An Instruction Embedding Model for Binary Code Analysis

Kimberly Michelle Redmond

Follow this and additional works at: https://scholarcommons.sc.edu/etd

Recommended Citation


This Open Access Thesis is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.
AN INSTRUCTION EMBEDDING MODEL FOR BINARY CODE ANALYSIS

by

Kimberly Michelle Redmond

Bachelor of Arts
University of South Carolina 2013

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Science
College of Engineering and Computing
University of South Carolina
2019
Accepted by:
Lannan Luo, Director of Thesis
Qiang Zeng, Reader
Csilla Farkas, Reader
Cheryl L. Addy, Vice Provost and Dean of the Graduate School
Dedication

I dedicate this to my mother and father. Thank you for allowing me to pursue my dreams, and for being an example of lifelong dedication and hard work.
ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Lisa Luo for her support and guidance in navigating this process, from the beginning of research to finalizing this thesis. Next, I would like to thank Dr. Qiang Zeng for his valuable help and insight into this topic. I thank Dr. Csilla Farkas for her cybersecurity expertise, and for assisting as a reader. I also wish to acknowledge the faculty, students, and staff of the CSE department for creating a supportive environment.
ABSTRACT

Binary code analysis is important for understanding programs without access to the original source code, which is common with proprietary software. Analyzing binaries can be challenging given their high variability: due to growth in tech manufacturers, source code is now frequently compiled for multiple instruction set architectures (ISAs); however, there is no formal dictionary that translates between their assembly languages. The difficulty of analysis is further compounded by different compiler optimizations and obfuscated malware signatures. Such minutiae means that some vulnerabilities may only be detectable on a fine-grained level. Recent strides in machine learning—particularly in Natural Language Processing (NLP)—may provide a solution: deep learning models can process large texts and encode the semantics of individual words into vectors called word embeddings, which are convenient for processing and analyzing text. By treating assembly as a language and instructions as words, we leverage NLP ideas in order to generate individual instruction embeddings. Specifically, we choose to improve upon current models that are only single-architecture, or that suffer from performance issues when handling multiple architectures. This research presents a cross-architecture instruction embedding model that jointly encodes instruction semantics from multiple ISAs, where similar instructions within and across architectures embed closely together. Results show that our model is accurate in extracting semantics from binaries alone, and our embeddings capture semantic equivalences across multiple architectures. When combined, these instruction embeddings can represent the meaning of functions or basic blocks; thus, this model may prove useful for cross-architecture bug, malware, and plagiarism detection.
# Table of Contents

Dedication ................................................................. iii  

Acknowledgments ......................................................... iv  

Abstract ...................................................................... v  

List of Tables ................................................................. viii  

List of Figures ................................................................. ix  

Chapter 1 Introduction .................................................. 1  

Chapter 2 Related Work ............................................... 6  
  2.1 Binary Code Similarity Detection .............................. 6  
  2.2 Machine Learning Approaches ............................... 8  

Chapter 3 Model Design ............................................... 13  
  3.1 Background .......................................................... 13  
  3.2 Cross-Architecture Instruction Embedding Model ........ 18  
  3.3 Pre-Processing ..................................................... 22  

Chapter 4 Evaluation ................................................... 24  
  4.1 Model Accuracy .................................................... 24  
  4.2 Instruction-Level Accuracy .................................... 26  

vii
LIST OF TABLES

Table 4.1 A sample of ARM and x86 instructions, displaying their most similar instructions from within the same architecture. .................. 27

Table 4.2 A cross-architecture sample of ARM instructions paired with x86 instructions that have the highest cosine similarity scores. ........... 27
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An example of assembly code in x86 and ARM architectures.</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>An illustration of CBOW and Skip-Gram architectures.</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>A 3-dimensional t-SNE projection of x86 and ARM architectures, trained separately.</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Our instruction embedding model. The current instruction, CALLQ FOO, is trained on both ARM and x86 instructions that are treated as its context.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>ROC for the cross-architecture basic block similarity comparison test.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>ROC for the mono-architecture instruction similarity task.</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>ROC for the cross-architecture instruction similarity task.</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>All instruction embeddings projected onto the same 2-dimensional space. Red is x86; blue is ARM.</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>5 randomly chosen instruction pairs projected onto the same 2-dimensional space. Red is x86; blue is ARM.</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>ROC for normalized instructions; no alignment step.</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>ROC for normalized one-to-many alignment.</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>ROC for normalized simple alignment.</td>
<td>32</td>
</tr>
<tr>
<td>4.9</td>
<td>ROC for normalized within-three-instructions alignment.</td>
<td>32</td>
</tr>
<tr>
<td>4.10</td>
<td>ROC for normalized two-zone alignment.</td>
<td>32</td>
</tr>
<tr>
<td>4.11</td>
<td>ROC for normalized LCS alignment.</td>
<td>32</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The ubiquity of technology has given rise to the great numbers and diversity now found in computers, smartphones, game consoles, and other devices. In 2018, 77% of Americans owned smartphones; 73% owned a desktop or laptop computer; and 53% owned tablet computers [55]. Around 90% of all American households contain at least one of these devices, where the typical household contains around five of them [54]. Worldwide, Gartner forecasts ownership of over 20 billion IoT (Internet of Things) devices in 2020 [19]. To keep up with widespread use of tech, companies often create their own software applications for users to run across their devices. The ability to cross-compile programs on various architectures has made software distribution easy, and more than ever, users are able to access applications on multiple devices.

Software poses more security risks, however, the more it is distributed. Assessing programs for such risks is not a trivial matter: many programs are proprietary, and thus closed-source, meaning that end users cannot access and analyze the source code for vulnerabilities. This includes work and office software (e.g., Microsoft Office), entertainment applications like Facebook or Netflix, and software on embedded devices (i.e., firmware). The implication is that when vulnerabilities exist in the source code, or in libraries that the source uses—OpenSSL before 1.0.1g contained Heartbleed)—then those vulnerabilities will be present in any device running that program. Further, compilers vary across devices; some vulnerabilities may only occur due to the compiler used for a particular device. Since code is frequently cross-compiled for various architectures, the most popular architectures will include those used for
computers and phones, such as x86 and ARM. With most of the world using these 'connected' devices, this puts many people at risk.

Without access to source code, we are left to analyze the binary machine code produced by the compiler. Thus, we can use binary code analysis to assess programs for vulnerabilities and risks. Analysis methods can be applied to various security-related tasks such as spyware discovery, malware detection, and user-side crash analysis; developers may also analyze programs to detect code clones and plagiarism. Since most widely-used programs are proprietary, and therefore closed-source, binary code analysis is often the only analysis tool available to end users.

However, evaluating binary code across architectures is a difficult task. A section of code (referred to here as a code component) in one instruction set architecture (ISA) may not resemble the same code component in another ISA; this is often due to differences in instruction sets, function offsets, memory access, CPU registers, and compiler optimization. Current research assumes that code similarities occur at the function level [13, 56], but this is not always true. For example, sections of functions may be plagiarized and inserted elsewhere in the program [27], or malware may be split up and distributed throughout the program [45]. In order to examine code components beyond the function level—without access to source code—there exists a need for fine-grained binary code analysis. While prior research has attempted fine-grained analysis techniques [27, 37, 46], they only work on a single architecture.

