparison Task

We next conduct the cross-architecture basic-block similarity comparison task to evaluate the quality of the learned model. We divide the dataset which contains 202,252 *similar* basic-block pairs into two parts: 90% of them are used for training; 10% of them and another 20,633 *dissimilar* block pairs (selected from the dataset open-sourced by our prior work [67]) for testing. Note that we only need similar block pairs for training; and for testing both similar and dissimilar pairs are used.

To measure the similarity of two basic blocks, we first compose all the instruction embeddings for each basic block, and then use the cosine similarity of the two composed embeddings to measure the basic block similarity. For simplicity, we use the sum of all the instruction embeddings of a basic block to represent it. This simple summation has proven to be a successful way of obtaining sentence or document embeddings that can be used as features in specific tasks [65], [24] such as answer sentence selection [65].

Figure 4 shows the ROC curve evaluated on the testing dataset; and our model achieves AUC = 0.90. Recent work [19], [21], [62] looks at the statistical information of a basic block, and uses several manually selected features (such as the number of instructions and constants) of a basic block to represent it. But such an approach causes significant loss of information about the instructions being used and their dependencies. As a result, the statistics-based representation is efficient but inaccurate—a SVM classifier based on such features can only achieve AUC = 0.85 according to our prior work [67].

Therefore, our model, capturing the meaning of instructions and the dependency between them, can provide more precise basic-block representation and efficient comparison. It is worth mentioning that many prior systems [23], [41], [53], [21], [62] built on basic-block comparison can benefit from our model.

VI. Future Work

Improvements. Currently, heuristics are used to decide parameter values; e.g., the window size is set as 20. We will investigate the stability of the cross-architecture instruction embedding model with respect to different hyperparameters—including the sliding window size, the number of epochs, and the instruction embedding dimension.

Two solutions are proposed in Section IV-D to find the alignment links between instructions. We have tried the first simple solution, and plan to explore the second one to attest how important alignment information is in learning cross-architecture instruction embeddings.

The sliding window based on program paths can reflect the context information of instructions more precisely and may generate better instruction embeddings. We plan to explore dynamic analysis to generate a set of semantically-equivalent paths from two programs compiled for different architectures, and use them for training. We will evaluate the model trained on paths in terms of accuracy and efficiency.

Applications. A prominent application of cross-architecture instruction embeddings (similar to multilingual word embeddings) is that the induced instruction embeddings enable us to *transfer* a classifier trained on one architecture to another without any adaptation. We plan to investigate the transferability by applying our model to the cross-architecture program/function classification problem. For example, we will train a classifier using the code compiled for x86, and check whether it can directly work on ARM.

Moreover, we plan to apply our model to other important code analysis tasks, such as cross-architecture bug search, and compare our model to recent approaches [21], [62], [67], [14].

VII. Conclusion

To the best of our knowledge, this is the first work that aims to learn cross-architecture instruction embeddings that tolerate the syntactic differences of instructions across architectures and capture their important semantic features. We adopt a joint learning approach to building the instruction embedding model, such that instructions with similar semantics, regardless of their architectures, have embeddings close together in the vector space. Our instruction similarity tests and cross-architecture basic-block similarity comparison task demonstrate the good quality of the learned instruction embeddings. The proposed model may be applied to many cross-architecture binary code analysis tasks, such as vulnerability finding, malware detection, and plagiarism detection.

REFERENCES

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

[2] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[3] BAP: The Next-Generation Binary Analysis Platform, "http://bap.ece.cmu.edu/," 2013.

[4] N. Bel, C. H. Koster, and M. Villegas, "Cross-lingual text categorization," in *International Conference on Theory and Practice of Digital Libraries*. Springer, 2003, pp. 126–139.

[5] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *KDD*. ACM, 2003, pp. 39–48.

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

[7] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation: Parameter estimation," *Computational linguistics*, vol. 19, no. 2, pp. 263–311, 1993.

[8] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *FSE*. ACM, 2016, pp. 678–689.

[9] G. G. Chowdhury, "Natural language processing," *Annual review of information science and technology*, vol. 37, no. 1, pp. 51–89, 2003.

[10] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the 26th USENIX Conference on Security Symposium, Security*, vol. 17, 2017.

[11] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[12] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *PLDI*, 2016.

[13] ——, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 79–94.

[14] ——, "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*. ACM, 2018, pp. 392–404.

