2022

# Evaluating Similarity of Cross-Architecture Basic Blocks

Elijah L. Meyer
*Wright State University*

# EVALUATING SIMILARITY OF CROSS-ARCHITECTURE BASIC BLOCKS

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

ELIJAH L. MEYER
B.S.C.S, Wright State University, 2019

2022
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

April 27, 2022

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Elijah L. Meyer ENTITLED Evaluating Similarity of Cross-Architecture Basic Blocks BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

_____

Junjie Zhang, Ph.D.
Thesis Director

_____

Michael Raymer, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination:

_____

Junjie Zhang, Ph.D.

_____

Lingwei Chen, Ph.D.

_____

Meilin Liu, Ph.D.

_____

Barry Milligan, Ph.D.
Vice Provost for Academic Affairs
Dean of the Graduate School

# ABSTRACT

Meyer, Elijah L. M.S.C.S., Department of Computer Science and Engineering, Wright State University, 2022. Evaluating Similarity of Cross-Architecture Basic Blocks.

Vulnerabilities in source code can be compiled for multiple processor architectures and make their way into several different devices. Security researchers frequently have no way to obtain this source code to analyze for vulnerabilities. Therefore, the ability to effectively analyze binary code is essential.

Similarity detection is one facet of binary code analysis. Because source code can be compiled for different architectures, the need can arise for detecting code similarity across architectures. This need is especially apparent when analyzing firmware from embedded computing environments such as Internet of Things devices, where the processor architecture is dependent on the product and cannot be controlled by the researcher.

In this thesis, we propose a system for cross-architecture binary similarity detection and present an implementation. Our system simplifies the process by lifting the binary code into an intermediate representation provided by Ghidra before analyzing it with a neural network. This eliminates the noise that can result from analyzing two disparate sets of instructions simultaneously. Our tool shows a high degree of accuracy when comparing basic blocks. In future work, we hope to expand its functionality to capture function-level control flow data.

# Contents

# List of Figures

# List of Tables

# Acknowledgment

I would like to thank my parents and my brothers for their patience with me. I want to thank my thesis advisor, Dr. Junjie Zhang, for his guidance and support in every step of this process. I would also like to thank Dr. Lingwei Chen for his insight in the use of neural networks, and James Hamilton for his help with the creation of a graphic used in this thesis. Finally, I want to thank Dr. Junjie Zhang, Dr. Lingwei Chen, and Dr. Meilin Liu for taking the time to evaluate my thesis.

# Abbreviations

CFG — Control Flow Graph

CPU — Central Processing Unit

IR – Intermediate Representation

LSTM – Long Short-Term Memory

NLP — Natural Language Processing

# Introduction

We propose a solution for detecting similarity in binary code basic blocks compiled for both the ARM and the x86 architectures. Our tool translates code blocks into an intermediate representation and uses these translated blocks to train a neural network. This neural network converts code blocks into a numeric vector that captures semantic information about the block. We compare the vectors derived from code blocks to determine whether the blocks are similar.

## 1.1 Motivation

Code similarity detection is a crucial facet of cyber security. Applications of this process include code plagiarism detection, malware detection, and vulnerability search [1]. While examining the source code of two programs may be the most straightforward way to detect code similarity, a program's source code is often unavailable to be analyzed. This may be because the source code is proprietary, or, in the case of Internet of Things devices, because the origin of the source code is not known. Firmware can be extracted from an Internet of Things device, but even if its source code is publicly available, the device will not provide any indication of where its source code can be found or what libraries it uses. Therefore, security researchers must be able to perform analysis on binary code without relying on any additional information. Further complicating the matter, source code is frequently reused, meaning one vulnerability may be propagated to multiple devices with different processor architectures [2].

Because the same code may be compiled for a diverse range of processor architectures, a method to detect similarities in binary code across architectures is essential. When analyzing binaries found "in the wild," such as Internet of Things firmware, it is impossible to control for the CPU architecture of the binary. A cross-architecture approach allows information previously gathered from code from one architecture to be applied to code from another architecture. For example, possible vulnerable functions from the ARM architecture could be compared with known vulnerable functions from the x86 architecture. If an ARM function was determined to be similar to one of the vulnerable functions, it could be concluded that the ARM function contains the same vulnerability. The alternative to this cross-architecture similarity detection would be to start binary analysis from scratch whenever a binary from a different architecture needed to be analyzed.