It is worth noting that, once disassembled, a binary can be expressed in some assembly language (shown in Figure 1.1). Machine learning has made great strides in processing languages through Natural Language Processing (NLP), which is efficient at extracting semantics from documents representing any language. Given some corpus of text for training, machine learning models have been designed for tasks such as summarizing documents [6] and translating between languages [2]. One such model—the word embedding model Word2Vec—learns the meanings of words from
their surrounding contexts: that is, from the words with which they tend to appear [43, 44]. When the model is trained, each word is represented as a high-dimensional vector that reflects the contexts found with that word; this is called its word vector or word embedding. Word2Vec gained fame when its creators proved how, like combining words to represent a concept, feature vectors could be combined mathematically: when the vector for "man" was subtracted from the vector for "king," followed by adding the vector for "woman," the resulting vector approximated the true vector for "queen" [42].

The reason this works is because word vectors are trained on the assumption that even across different texts, similar words will appear in similar contexts. For example, the word "king" should appear in contexts relating to kingdoms, rulers, and monarchies. During the neural training process, where embeddings are typically encoded in a hidden layer, words that share similar contexts will prompt the model to encode similar features for them—that is, their shared features will be pushed together toward sameness. The resulting features from the hidden layer for a target word will represent its final word embedding; therefore, similar words will produce similar embeddings. While this is usually true for training on documents from a single language, the same theory should hold true for documents from multiple languages. If two documents are considered parallel—that is, they represent the same content translated into each language—then training on parallel corpora for multiple lan-

---

<table>
<thead>
<tr>
<th>X86</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDQ RSI,R12</td>
<td>ADD R1,R0,R7</td>
</tr>
<tr>
<td>MOVL EDI,0</td>
<td>MOV R0,0</td>
</tr>
<tr>
<td>TESTL ESI,ESI</td>
<td>CMP R3,0</td>
</tr>
<tr>
<td>MOVQ RDI,[RDI]</td>
<td>LDR R0,[R0]</td>
</tr>
<tr>
<td>CALLQ FOO</td>
<td>BL FOO</td>
</tr>
</tbody>
</table>

Figure 1.1 An example of assembly code in x86 and ARM architectures.
languages implies that their word contexts can be used for words across both languages [1, 21, 39]. Thus, multilingual word embedding training should capture equivalent semantics from any language, if their content is also equivalent.

Since basic block code is structured and executed sequentially—with a program being like a document, where instructions in basic blocks are akin to words in sentences—this thesis seeks to treat disassembled binary code as a document containing its own language. Here, the goal of NLP model training is to encode the meanings of independent assembly instructions as vectors, and to consider them, as a group, contributing to the meaning of their basic blocks. A multilingual embedding model that uses similar documents from multiple languages means that when training on multiple assembly languages, embeddings will capture equivalent contexts for code, even if the same program was compiled on separate architectures. Therefore: if instructions within the basic block of one architecture can be understood to represent the block’s overall meaning, then comparing basic blocks or code components comprised of embeddings trained across architectures should yield similar results for any similar code component, regardless of its original architecture. Thus, I leverage NLP techniques to train multiple instruction set architectures in order to produce meaningful, non-architecture-specific instruction embeddings.

The structure of this thesis is as follows: we will first explore related work in binary code analysis and machine learning. Much research has been done on binary code clone detection; and recent work in machine learning has optimized natural language processing for a variety of uses. More specifically, a few attempts have been made at generating embeddings for machine language, but we will find that they succeed in a limited scope. Thus, this thesis proposes a fine-grained approach to binary code analysis and similarity detection. We will then explain the design of our instruction embedding model, and how natural language processing is leveraged to produce instruction embeddings. An evaluation chapter will cover performance
results of this model: accuracy will be assessed both mono- and cross-architecturally, and on the instruction- and basic block-levels. Finally, we will discuss the implications of these results and directions for future research.
Chapter 2
Related Work

Binary code analysis is an active field of research, and many prior works have explored methods to improve our understanding of code binaries. More specifically, we examine those works which have leveraged binary code analysis in code similarity detection tasks. Binary code analysis offers many diverse approaches; but due to the difficulty of detecting code signatures and translating across different instruction set architectures (ISAs), virtually all current methods suffer from limitations.

In this chapter, we will cover two major areas of interest. First, we review traditional research related to binary code analysis, covering both single-architecture and multi-architecture approaches. In the second section, we will examine how machine learning techniques have been applied in binary code analysis. With all advances considered, we will find that there is a gap in fine-grained, cross-architectural approaches to understanding binary code.

2.1 Binary Code Similarity Detection

Since we wish to examine code similarities across architectures, it is important to refer to previous work on code similarity detection. Static code analysis is a common approach for plagiarism detection, which includes string-based [3, 4], AST-based [28, 58, 64], token-based [29, 57], and PDG-based [10, 16, 34] methods. But as most static analysis relies on source code, we cannot adapt these solutions to closed-source or proprietary software. Even when we have analysis methods designed for binaries, such as extracting binary semantics through symbolic execution [17], the available
tools—such as BitBlaze [60] and BAP [5]—are computationally expensive and do not scale to very large programs, despite their good performance [37].

In order to better detect software piracy, there are also approaches based on software birthmarks [50]. A static software birthmark reflects unique characteristics the developers have ingrained in the original program, such as variable fields or class structures; however, attackers are often able to alter or obfuscate these fields. In contrast, a dynamic birthmark is reflected in the execution of a program with some given input: this results in unique execution paths acting as program signatures. Dynamic birthmark approaches include API-based [59, 61], instruction-based [51, 63], system call-based [65, 66], and value-based [27, 26] methods. These methods are especially useful for malware detection, as malicious binaries tend to contain user-defined function or system calls [30]. However, dynamic birthmarks can be weakened by differences in compiler optimization; a compiler may restructure or eliminate code that was intended to be compiled in a different way. Further, since dynamic analysis requires active runtime on a system, this makes it more difficult to analyze programs across multiple architectures. This difficulty only increases with the inclusion of embedded devices.

Researchers have begun to address the need for multi-architecture approaches in binary code similarity detection. A prototype model from [56] identifies bug signatures from multiple instruction set architectures (x86, ARM, and MIPS) by translating binary code functions into intermediate representations. However, function-level approaches are coarse-grained and will miss important code components that are small or distributed throughout a program. Further, their prototype relies on a graph matching-based algorithm that does not scale well to larger programs. Another approach, discovRE [13], uses pre-filtering to improve the graph matching process; this limits their process to examining only relevant sections of binaries, but it is not wholly reliable and remains computationally expensive.
Since the latest advances do not scale well for large programs, and have high computational costs, we need more efficient solutions in binary code analysis. Given widespread use of various computing devices, these solutions also need to handle code semantics across multiple architectures. For variety and efficiency in handling large programs and sets of data, we look to machine learning techniques for modern and scalable approaches.