[15] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[16] P. Dhillon, D. P. Foster, and L. H. Ungar, "Multi-view learning of word embeddings via cca," in *Advances in neural information processing systems*, 2011, pp. 199–207.

[17] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[18] P. Dufter and H. Schütze, "A stronger baseline for multilingual word embeddings," *arXiv preprint arXiv:1811.00586*, 2018.

[19] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[20] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 346–359.

[21] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *CCS*, 2016.

[22] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008.

[23] D. Gao, M. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," *Information and Communications Security*, pp. 238–255, 2008.

[24] S. Gershman and J. B. Tenenbaum, "Phrase similarity in humans and machines." in *CogSci*, 2015.

[25] S. Gouws, Y. Bengio, and G. Corrado, "Bilbowa: Fast bilingual distributed representations without word alignments," in *International Conference on Machine Learning*, 2015, pp. 748–756.

[26] S. Green, N. Andrews, M. R. Gormley, M. Dredze, and C. D. Manning, "Entity clustering across languages," in *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2012, pp. 60–69.

[27] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *ICSME*. IEEE, 2017, pp. 125–136.

[28] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 1909–1915.

[29] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, 2016, pp. 1606–1612.

[30] K. Iwamoto and K. Wasaki, "Malware classification based on extracted api sequences using static analysis," in *Proceedings of the Asian Internet Engineeering Conference*. ACM, 2012, pp. 31–38.

[31] Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Transactions on Software Engineering*, 2015.

[32] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007.

[33] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 2002.

[34] A. Klementiev, I. Titov, and B. Bhattarai, "Inducing crosslingual distributed representations of words," *Proceedings of COLING 2012*, pp. 1459–1474, 2012.

[35] T. Kočiskỳ, K. M. Hermann, and P. Blunsom, "Learning bilingual word representations by marginalizing alignments," *arXiv preprint arXiv:1405.0947*, 2014.

[36] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 2006, pp. 253–262.

[37] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification." in *AAAI*, vol. 333, 2015, pp. 2267–2273.

[38] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *ICONI*, 2017.

[39] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *ICSE*. IEEE, 2012, pp. 310–320.

[40] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 872–881.

[41] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *FSE*. ACM, 2014.

[42] ——, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, no. 12, pp. 1157–1177, 2017.

[43] L. Luo and Q. Zeng, "Solminer: mining distinct solutions in programs," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 481–490.

[44] T. Luong, H. Pham, and C. D. Manning, "Bilingual word representations with monolingual quality in mind," in *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*, 2015, pp. 151–159.

[45] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedsings of Workshop at ICLR*, 2013.

[47] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation," *arXiv preprint arXiv:1309.4168*, 2013.

[48] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013.

[49] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation," in *Advances in neural information processing systems*, 2013, pp. 2265–2273.

[50] S. A. Mokhov, J. Paquet, and M. Debbabi, "The use of nlp techniques in static code analysis to detect weaknesses and vulnerabilities," in *Canadian Conference on Artificial Intelligence*. Springer, 2014.

[51] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *AAAI*, vol. 2, no. 3, 2016, p. 4.

[52] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *ICSE*. IEEE, 2017.

[53] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

[54] L. Prechelt, G. Malpohl, and M. Phillippsen, "Jplag: Finding plagiarisms among a set of programs," *University Karlsruhe*, 2000.

[55] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 2003.

[56] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin, "Learning sentiment-specific word embedding for twitter sentiment classification," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2014, pp. 1555–1565.

[57] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, 2004.

[58] P. Wang, B. Xu, J. Xu, G. Tian, C.-L. Liu, and H. Hao, "Semantic expansion using word embedding clustering and convolutional neural network for improving short text classification," *Neurocomputing*, vol. 174, pp. 806–814, 2016.

[59] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *arXiv preprint arXiv:1707.04742*, 2017.

[60] J. Wieting, M. Bansal, K. Gimpel, and K. Livescu, "Towards universal paraphrastic sentence embeddings," *arXiv preprint:1511.08198*, 2015.

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

[62] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017.

[63] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[64] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016, pp. 1480–1489.

[65] L. Yu, K. M. Hermann, P. Blunsom, and S. Pulman, "Deep learning for answer sentence selection," *arXiv preprint arXiv:1412.1632*, 2014.

[66] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1105–1116.

[67] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *NDSS*, 2019.