The main challenge to detect code similarity across architectures is that the instructions in each architecture do not have a one-to-one correspondence with one another. Each processor is created with different idiosyncrasies and design trade-offs, and the processor's machine language instructions must work within this environment. This causes instructions from different architectures to have important differences, even if the instructions are meant to accomplish the same task. A processor with 64-bit registers may be able to perform a calculation with one instruction, while a processor with 32-bit registers may need to use multiple registers or write to the stack to perform the same calculation. A CPU architecture with a more complex instruction set may use one instruction to perform a set of operations that another architecture would require several instructions to perform. Because the instruction sets of different architectures do not exactly correspond with one another, we rely on machine learning to detect code similarity.

Multiple tools to accomplish this task already exist. However, they have limitations that can be detrimental in certain circumstances. Some of these tools examine select features of the code rather than the code itself, which can yield sub-optimal results. They may also only be able to analyze a complete function, which renders them useless if a partial

function must be analyzed. Tools that avoid these limitations require using multiple neural networks in concert, which complicates the use and evaluation of these tools. We intend to create a binary code similarity detection tool that analyzes the code itself and operates at a finer granularity than the function level without requiring the training and coordination of multiple neural networks.

## 1.2 Background

We utilize several technical concepts in this thesis. In this section, we describe these concepts.

### 1.2.1 Basic Block

A basic block is a section of code in which all instructions are guaranteed to be executed one after the other. There are no branches into or out of the middle of a basic block; it is only possible to branch into a basic block at its beginning. If a program branches to the beginning of a basic block, execution will continue through every instruction in the basic block until its end, when control may be able to branch away [3].

### 1.2.2 Ghidra P-code

P-code is a register transfer language used by the Ghidra reverse engineering tool to act as an intermediate representation for machine language instructions. This intermediate language is used so that Ghidra's analysis tools and any program utilizing the Ghidra API can function on binaries from a wide variety of processors [4].

Ghidra performs this function by translating each machine language operation into one or more P-code operations. This simplifies program analysis by reducing the set of instructions analysis tools must handle to 62 P-code operations, as opposed to every machine language instruction from multiple architectures. This also allows Ghidra tools to be

extended to any existing processor architecture as long as a translation from that processor's instructions to P-code operations is written. Additionally, translation to P-code makes explicit any implicit side effects of complex instructions [4].

### 1.2.3 Natural Language Processing

Natural Language Processing (NLP) is the name given to the collection of theories and techniques used to create machines capable of processing, representing, and utilizing human languages. This field relies heavily on machine learning to accomplish this, particularly neural networks. Some of the tasks included in this field are translation of languages, classification of various writings into categories, generation of summaries of human-language texts, and refining document searching using language patterns [5].

### 1.2.4 Long Short-Term Memory

A Long Short-Term Memory (LSTM) is a structure used in neural networks. An LSTM accepts sequences of data as input and processes one item in the sequence at a time, learning more as it progresses. Once it finishes processing the entire sequence, it outputs a numeric vector that represents the entire sequence. LSTMs are frequently used in NLP applications to learn how to meaningfully represent sentences by learning one word at a time. The LSTM returns vectors of numbers that represent the sentence's semantics, such that sentences with similar meanings produce vectors close to each other in value, while sentences with dissimilar meanings produce vectors far from each other in value [6].

Other neural network structures learn to represent sequences in a similar manner, but they prioritize data recently encountered in the sequence and tend to "forget" data from earlier in the sequence. The LSTM was designed specifically to address this shortcoming. It is capable of "remembering" the overall structure of the sentence throughout the process of embedding the sentence into a vector. Because of this ability, it is widely used for NLP tasks [6].

### 1.2.5 Siamese Neural Network

The Siamese neural network is a neural network architecture consisting of two identical ”sub-networks” connected to one another. This network accepts two inputs at once and outputs a score reflecting how similar the two inputs are. Given a dataset of pairs of inputs, each with a ground truth label reflecting whether or not the inputs are similar, the network can be trained in a supervised manner to recognize similar and dissimilar inputs [7].