2.2 Machine Learning Approaches

The past decade has demonstrated that machine learning is a useful and flexible tool for many applications. Machine learning has excelled at stock price prediction [52], DNA genotyping [11], and learning policies for playing games [47]. It is also adept at processing written and audible language, a subfield called natural language processing (NLP), where computers are trained to understand and manipulate human language to process and produce texts; NLP tasks include information retrieval and speech recognition [8]. Another natural language processing application is machine translation, where neural networks are used to automate translation between texts of different languages [2, 7]. The variability in machine learning applications can allow for cross-disciplinary approaches to solving modern problems.

The strength and flexibility of machine learning, particularly in deep learning, has also been beneficial for code analysis tasks. Since the use of software applications has only continued to grow, this has been an important effort in streamlining the process of identifying and correcting security vulnerabilities. Recent code analysis tasks that apply machine learning include software vulnerability identification [22, 33, 48], code similarity and clone detection [36, 67, 68], malware analysis [62], automatic patch generation [35], and bug detection [25], among many others. Research into machine learning for code analysis only continues to improve and proliferate with each passing year.
Despite continuing advances, current methods remain limited in their scope. The most direct solutions for analyzing code—whether for identifying security vulnerabilities or code similarities—examine the source code [49, 67]. However, as previously noted, we often lack access to source code when dealing with proprietary software. Even when analysis solutions do handle code binaries, they are generally limited to examining code on the code component or function level [13, 56]. Further, most tools are designed to work on a single instruction set architecture. This neglects vulnerabilities in source code that manifest across diverse devices, or vulnerabilities that are exclusive to certain architectures.

These limitations are problematic when attackers can obfuscate code signatures. Malicious or plagiarized code can be broken apart and distributed throughout a program, or junk code can be inserted to hide malicious code patterns. Vulnerabilities only become harder to detect when we consider differences in instruction sets across architectures. To counter this, a fine-grained approach should be able to examine all available code and grasp its semantics, identifying when a vulnerability exists even if it does not match a function-level signature. Ergo, a reliable tool should identify when a program contains malicious or buggy code, even if it is written or executed through a different set of commands. Since these commands would accomplish the same results as the original code, they should be understood to be similar in meaning. If instruction sets hold distinct meanings for each instruction, then they can be considered analogous to real languages formed of distinct words. It is with this insight that we look into advances in natural language processing.

2.2.1 Natural Language Processing

The field of natural language processing (NLP) has demonstrated impressive mastery over processing and understanding human language. Typically, NLP models are designed to extract semantics of words and represent them as word embeddings, which
embed the meaning of a given word into a high-dimensional numerical vector. These vectors can be one-hot encoded (a vector containing only 0, where a single, unique dimension represents that word as a 1); or more often, each dimension is a reflection of semantic properties found in the text. Such models work on the assumption that "you shall know a word by the company it keeps" [15]. In essence, similar words will appear in similar contexts; thus, similar words should be represented by similar embeddings after training. A popular word embedding model is Word2Vec, created by a team at Google, which is known for its efficiency in converting text into accurate embeddings [43]. While highly flexible, Word2Vec is primarily used for natural human languages.

In recent years, NLP has also been extended to process non-natural machine languages [62]. Since NLP models are generally designed to examine the contexts of words, like in sentences and documents, the same logic applies to any language that works within some structural context. This is viable for machine language because computer code is sequential in nature: programs are divided into basic blocks, and basic blocks are composed of instructions that are executed in sequence. Therefore, much like words ordered within sentences—where a word’s meaning is reflected by its definition and its role in the sentence—the type of instruction and its location in a program are meaningful for understanding the instruction. Unlike natural words, an instruction’s ISA definition is denoted by its opcode; its context-dependent meaning is represented by its operands. So while most NLP models are designed for human languages—whether targeting specific languages, such as English, or allowing for any of them—some models have been created solely with machine language in mind.

One such model is Instruction2vec, which converts instruction semantics into vectors [33]. However, basing its design on monolingual Word2Vec, Instruction2vec only works on a single architecture. Another model, Asm2Vec, generates function embeddings by converting functions into vectors [12]. While this is useful for identifying
known function-level bug signatures, its granularity is coarse, and it will miss code components or similarities that are not contained in functions. Like other tools, it is also limited to single architectures: a model proposed by Pechenkin et al. [53] does possess instruction-level granularity, which is accomplished through improvements on Long Short-Term Memory (LSTM) neural networks from previous authors, but it only works on a single architecture. It also does not train instructions as they appear: due to high variability in the x86 ISA, instructions are instead represented as a combination of their constituent parts (registers, offsets, etc.) to reduce input size. Since current methods can fail in identifying obfuscated code or similarities across ISAs—thus signaling high variation in software vulnerabilities and similarities—this generic-instruction approach may not be very useful.

In order to fully analyze binary code, regardless of how the program is written or structured internally, we need fine-grained solutions that can extract code semantics and understand the role each instruction plays toward the final outcome of the program. And since software is often distributed across various platforms, binary code analysis for these applications will ultimately require tools that are not limited by architecture choices. Next, we examine research into cross-architecture natural language processing techniques.

2.2.2 Cross-Architecture Approaches

Previous research has acknowledged the gaps in architecture-specific binary code analysis [13, 56, 68]. In light of this, some attempts have been made at cross-architecture machine learning approaches—but due to the inherent difficulty of handling cross-architecture ISAs, these come with limitations. One example, called MOCKINGBIRD [24], is a semantics-based analysis tool designed to detect code similarities across multiple architectures; however, it evaluates similarities based on prior examples of dynamically-extracted code signatures. This will miss vulnerabilities in the real world
that do not match code signatures, and known signature semantics may not be as robust as semantics captured in learning high-dimensional vectors from varied sets of programs.

Genius [14] and Gemini [68] are two tools that capitalize on cross-architecture embeddings, but they have their own limitations. Genius is a bug search engine that converts control-flow graphs (CFGs) into high-dimensional vectors, but these CFGs only represent function-level code. In Gemini, learning features are manually selected—and these chosen features are largely based on statistical patterns found in basic blocks. This overlooks much of the natural semantic meaning of instructions (e.g., opcodes and operands) and how they work together to form a basic block (e.g., how they operate on a group of registers in sequential order). Also, statistics-based features would miss distribution patterns in vulnerable code that has been split apart or obfuscated. Another model, INNEREYE-BB [70], uses LSTM to encode basic blocks into vectors across multiple architectures; however, each assembly language must be trained on separate models. This is problematic if similar programs do not produce similar cross-architecture embeddings as their outcomes; such outcomes may occur if separately-trained models focus on architecture-specific clustering patterns.

To account for model limitations in granularity and architecture options, this thesis proposes filling the research gap with a cross-architecture instruction embedding model that is fine-grained on the instruction level and is not limited to single architectures. Such a model can be used to identify code semantics—and from there, code similarities and vulnerabilities—across multiple architectures. In the next chapter, a description of this instruction embedding model will explain how it extracts instruction-level semantics for multiple instruction set architectures.
Chapter 3
Model Design

In order to improve our ability to analyze code components compiled across multiple architectures, regardless of their function or location in a program, we consider a cross-architecture instruction embedding model that leverages natural language processing techniques to capture the meaning of assembly instructions. Like word embeddings, instruction semantics will be encoded as instruction embeddings.

Section 3.1 will review the background on word embeddings, including a basic explanation of how word embedding models are designed; specific models will then be discussed, including monolingual and multilingual varieties. Section 3.2 will explain in greater detail the generation of instruction embeddings through our cross-architecture instruction embedding model. This includes an explanation of how basic blocks are arranged, and how instructions are ordered therein. Finally, Section 3.3 will cover how binaries are pre-processed for our embedding training process.

3.1 Background

3.1.1 Word Embedding Models

Prior to the use of neural network models, researchers adopted a straightforward approach in statistical language modeling to represent word and phrase semantics. In statistical language modeling, a probability distribution represents the relative likelihoods that various words and phrases will appear in a text; one example is n-gram models, where the probability of observing a word $n$ can be approximated by
the observation of the preceding \( n - 1 \) words. However, these simplistic models rely on massive training corpora to achieve diversity in combinations of words and phrases; there are not enough texts to represent all possible kinds of documents that can exist. With the advent of machine learning in recent decades, researchers have taken steps to create more complex models to infer meaning from limited training data. Extraction of fine-grained word semantics from documents—distributed representations called *word embeddings*—are one achievement of recent research.

The generation of word embeddings is typically accomplished through neural network models \([9]\), which have excelled in feature learning due to the copious amounts of texts that are available for training; both labeled and unlabeled texts can be useful under various types of model supervision. Embedding models evaluate the meanings of words based on the *context* of their surrounding words. This is based on the idea that "[y]ou shall know a word by the company it keeps" \([15]\). For example, consider that a discussion of acting might include the words perform, film, stage, and theatre; these words inform the concept of what an actor is, and this *context* would help define the word embedding for actor. Further, depending on its placement within a sentence as a subject or an object, the word *actor* may be replaced with the words *actress* or *thespian* if it better reflects the identity of the actor. It is reasonable to conclude that if these words are roughly interchangeable in a sentence, then they are *similar* in role and meaning. A successful word embedding model would then "push" the embeddings for actor, actress, and thespian to be most similar to one another out of all the acting-related words.

Word embedding models have already found a lot of success in generating meaningful embeddings for human language. Mikolov’s team \([44]\) popularized two standard embedding architectures: the Continuous Bag-of-Words (CBOW) and Skip-Gram (SG) models are two-layer neural networks that are efficient in both speed and memory. In the network, both the input and output layers are represented by one-hot
vectors of size $1 \times V$, with $V$ being the size of the vocabulary. For the CBOW model, the context words surrounding a target word $w(t)$ are accepted as input, with the goal of predicting $w(t)$. For SG, the model is simply reversed. The target word $w(t)$ is used to predict its surrounding context window. See Figure 3.1 (from [41]) for an illustration of these models.

We will examine the CBOW model as an example. As sliding windows of context words are accepted as input, with the goal of predicting a target word in the output layer, errors are backpropagated and weight matrices get updated to encourage accurate predictions. Each word is represented by a feature vector in the hidden layer, and this is where we extract our final word embeddings. When words appear in similar contexts in documents, the model also learns which target words share these contexts, making them similar; therefore, they produce similar embeddings from the training process. While this explanation is adequate for vocabularies comprised of a single language, it cannot singly learn contexts across documents from multiple languages. This has been addressed by later researchers, and the next section will cover multilingual embedding models.
Cross-lingual semantics have gained traction in natural language processing tasks such as information retrieval [32] and machine translation [69]. For deriving word distributions, the CBOW and Skip-Gram architectures have become standard in models like Word2Vec, a popular and effective tool for learning word embeddings [43]. However, researchers interested in word representations for multiple languages are limited to training separate models for each one, using corpora from a single language at a time. If we wish for cross-lingual embeddings to represent the same semantics across languages—that is, for translatable and equivalent words to have similar embeddings—then this may not be achievable through separate monolingual training. The inability to embed similar semantics is often due to structural differences in languages and clustering patterns found within monolingual corpora. An example of separately-trained ARM and x86 embeddings projected through t-SNE [40] can be seen in Figure 3.2 from [70]: despite training on similar binaries, instructions from single architectures cluster together because they are strongly influenced by syntactic variation found only in that architecture’s training data.

Some researchers have designed word embedding models to address these issues, and they extract cross-lingual semantics through a variety of approaches. Klementiev et al. [31] propose a model that learns separately on different languages; during training, the learned representations are jointly induced and aligned through a regularization term according to co-occurrence statistics from parallel data. Another approach also separately trains languages, but bridges word representations across languages through multilingual mapping [18, 41]. One model from Zou et al. [69] is the first that directly induces bilingual word embeddings from unlabeled data, accomplished through an objective function that incorporates both monolingual semantics and bilingual word alignment counts. However, while these models demonstrate the efficacy of distributed word representations to reflect bilingual semantics, the reliance
on parallel word alignment does not allow for the mixed word order that can often appear in equivalent bilingual phrases. Such an approach would not be suitable for machine languages, where equivalent basic blocks can lack instruction-level alignment.

Researchers acknowledge the need for more flexible approaches, and later models have been designed that do not require word-level alignments for training corpora. For example, Chandar A P et al. [1] take an autoencoder approach that leverages bilingual corpora to learn document representations that work for two languages; a classifier trained on one language should then work on the second language. Further models are designed to train on data that is parallel, but on the sentence level rather than word level [21, 23, 39]. These models use sentence-parallel corpora to extract a bilingual signal to align cross-lingual embeddings during training. Gouws et al. [21] employ a loss function that minimizes the vector distance between similar words. Notably, their model only requires a limited set of sentence-parallel corpora to extract the bilingual signal; monolingual data is sufficient for most of its training. In contrast,
the model from Luong et al. [39] trains exclusively on sentence-aligned data from two languages, and the inclusion of word alignments is optional; their results are on par, or better, than those of Gouws et al. in cross-lingual document classification.

To the best of our knowledge, there is no model for generating word distributions that is designed to handle multiple non-natural languages. It is following the insight and promising results of [39], who acknowledge the problem of word-level alignments across languages, that we adapt their bilingual model for generating our cross-architecture instruction embeddings.

3.2 Cross-Architecture Instruction Embedding Model

It is important to remember that meaningful, semantically accurate word embeddings should work on both monolingual and multilingual tasks. For our instruction embeddings, this means that a command to subtract (e.g., \texttt{SUB SP,SP,0} in ARM) should be semantically similar to other subtraction-related instructions in both the original and other architectures, such as \texttt{SUBQ RSP,0} in x86. Since the model of [39] trains bilingual embeddings with the additional goal of retaining high monolingual quality, we follow this approach in designing our cross-architecture model.