The sub-networks are trained to express their given inputs as meaningful numeric vectors. Similar inputs will be represented as vectors close in value, while dissimilar inputs will be represented as vectors far from one another in value. Once the vectors are generated for each pair of inputs, a calculation is performed on the vectors to generate a similarity score. This score is compared with the ground truth similarity label to train the network. The sub-networks are updated during the training process, but they remain identical to one another [7].

## 1.3 Goals

In this thesis, we developed a tool for detecting the similarity of code basic blocks. We focus on basic blocks because they are the smallest units of a function that carry significant semantic information.

Security analysts frequently study executable binaries. The source code associated with these executable binaries is not necessarily accessible by, or known to, the analysts. Therefore, our system must be able to detect code similarity using only the binaries themselves, without relying on additional information.

The same source code can be compiled to multiple different CPU architectures. A mono-architecture code similarity detection tool has severely limited utility in an environment such as Internet of Things firmware analysis, where any number of architectures may be encountered. In contrast, a cross-architecture solution allows for information gath-

ered during the analysis of code compiled for one architecture, such as known vulnerable functions, to be used for analyzing code compiled for another architecture. Therefore, our system must be able to compare basic blocks in a cross-architecture manner.

When analyzing basic blocks, many manually selected features of the block can be obtained. However, it is difficult to determine whether the optimal set of features has been selected. Our system must use a neural network to be sure that the information considered relevant generates good results.

The instruction sets of two CPU architectures do not have a one-to-one translation. Even instructions that accomplish the same task will have slight variations in what they do. Many instructions make secondary changes to the execution environment that are not explicitly stated. Our tool must reduce the possible confusion from these differences by converting blocks into an intermediate representation before comparing them.

## 1.4 Organization

Chapter 1 outlines the motivation, background, and goals of this thesis. Chapter 2 describes work related to our system and how the work differs from our approach. Chapter 3 explains the design of the system in this thesis. Chapter 4 describes the implementation of our system. Chapter 5 lists our results and identifies future work to improve our system. Finally, chapter 6 concludes this thesis.

# Related Work

A great deal of work on cross-architecture code similarity detection already exists. This problem has already been the subject of several papers. In this section, we describe some of the more influential work on this subject and how our work approaches the problem differently.

Eschweiler et al. [8] created a system called discovRE, which converts a function into two sets of features: numeric and structural. The numeric features are a manually selected set of counts of attributes of a function (for example, number of function calls, number of basic blocks, number of local variables, and number of instructions) expressed as a vector. The structural features include the function's control flow graph (CFG), which is a graph showing how the function's basic blocks can branch to one another during the flow of execution. The structural features also include a set of features for each individual basic block, consisting of all applicable numeric features used previously as well as any string references or constants used in the basic block. This system assumes that these features will remain consistent across different CPU architectures.

DiscovRE searches a function against a database of previously analyzed functions for matches. Because CFG matching is computationally expensive, the system first compares the numeric features of the target function against the numeric features of the stored functions. This allows the system to only perform CFG matching on likely candidates. DiscovRE then uses a graph matching algorithm to compare the structural features of the target function against those of the stored functions.

One major difference between discovRE and our tool is that discovRE uses manually selected features, while our tool uses a neural network. While Eschweiler et al. were able to demonstrate that their choice of features was effective, it is difficult to determine whether this choice was optimal. Our tool allows the neural network to discover what is most relevant during training. Additionally, discovRE only works at the function level. Our tool matches basic blocks, so it could be used in a scenario where only part of a function is available for analysis, while discovRE could not. Finally, the process of matching CFGs one at a time produces serious scalability issues. Ruling out functions using the numeric features helps with scalability somewhat, but later research indicates that this practice harms the system's accuracy [9].

Feng et al. [9] uses machine learning techniques proven successful in the computer vision field to create a more scalable cross-architecture code matching tool. This tool, called Genius, expresses the functions it analyzes as a CFG with additional manually selected features associated with each basic block. These features include string and numeric constants appearing in the basic block, the number of function calls in the block, the number of instructions in the block, and the number of "offspring" the basic block has, meaning the number of blocks the block can branch to. A clustering algorithm is then run on these enhanced CFGs. The distance between a CFG and each centroid discovered by the clustering algorithm is considered the feature vector representing the CFG.