To extract semantics from within and across architectures, our model takes parallel training data from two architectures simultaneously, but it considers their mono-architecture and multi-architecture information separately. For the mono-architecture component, context concurrence information is taken from the training data of each individual architecture. The multi-architecture component extracts semantic equivalence signals between parallel basic blocks from the two architectures. When trained together, the resulting embeddings should share the same vector space: similar vectors within one architecture should cluster together, and their counterparts in another architecture should also cluster with them. Similar instructions should effectively map onto each other, without the requirement of a mapping process.
Joint Objective Function. Our model is based on a joint objective function that leverages the objective functions of both the mono-architecture and multi-architecture components.

$$J = \gamma \sum_{i=1}^{N} J(Mono_{a_i}) + \beta \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} J(Multi_{<a_i,a_j>})$$ (3.1)

The mono-architecture clustering patterns of a single architecture, $a_i$, are captured by $Mono_{a_i}$, $\forall i \in \{1, ..., N\}$. The semantic equivalence signal across architectures is extracted through $Multi_{<a_i,a_j>}$, $\forall i,j \in \{1, ..., N\}$, $i \neq j$. The objective functions for these components, respectively, are represented by $J(Mono_{a_i})$ and $J(Multi_{<a_i,a_j>})$.

For two architectures, such as x86 and ARM, the mono-architecture objective will be represented by $J(Mono_{x86}) + J(Mono_{ARM})$, and the multi-architecture objective by $J(Multi_{<x86,ARM>})$. The two hyperparameters, $\gamma$ and $\beta$, balance the influence of each component. For our model, $\gamma = 1$ and $\beta = 4$.

As noted in [39], any word embedding model would be suitable for the mono-architecture component; the cross-architecture component typically adopts a different objective function. To preserve natural consistency, we use the same objective function for both components. Specifically, we use the CBOW architecture in Bivec [38] as adapted from the Word2Vec [44] tool. Despite relative popularity of the Skip-Gram architecture, our preliminary results showed that CBOW was superior for our model. The CBOW architecture maximizes the following objective function:

$$J = \frac{1}{T} \sum_{t=1}^{T} \log P(e_t|e_{t-n}, ... e_{t-1}, e_{t+1}, ..., e_{t+n})$$ (3.2)

where $T$ represents the length of the basic block, and $n$ the length of the sliding window. Maximizing the log probability, $\log(P)$, means that instructions which frequently appear together should have the highest probabilities for co-occurrence.
Figure 3.3 Our instruction embedding model. The current instruction, CALLQ FOO, is trained on both ARM and x86 instructions that are treated as its context.

The key insight is that this objective function also applies cross-architecturally. Given parallel training data, this is accomplished by treating parallel basic blocks from each architecture as similar contexts for a single instruction. If a pair of basic blocks is dissimilar (non-parallel), then their instructions should have low likelihoods for appearing concurrently; a similar block pair would maximize a higher likelihood. Thus, this represents the multi-architecture component of our model. See Figure 3.3 for an illustration of how parallel basic blocks are trained cross-contextually.

Figure 3.3 shows two basic blocks being trained in our embedding model. The current instruction, CALLQ FOO, is aligned with the ARM instruction BL FOO. Not only are the surrounding x86 instructions used as context for CALLQ FOO, but the surrounding sliding window for its ARM counterpart BL FOO are also treated as context. This is how our model learns predictive signals both mono-architecturally and cross-architecturally. Which instructions in the target language are chosen to stand in place of the current source instruction depends on how instruction alignments are determined for basic blocks.

Instruction Alignment. The previous sections have already outlined the importance of parallel training data, such that similar basic blocks across architectures
should be aligned during training. It is already understood that due to differences in syntactical structure and grammar across human languages, sentences may be similar in meaning but dissimilar in word order—for example, the semantically equivalent sentences "I miss you" in English and "Tu me manques" in French: the latter is equivalent in word order to "You me miss," which is translated as "You are missing to me." Despite the same meaning expressed by both sentences, precision may be lost when they are aligned and trained on a word-by-word level.

Word order, or instruction order, is particularly critical when comparing binaries across architectures. Architectures and compilers are designed with different priorities in mind when optimizing code for run time, so instructions may be ordered in different sequences across devices. This may affect which opcodes or registers are used for each instruction, and in turn, this affects which instructions will get trained for embeddings. By default, our model assumes naive alignment: that is, it assumes that instructions are already ordered and aligned across basic blocks. If a basic block of the source architecture is of length $M$, and a target basic block of length $N$, then source instruction $i$ is aligned with the target instruction at position $i \times |N|/|M|$.

In order to improve precision and accuracy, we also examine an optional instruction alignment step that will communicate to the embedding model which instructions are aligned across basic blocks. To create alignment labels, all instructions will first be normalized to simple forms for opcodes; this is because there are many variations on the same basic opcodes, especially within the complex x86 architecture. An example is converting all opcodes for multiplication (e.g., UMULL, UMLAL, SMLAL) to a single MUL. After all instructions are normalized across ARM and x86 documents, they are listed in a dictionary. The dictionary begins each entry with an opcode from the source language (here, x86) listed first, and all matching opcodes from the target language (ARM) are listed after it. We create an alignment program that steps through the training data and assigns alignment labels according to whether the dictionary
defines instruction pairs as similar.

Since there are often several possible matches for instructions, various alignment algorithms are also examined. These include one-to-one and one-to-many matching; more specifically, some algorithms also determine alignment according to the relative closeness of instruction positions across basic blocks. Using the assigned alignment labels from each of these algorithms, the normalized-opcode corpora will be trained and evaluated for accuracy; the idea is that aligning basic blocks on the instruction level should improve on naively-aligned instruction embeddings. Results for all instruction embedding and alignment tests are detailed in the following chapter.

3.3 Pre-Processing

When compiling programs on our choice architectures, x86-64 and ARM, basic blocks of one architecture were matched with corresponding basic blocks of the other architecture; all block pairs were labeled for the ground truth of their similarity or dissimilarity (see Section VII-B in [70] for more information). In order to prepare these binaries for model training, and to sufficiently generalize data to reduce instances of out-of-vocabulary instructions, various steps were taken to convert the data into a suitable form. Then, various parameters were tested to determine the optimal parameter settings for our model.

**Out-of-Vocabulary (OOV) Instructions.** The issue of handling OOV words is a well-known one in NLP; but where natural language researchers can simply include more diverse text corpora to increase their vocabulary set, even increasing the amount of binary data is an inadequate solution when training on assembly languages. The use of unique function names, registers and labels, string literals, and numerical constants only exacerbates the risk of encountering previously-unseen instructions. Even the minute difference between `MOVL EDX, 3` and `MOVL EDX, 4` would require training two separate embeddings. Using trained embeddings to perform
tasks on new binaries, that have different function names or literals, would result in encountering many instructions that do not exist in the trained vocabulary.