When a new function is analyzed by Genius, its CFG containing basic block features is created and its distance from each centroid discovered when training the stored functions is calculated. This vector is then used to calculate similarity with other functions: if the vectors of two functions are close in value, the functions are similar, while if the vectors differ widely in value, the functions are different. This tool differs from ours in that it relies on manually generated features and that it cannot be used on code fragments smaller than complete functions, much like discovRE.

Xu et al. [1] improved the performance of Genius by using a neural network instead of

a clustering algorithm to embed functions. This tool, called Gemini, generated an enhanced CFG from every function in its training dataset in a manner similar to Genius. Gemini then used these enhanced CFGs to train a Siamese network consisting of two Structure2Vec networks, which are used to learn embeddings for graphs. The training data for this network consisted of pairs of enhanced CFGs, where a pair of CFGs compiled from the same source code function into different architectures was considered a similar pair, while a pair of CFGs compiled from different source code functions was considered a dissimilar pair.

Gemini outperformed Genius in several metrics, including similarity detection accuracy, efficiency in creating CFG embeddings, and time to train the model. While Gemini, like our tool, uses a neural network instead of manually selected features, it shares the same limitation as previous tools that it cannot be used at a finer granularity than the function level.

Zuo et al. [2] developed a cross-architecture code similarity detection tool called InnerEye that can be used at the basic block level. This work served as the inspiration for our own. The major idea behind the design of this tool is that cross-architecture binary similarity detection shares many similarities to the field of Natural Language Processing. A given CPU architecture could be considered a language, and therefore attempting to find a function compiled for a different architecture from the same source code is analogous to a translation. Furthermore, individual machine language instructions could be considered ”words,” while basic blocks could be considered ”sentences.”

Using this understanding, Zuo et al. adapted tools that had proven successful in the field of Natural Language Processing to create InnerEye. This tool consisted of a Siamese network containing two Long Short-Term Memory networks, which accepted one ARM basic block and one x86 basic block as input and returned a similarity score as output. LSTMs are frequently used in NLP because they can process a sentence as a sequence of inputs, learning the sentence representation one word at a time. Similarly, InnerEye used LSTMs to learn basic block representations one instruction at a time.

Previous tools were only usable at the function level, in large part because of the difficulty involved in gathering a set of labeled basic block pairs. However, in order to train InnerEye, a method was devised to gather this data. The backend of the LLVM compiler was modified so that compiled binaries' basic blocks would be labeled. Regardless of the target architecture, the basic blocks compiled from the same section of source code would have the same label. This allowed for the creation of a dataset of similar and dissimilar cross-architecture basic block pairs for training InnerEye. This compiler modification was made publicly available at [10], and we use it to create our own dataset.

While our work is inspired by InnerEye, it differs in one crucial area. In order to train a model that encounters disparate inputs such as ARM and x86 instructions, these inputs must first be unified into a common space. InnerEye unifies the ARM and x86 instructions by creating and training another neural network to embed individual instructions into semantically meaningful vectors. Basic blocks consisting of sequences of these vectors are given to the Siamese network.

Our work unifies the disparate instructions by lifting machine language instructions into Ghidra P-code. This causes all of the instructions encountered by our tool to be expressed in the same language. An additional benefit is that translation into P-code makes any implicit side effects of an instruction explicit by translating instructions with side effects into multiple P-code instructions. Our intuition is that making these effects explicit will improve the neural network's ability to capture the semantics of each basic block. Finally, using P-code makes our model much simpler than if a second neural network was involved, making it easier to debug and diagnose problems.

# Design

Next, we describe the overall design of our system and the challenges we had to overcome to meaningfully represent binary basic blocks in a cross-architecture manner.

## 3.1 Problem Formulation

When the same source code is compiled into different architectures, the results can differ widely. Because this problem is similar to the problem of natural language translation, we leveraged machine learning techniques that have proved useful in this field [2]. However, the different instruction vocabularies used for different architectures can add noise to the training data. Therefore, we designed our system to lift executable code into an intermediate representation before learning to represent it in a cross-architecture manner.

One challenge of representing code in this manner is selecting the appropriate level of granularity for the representation. Certain existing models can represent represent an entire function at a time, while others represent each basic block of the function individually. In this thesis, we chose to represent basic blocks individually. We chose this in order to extend the utility of our model to representing fragments of entire functions.