Therefore, our disassembled binaries were pre-processed to simplify the training process; this is also because specific variable names are less important to an instruction’s meaning than how that instruction functions, and variable names will vary widely across programs. Replacing variables with generic labels allows for similar instructions to receive the same embedding, regardless of which binaries are used for training or testing; this also reduces the likelihood of encountering OOV instructions due to differences in names or literals. Here, we replace all function names with \texttt{FOO}. All string literals are replaced with \texttt{<STR>}, and other symbols with \texttt{<TAG>}. Numerical constants are replaced with 0; but if they were originally negative numbers, then the minus (-) sign is still retained. Examples of pre-processed instructions include \texttt{CALLQ FOO} and \texttt{MOVL ESI, <STR>}.

\textbf{Choice of Parameters.} Skip-Gram models typically perform better in several contexts, including documents with infrequent words \cite{39, 44}; but after a preliminary test of CBOW and SG models, CBOW proved to be a superior choice for our cross-architecture training. Further, negative sampling provided more accurate results than using a hierarchical softmax layer. Other parameter tests included dimension size (number of features), where 200 dimensions outperformed 100, 150, 300, and 500 dimensions. The optimal number of training epochs was also determined to be 10.
CHAPTER 4

EVALUATION

Our cross-architecture instruction embedding model was trained and evaluated using the Bivec [38] word embedding model on the Continuous Bag-of-Words (CBOW) setting; we used a learning rate of 0.05 that decayed to 0 as training completed. Parameters included the following: an embedding dimension size of 200; sliding window of size 5; subsampling rate $1e^{-5}$; negative sampling with 30 samples; and hyperparameters $\gamma = 1$, $\beta = 4$ as in Equation 3.1. The model was trained on two parallel documents, containing altogether 202,252 semantically similar pairs of basic blocks. These open-sourced binaries were prepared using OpenSSL (v1.1.1-pre1) and four Linux packages: coreutils (v8.29), findutils (v4.6.0), diffutils (v3.6), and binutils (v2.30) [70]. They were compiled on ARM and x86-64 architectures at various optimization levels.

The following sections share our results in measuring for basic block-level similarity, instruction-level similarity, and shared dimension space across both architectures. Then, we will explore the results of including instruction normalization and alignment.

4.1 MODEL ACCURACY

Once instruction embeddings were trained, our model design was evaluated for its quality and accuracy. Since our goal is to extract cross-architecture basic block semantics—that is, to identify similar code components across architectures—then a sensible test is basic block similarity comparison. Since instruction embeddings are high-dimensional vectors representing the semantics of each trained instruction—
and individual vectors can be compared for similarity—then combining instruction embeddings to produce a basic block embedding means that similarity can also be assessed on the basic block-level. This is analogous to combining word vectors to produce a sentence vector, as demonstrated in [20].

The formation of basic block embeddings is as follows: an evaluation program examines each pair of basic blocks across two architectures, ARM and x86. This is our test set, which is composed of basic blocks not present in the training set; pairs are already labeled as similar or dissimilar. Each instruction embedding within a basic block is summed together, through element-wise addition, if the embedding for that instruction is available from the training process. Despite efforts to reduce out-of-vocabulary instructions, some possibility remains that the test set will include instructions unseen by the model during training. If a basic block is comprised of other previously-seen instructions—which will have available embeddings—then for

Figure 4.1  ROC for the cross-architecture basic block similarity comparison test.
simplicity, the unseen instruction is skipped from counting toward the basic block embedding. If a basic block is composed entirely of unseen instructions, then it is assigned an embedding consisting only of small values (e.g., 0.1). This is because some similarity metrics cannot be computed against an empty vector. The resulting basic block embedding retains the same dimensions as the instruction embeddings.

To determine basic block-level similarity, various similarity functions are available to use: cosine similarity, Euclidean distance, and Manhattan distance. Similarity and dissimilarity for basic blocks are labeled as 1 and -1, respectively. For this task, each metric was used to determine similarity between basic blocks across architectures. The results are shown in Figure 4.1: the model achieved an AUC of approximately 0.90 under cosine similarity, which is more accurate than several models evaluated on natural languages [21, 39]. This also outperforms models that generate basic block embeddings according to their statistical information, which only achieved an AUC = 0.85 [70].

4.2 Instruction-Level Accuracy

Our model should accurately determine not only basic block similarity, but also fine-grained instruction similarity. This is important for several reasons: first, the simple reason is that basic block embeddings will only be accurate if their constituent instruction embeddings are accurate. Also, we claim that our cross-architecture instruction embedding model captures semantics both within and across architectures; this means that instructions should be similar to their equivalent counterparts from either architecture. For example, if an aligned basic block pair performs similar sets of additions, then these addition-related instruction embeddings from both architectures should be close as vectors. Likewise, addition-related instructions should be considered similar within a single architecture: in ARM, the instruction ADD r1, r0, r5 should be more similar to ADD r1, r0, r6 than to a subtraction or branch instruction (see Table 4.1).
Table 4.1 A sample of ARM and x86 instructions, displaying their most similar instructions from within the same architecture.

<table>
<thead>
<tr>
<th>ARM</th>
<th>Cos Sim</th>
<th>ARM</th>
<th>Cos Sim</th>
<th>ARM</th>
<th>Cos Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1,R0,R7</td>
<td>0.643148</td>
<td>LDR R2,[SP+0]</td>
<td>0.590213</td>
<td>RBCS R0,R1,R3,ASR0</td>
<td>0.725797</td>
</tr>
<tr>
<td>ADD R1,R0,R5</td>
<td>0.630020</td>
<td>CMP R7,0</td>
<td>0.559265</td>
<td>LDRNE R7,[SP+0]</td>
<td>0.627820</td>
</tr>
<tr>
<td>RSC R3,R3,0</td>
<td>0.543972</td>
<td>LDR R3,&lt;TAG&gt;</td>
<td>0.526987</td>
<td>LDR R4,[R0+R10+LSL0]</td>
<td>0.620364</td>
</tr>
</tbody>
</table>

Table 4.2 A cross-architecture sample of ARM instructions paired with x86 instructions that have the highest cosine similarity scores.

<table>
<thead>
<tr>
<th>X86</th>
<th>Cos Sim</th>
<th>ARM</th>
<th>X86</th>
<th>Cos Sim</th>
<th>X86</th>
<th>Cos Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVQ RDI,[R12+0]</td>
<td>0.476179</td>
<td>MOVUPS XMM1,[RSP+0]</td>
<td>0.912346</td>
<td>PUSHQ R14</td>
<td>0.653825</td>
<td></td>
</tr>
<tr>
<td>MOVZBL ED1,[RSP,RBX+0]</td>
<td>0.475152</td>
<td>MOVZWL ECX,[RSP-0]</td>
<td>0.922589</td>
<td>MOVQ RD1,R12</td>
<td>0.619541</td>
<td></td>
</tr>
<tr>
<td>MOVQ [RCX+0],0</td>
<td>0.446267</td>
<td>MOVZBL EBP,[RBX,RAX]</td>
<td>0.887129</td>
<td>MOVQ RAX,[RBX]</td>
<td>0.648331</td>
<td></td>
</tr>
</tbody>
</table>

And finally, in binary code analysis, vulnerable code may exist in fragmented sections rather than within a single basic block or function. Thus, we examine accuracy on a fine-grained level.

To determine whether two instructions (within a language, or between two languages) are similar, their high-dimensional vectors were compared for cosine similarity scores. Just as in basic block embedding scores, a number near -1 means they are dissimilar; a score around 0 is a neutral relationship; and a score near 1 suggests high similarity.

**Mono-Architecture Instruction Similarity.** This task was designed to assess mono-architecture accuracy, or whether an instruction within one architecture is similar to other instructions sharing similar functions and opcodes. Therefore, 100 pairs of instructions were chosen at random: half were labelled as similar, and the other half as dissimilar. This is based on whether the instructions shared similar op-
codes, thus performing similar functions; operands were not considered for this task. Positive results should indicate successful extraction of mono-architecture clustering properties, which is the mono-architecture objective for our model.

The results shown in Figure 4.2 achieve an accuracy of 0.82 for ARM and 0.74 for x86 instruction pairs. This discrepancy may be due to a smaller opcode vocabulary in ARM, and higher opcode variation in x86. Where basic blocks in ARM tend to repeat the same opcode multiple times—suggesting a lot of similar clustering patterns—basic blocks in x86 tend to have a greater variety of instructions, so similar opcodes are repeated in clusters less often. Table 4.1 shows that the most similar instructions tend to share the same opcodes, which is expected.

**Cross-Architecture Instruction Similarity.** After the mono-architecture task was completed, the cross-architecture task also had to be examined. While our embedding model is designed to capture mono-architecture clustering properties, its cross-architecture objective is also meant to extract cross-architecture signals across aligned basic blocks. Here, cosine similarity was used again. A successful model will show high similarity scores for instructions with similar functions and opcodes,
regardless of architecture.

When the same evaluation was applied to 50 randomly-chosen pairs of cross-architecture instructions, half labeled as similar and half dissimilar, the task achieved an AUC of 0.72. This is on par with the mono-architecture task, suggesting similar levels of success for both objectives. In Table 4.2, the top cross-architecture pairs tend to be very similar instructions. The model is especially good at identifying branch instructions, as we can observe that ARM’s BLT <TAG> is most similar to JL <TAG> and JLE <TAG> in x86.

4.3 Instruction Embedding Projection Space

It is also worth observing how the resulting instruction embeddings appear together within a shared dimension space. When trained separately on monolingual embedding models, languages tend to cluster away from each other in separate dimension spaces; this happened for assembly languages in previous research into ARM and x86 embeddings ([70], or refer here to Figure 3.2). The goal of our cross-architecture model training is to encourage multiple assembly languages to converge in the same
dimensional space, which implies semantic equivalence if two instructions embody the same space. Ideally, this would allow for direct mapping of instructions from one architecture to their translations in another architecture.

For this task, the t-SNE [40] algorithm was used to project instruction embeddings onto a 2-dimensional window. Results are shown in Figures 4.4 and 4.5. Figure 4.4 displays a projection of all instruction embeddings trained on both architectures: here, we observe that x86 and ARM instructions tend to cluster on top of each other. This is a positive result that contrasts with the clear separation of architectures in Figure 3.2. In Figure 4.5, five instruction pairs were randomly selected to further illustrate the close proximity of similar instructions. For example, it can be observed that the ARM instruction `SUB SP,SP,0` appears in the same location as its x86 counterpart, `SUBQ RSP,0`.

### 4.4 INSTRUCTION ALIGNMENT

Finally, instruction embeddings should be optimized for the greatest possible fine-grained accuracy. While many word embedding models require sentence-level parallelism, this does not mean that words within a sentence are aligned. In its default state, our model assumes naive alignment: instructions in the source architecture are automatically aligned with instructions of the closest similar position in the target architecture. To optimize results, it may be beneficial to align instructions prior to training so that they are not mismatched with instructions that appear in a different order. Therefore, for this task, an optional instruction alignment step is explored as a method for increasing model accuracy.

As detailed in Chapter 3, a normalization step was performed to simplify the instruction vocabulary. This makes it easier to create an alignment dictionary, such that alignments can be judged according to whether they share similar opcodes across architectures. However, this generalization does slightly reduce accuracy; the model
baseline accuracy achieves an AUC = 0.88 (Figure 4.6) on normalized instructions without any alignment, compared to 0.90 for the standard instruction set. It is on this baseline accuracy that we base the performance of various algorithms for instruction alignment.

First, we examine one-to-one versus one-to-many alignments. In one-to-one alignment, we assume that a source instruction pairs with only one other instruction in the target architecture. One-to-many alignment allows for multiple matches, resulting in many additional alignments. Our goal is to encourage similar opcodes to embed close together, regardless of operands. A preliminary test of both varieties used a simple approach: in one-to-many matching, all similar opcodes across architectures were recorded as aligned. Figure 4.7 shows a decrease in accuracy with an AUC = 0.85. In simple one-to-one matching, for each source instruction, the target instructions were scanned in order; as soon as a matching target opcode was found, it was recorded as the single alignment for the source instruction. This achieved an AUC = 0.88, as shown in Figure 4.8. These basic alignment tests show that one-to-one outperforms one-to-many alignment, so we focus on the performance of three more specific one-to-one matching algorithms from here.
Figure 4.8  ROC for normalized simple alignment.

Figure 4.9  ROC for normalized within-three-instructions alignment.

Figure 4.10  ROC for normalized two-zone alignment.

Figure 4.11  ROC for normalized LCS alignment.
Matching Within Three Instructions. The average basic block of one architecture does not tend to differ significantly in length from its equivalent basic block in another architecture. It is with this observation that we examine opcode matching within three instructions: if an instruction is at position $i$ in architecture $a_j$, then it is considered aligned with the first matching instruction $i \pm 3$ in architecture $a_k$. Figure 4.9 shows that this algorithm achieves an AUC = 0.86; this may be affected by basic blocks which differ greatly in length.

Instructions in the Same Zone. To account for differences in length, another algorithm only considers instructions if they reside in the same zones relative to their basic blocks. Here, we split basic blocks into two zones: the first and second halves. If an instruction in architecture $a_j$ appears in the first $M/2$ instructions, where $M$ is the length of one basic block from $a_j$, then it accepts the first matching opcode found in the first $N/2$ instructions of the other architecture $a_k$, where $N$ is the length of that basic block. The same rule applies for the second half. Figure 4.10 displays the results, where zoning outperforms within-three matching with AUC = 0.87.

Longest Common Subsequence. The Longest Common Subsequence (LCS) algorithm searches for the longest possible sequence common to two sets, provided that each element of the sequence appears in the correct matching order; it does not matter if unrelated elements appear in between. For this task, we searched for the LCS of matching opcodes between the basic blocks of our two architectures. Results shown in Figure 4.11 indicate an AUC = 0.86, borderline 0.87, which is mostly on par with the two-zone alignment algorithm.
Chapter 5

Discussion

Results from various evaluation tasks suggest that our novel cross-architecture instruction embedding model works well in its preliminary stages, and may prove to be even more accurate with future refinements. With a baseline accuracy of 90%, the majority of cross-architecture basic blocks are correctly labeled as similar or dissimilar. This is impressive when considering numerous differences in their ISAs, and the presence of out-of-vocabulary instructions that could not contribute to basic block embeddings. The t-SNE projections also demonstrate how well our model bridges embeddings across different architectures; rather than seeing separate architecture embeddings diverge from each other, similar instructions across architectures embed near one another.