## 3.2 System Overview

In order to train our neural network, we created a dataset of similar and dissimilar basic block pairs. We created this dataset by compiling source code with the LLVM compiler.

The source code is first compiled into LLVM intermediate representation, and this representation is then compiled into both ARM and x86 assembly language. We use a modified version of the LLVM compiler that labels basic blocks such that ARM and x86 basic blocks compiled from the same source code will share the same label.

The ARM and x86 blocks are submitted to the Ghidra reverse engineering tool in order to lift the instructions in the blocks to P-code, Ghidra's intermediate representation. The block labels associated with the ARM and x86 blocks are then applied to the P-code blocks. This allows us to create pairs of P-code basic blocks compiled from the same source code from different architectures by pairing up blocks with matching labels.

The cross-architecture pairs of P-code blocks are then fed to our neural network. This network consists of LSTM layers joined together in a Siamese architecture. The LSTM layers learn to express each P-code block as a vector of numbers such that blocks with similar meaning will be expressed as vectors of numbers with similar values and blocks with widely varying meaning will be expressed as numeric vectors with large distances between their values. The Siamese layer verifies this by calculating the Euclidean distance between the vectors of each basic block pair and calculating a similarity score using this distance. This score is a value between 0 and 1, with 1 meaning that the blocks are identical and 0 meaning that the blocks are completely different. Shown in Figure 3.1 is a graphical representation of the flow of data in our model, from the source code to the output of the Siamese network.

In contrast, shown in Figure 3.2 is a graphical representation of the flow of data in the InnerEye system [2]. This system differs from ours in that our system unifies ARM and x86 instructions into the same space by translating instructions into P-code, while InnerEye performs the same task using a separate instruction embedding neural network model. This model must be designed and trained separately, then incorporated into the larger system. Using P-code instead of another model simplifies our system. It also handles unexpected input more gracefully. For example, InnerEye's instruction embedding model

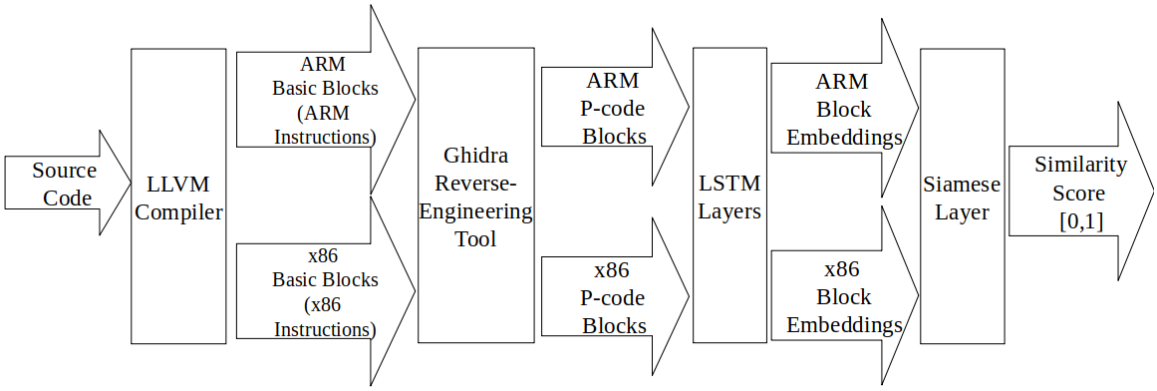


Figure 3.1: System Data Flow Overview

will be unable to meaningfully embed an assembly instruction it had not encountered during training. Our system translates assembly instructions to a much smaller set of instructions, reducing the likelihood of encountering an unknown input.

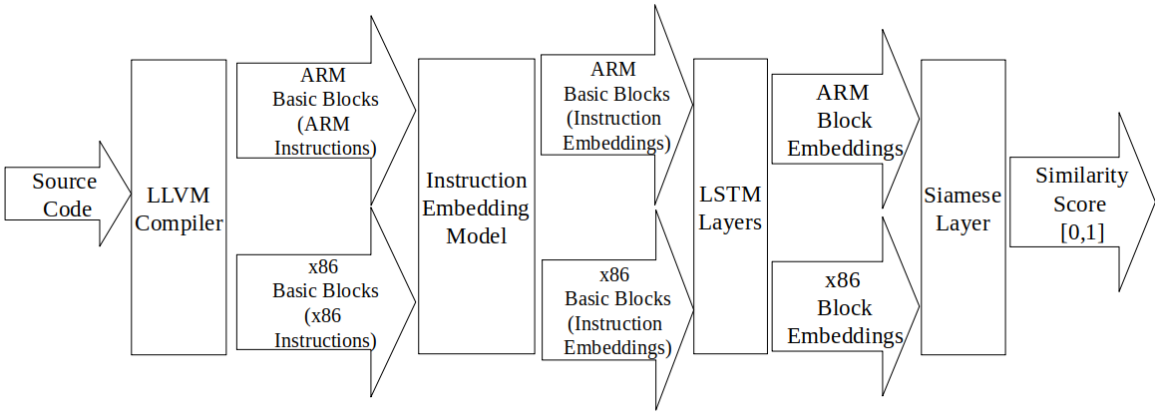


Figure 3.2: InnerEye Data Flow Overview

Our system design is explained in more detail in the following sections.

## 3.3 Training Dataset

A suitable dataset for training our model was not available, so we had to create one ourselves. Because the creation of this dataset was so important to this thesis, it is described here. In order to train a cross-architecture basic block embedding model, we collected a

dataset of labeled pairs of basic blocks. These pairs consisted of one ARM block translated into P-code and one x86 block translated into P-code, while the label described whether the blocks corresponded to the same source code. With this dataset, we trained a supervised model to calculate the similarity of basic block pairs.

The challenge in creating this dataset was determining which basic blocks corresponded to which sections of source code. The debug information included in the compiled executable files contained function names, but did not include any information relating to how the source code was compiled into basic blocks. To gather this data, we took advantage of a program provided by the creators of InnerEye [2] to modify the LLVM compiler to label basic blocks during compilation. This allowed us to gather pairs of matching basic blocks by matching these labels.

The LLVM compiler was modified by the creators of InnerEye [2] because of its segmentation of the compilation process. This compiler was designed to support a large number of ”front-ends” and ”back-ends”. A front-end is a module designed to accept and process a certain type of source code as an input, such as C++ code. A back-end is a module designed to produce assembly language code for a specific CPU architecture, such as ARM or x86. The LLVM compiler is able to use different combinations of front-ends and back-ends to provide a versatile compilation tool.

LLVM front-ends and back-ends achieve this modularity using an intermediate representation (IR). Front-ends accept source code and return an initial compiled output in LLVM IR. Back-ends accept this LLVM IR as input and return machine code for a CPU. We take advantage of this design by compiling source code files for our dataset into LLVM IR, then compiling the same IR file into both ARM and x86 assembly. We represent this graphically in Figure 3.3.

The modification to the LLVM compiler alters the back-ends in order to apply a label to each assembly basic block that reflects which LLVM IR basic block it was compiled from. The same label will be applied to any basic block compiled from an LLVM IR block

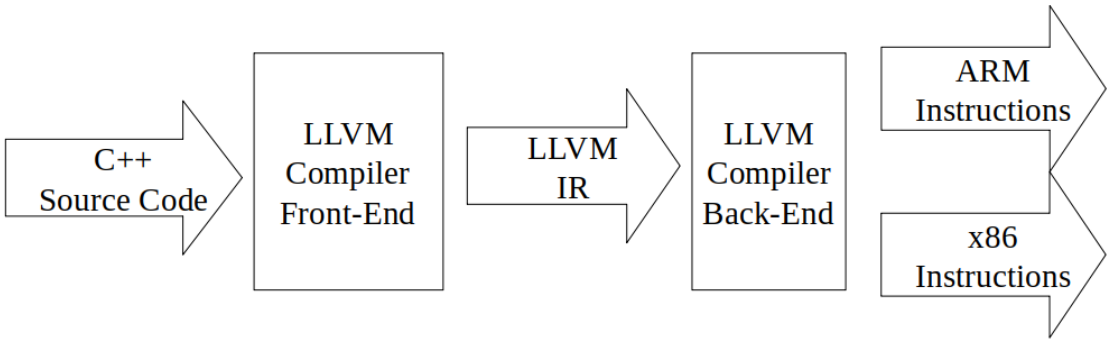


Figure 3.3: LLVM Compiler Data Flow Overview

regardless of the target CPU architecture. These labels provide us with the ground truth to determine whether blocks from different architectures are similar. We generate pairs of similar basic blocks by grouping pairs of blocks with the same label.