Results are also notable on the fine-grained level. The cross-architecture instruction similarity task indicates that our model is fairly strong in identifying similar instructions across architectures. However, it is weaker in identifying mono-architecture similarities within x86. This may be because the x86 ISA is more complicated than that of ARM; there are more opcodes designed to perform slightly different tasks, and this results in greater variation among basic blocks in x86. In contrast, ARM has a smaller opcode selection, where similar opcodes tend to cluster together for more complex operations; this is preferable for the mono-architecture objective of our model, which aims to capture these clustering properties. To improve results for all these tasks, a larger data set for training may provide more possible contexts in which these instructions can appear.
The instruction alignment task approaches our model’s standard performance, and it may prove to be promising with future work. To account for the drop in baseline accuracy for normalized instructions (88% versus 90%), the complexity of ISAs—particularly that of x86—may indicate that normalization generalizes opcodes too much on its own. For optimal accuracy, future alignment work might want to use the full, standard ISAs without normalization; still, it is useful for assessing the usefulness of our algorithms for the alignment task. The simplest one-to-one solution—accepting the first found opcode match as aligned instructions—held the best performance (88%) because it is probably the most likely algorithm to guarantee an alignment. The other methods, including within-three and two-zone matching, limit the number of available instructions which have the potential for alignment; this is likely to miss aligned instructions across basic blocks that differ significantly in length. A common example is when instructions from one architecture are repeated several times in a single basic block (such as multiple PUSH or POP commands), but a different architecture can execute the same actions using a single instruction. This would result in very different block lengths and possible missed alignments. Future work should take steps to ensure that alignments are guaranteed to be recorded, as long as they are sensible and available.

5.1 Applications of Research

There are many possible uses for our cross-architecture instruction embedding model. It could play a critical role in binary code analysis, which has become increasingly difficult as malicious coding techniques have grown cleverer and harder to detect; code theft is another concern that may be masked. Basic block or code component embeddings might be able to identify concerns such as malware, spyware, bugs, and plagiarism, regardless of architecture or platform. This might also be applicable in computer forensics, since binaries from running malicious code could be collected and
analyzed for vulnerabilities.

Outside of software security, it is also useful to understand binaries regardless of where they were compiled. Currently, there is no dictionary for translating between ISAs of different machines. If our model provides translational equivalence in embedding semantics, then a classifier trained on one architecture could feasibly be used to classify code from a different architecture. Therefore, only one model would be necessary to perform classification tasks on any binary.

5.2 Future Work

Since this work is preliminary and establishes a foundation for cross-architecture instruction embeddings, there are many possible improvements that can be made to the model. Our choices for settings are based on common parameters used in previous NLP research efforts. To increase robustness of the model, future work includes the following:

Parameters. Hyperparameter settings should be more rigorously assessed for optimal results. The current choice of parameters may work well for our particular data set, but an expanded data set may be better-suited for different design choices. This includes model settings, such as the choice of CBOW or Skip-Gram architectures; the balance of $\gamma$ and $\beta$ hyperparameters for mono-architecture and cross-architecture training; and the choices in training dimensions, sliding windows, negative samples, and epochs, where optimal settings may vary with the size or nature of the training set.

Out-of-Vocabulary (OOV) Instructions. Currently, our model attempts to reduce the number of OOV instructions by generalizing variable names, numerical constants, and string literals; when OOV instructions are encountered, they are skipped. While this may work for preliminary testing, or when the amount of OOV instructions is low, future work should attempt to include all possible instructions.
If there is a unique instruction that was never trained to have an embedding, a possible solution may be to assign it the embedding of a similar instruction, e.g., an instruction with the same opcode.

**More Architectures.** Of course, ARM and x86-64 do not represent all instruction set architectures. Future work should train on binaries from more architectures, including MIPS, and perform more cross-architecture comparisons. One possibility includes designing the instruction embedding model to handle more than two assembly languages at a time.

**Bug Signature Comparisons.** Much prior work on binary code analysis, which examines binaries on the function level, seeks to detect vulnerabilities based on their signature patterns. It would be useful to compare our model to these established models, in order to ensure that our embeddings encode vulnerability signatures at the basic block or code component level. Also, comparisons should be made between code that contains clear vulnerabilities and code that obfuscates them: both sets of code accomplish the same goal, despite being written differently. If our model is successful, it will be able to generate similar semantic embeddings for either case.
With the explosion of new devices and software applications in the past decade, researchers have struggled to keep up with new software vulnerabilities that have risen with it. As malicious users become more clever at integrating their code into programs, and obscuring its appearance, there grows a need for greater flexibility in analyzing code. Particularly, as many software programs are proprietary and closed-source, there is a pressing need to analyze binaries from compiled code. There is much prior research on binary code analysis, but it is often restricted to the detection of bug signatures and function-level semantics; such approaches will inevitably fail to detect vulnerabilities as they become more hidden and advanced. Further, vulnerabilities will become harder to identify as they continue to be expressed in the varying instruction set architectures (ISAs) of numerous devices.

The advent of machine learning, particularly Natural Language Processing (NLP), has given us new ways to capture the semantic meaning of documents from different languages. If some form of assembly code can be considered an architecture’s language, then leveraging NLP techniques may prove useful for ascertaining the meaning and function of code. Specifically, word embedding models have been a popular choice for encoding word semantics into high-dimensional vectors; they tend to be applicable to any language, provided with enough textual data for training. Bilingual models have also been constructed with the goal of bridging word meanings and translational equivalences across languages.

In this work, we present what may be the first NLP model that aims to capture
code semantics, from any architecture, using only the binaries themselves: there is no need for manually-selected features or the extraction of statistics to estimate feature vectors. Using only the original binaries as context for instruction sequences, our results have shown high accuracy in representing instructions as high-dimensional embeddings. Our basic block and instruction similarity tasks, both within a single architecture and across multiple architectures, have demonstrated that patterns of semantic similarity and dissimilarity can be accurately detected. Our results also show that projecting instruction embeddings into the same dimensional window places similar instructions close together in vector space, and dissimilar instructions farther apart. We also tested various algorithms for instruction-level alignment, and future refinements may improve accuracy of the model.

We hope that this new approach will prove itself useful for the future of binary code analysis. A fine-grained understanding of assembly instructions may serve well in deciphering binaries across different architectures, and we hope it may contribute to the creation of a translation dictionary for different ISAs. Other applications in software security include malware detection, vulnerability identification, and plagiarism detection.
Bibliography


[38] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Bilingual Word Representations with Monolingual Quality in Mind”. In: NAACL Workshop on Vector Space Modeling for NLP. Denver, United States, 2015.