We also generate pairs of dissimilar basic blocks by randomly selecting two pairs of basic blocks and creating a new pair out of one pair's ARM block and the other pair's x86 block. We ensure that two similar pairs of blocks are not used to create a pair designated as dissimilar by calculating the Jaccard distance between the ARM blocks of both pairs. If the Jaccard distance is high, this means that the ARM blocks are similar, so these pairs are not used to generate a dissimilar pair.

## 3.4 Intermediate Representation

In order to reduce noise in the training data and make our inputs more uniform, we lift the basic blocks into an intermediate representation before feeding them to the neural network. We use Ghidra's built-in P-code representation for this lifting.

An additional benefit of this process is that it allows us to avoid the out-of-vocabulary (OOV) problem encountered during the creation of InnerEye [2]. The OOV problem, which is a well-known problem in Natural Language Processing, occurs when a neural network encounters a word it did not encounter during training. It is unable to represent the word in

a useful manner. InnerEye operated on machine learning instructions, and while this tool mitigated the worst of the problem by abstracting away constant values, memory addresses, string values, and function names, the problem still persists. For example, if InnerEye encountered the instruction, ”mov r1, r2” during training but not the almost identical, ”mov r2, r1,” it would not be able to represent the latter instruction. This, along with the fact that the x86 architecture alone contains hundreds of instructions without considering each possible combination of registers, means that an incredibly large vocabulary is required to avoid the OOV problem [11].

This is less of a problem when lifting instructions to Ghidra P-code. Firstly, P-code instructions have abstraction built in for both memory and registers. Every constant value is represented as, ”constant,” every memory value is represented as, ”ram,” and every register value is represented as, ”register.” [4] Secondly, there are much fewer total instructions in P-code than in ARM and x86 combined. The size of the vocabulary used for our tool was 222 instructions. Our vocabulary treated instructions with different operands as different instructions, even if the opcode was the same. For example, the instruction, ”int_add register, register” was treated as a different instruction than, ”int_add register, ram.”

The limitations of the LLVM compiler modification provide a challenge to this lifting of instructions. The modification to the LLVM compiler only produces basic block labels when the compiler is set to produce an assembly (.s) file as output instead of an object (.o) file. An object file is the typical output of a compiler. It is a binary file containing machine language instructions that can be linked into an executable binary. In contrast, an assembly file is a text file that contains assembly language instructions as human-readable strings.

While the LLVM compiler can only output basic block labels to the assembly file format, Ghidra cannot read this format or lift its instructions to P-code. Therefore, each source code file must be compiled into ARM and x86 architectures twice, once to produce the ARM and x86 assembly files and once to produce the object files. We created a Ghidra script to analyze an object file, read in the corresponding assembly file as a text file, and

match the instructions in each labeled assembly file basic block with its corresponding object file basic block. The script then lifts the object file basic block into P-code and saves the assembly file block's label with the P-code block. These P-code blocks can then be paired with P-code blocks from the files from the other architecture. Shown in Figure 3.4 is a graphical representation of the data flow of this Ghidra script. It shows the relevant features of the object file and the assembly file, as well as the features of the P-code blocks this script produces.

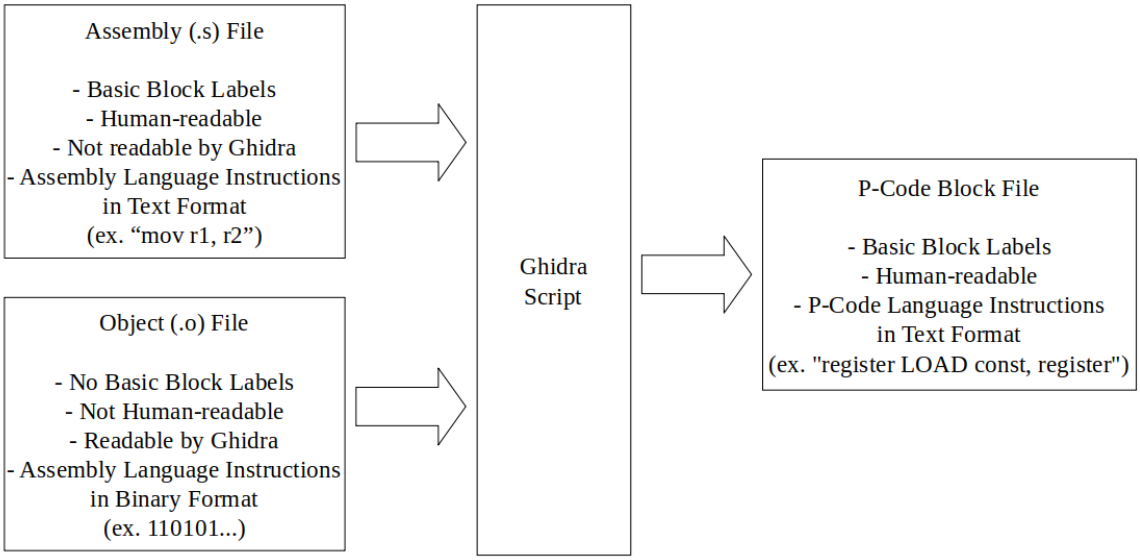


Figure 3.4: Creation of Labeled P-code Blocks

Another challenge in converting each basic block to P-code from its native representation is that the basic block labels in the assembly file do not correspond exactly to the basic blocks from the object file as detected by Ghidra. This is mostly the case in the x86 files, and appears to occur because the labeled basic blocks correspond to LLVM IR basic blocks, which are not necessarily divided into basic blocks in the machine language representation. For example, the basic block labels always make a distinction between the body of a "for" loop, the iteration condition of a "for" loop, and the incrementation of a "for" loop variable. The actual function may be compiled in such a way as to combine some or all of these operations without any branches between them, making them part of the same

basic block.

In order to align the basic blocks with their labels, we collect all of the labeled basic blocks associated with a function and compare them to the basic blocks from the object file that Ghidra collected for that function. If the sequence of opcodes in a labeled basic block match perfectly with the sequence of opcodes in one of Ghidra's basic blocks, the P-code for that basic block can be associated with the basic block's label. The Ghidra basic block can then be removed from further consideration.

The labeled basic blocks in a function that cannot be matched perfectly with a Ghidra basic block are assumed to be subsets of unmatched Ghidra basic blocks, which were labeled by the LLVM compiler based on semantic meaning of the instructions instead of control flow boundaries. The opcodes in the labeled basic block are checked against the opcodes in each unmatched Ghidra basic block. If the Ghidra basic block contains the pattern of opcodes in the labeled basic block as a subset of its opcodes, the P-code for only those machine language opcodes is associated with the basic block's label. These labeled P-code blocks can then be used to construct a dataset to submit to the neural network.

## 3.5 Neural Network

We train a Siamese neural network to embed P-code basic blocks into numeric vectors in such a manner that similar basic blocks have embeddings that are close to one another in value. Basic blocks are "similar" if the blocks' operations have the same semantic meaning, regardless of whether or not the blocks have the same CPU architecture.

A Siamese neural network accepts two inputs and a label expressing whether these inputs are similar or dissimilar (1 or 0, respectively). It is then trained to assign a similarity score to pairs of inputs. Our Siamese network includes a Long Short-Term Memory (LSTM) neural network that is trained to represent P-code basic blocks as numeric vectors. The LSTM network was selected due to its success in embedding natural language

sentences [2].

Each basic block in a pair is embedded into a numeric vector using the same LSTM network, and the Euclidean distance between the two vectors is calculated. This distance is converted into a similarity score by calculating the exponential function of the negated distance. This equation is shown in Equation 3.1, where ”dist” is the Euclidean distance between two vectors.

$$Sim\_score(dist) = \exp(-dist) \tag{3.1}$$

Equation 3.1 was also used by InnerEye to calculate the similarity score between two vectors [2]. This formula is used because as the distance between the vectors approaches 0, the result approaches 1. As the distance between the vectors grows larger, the result grows closer to 0. This similarity score is then compared with the label to train the LSTM.

# Implementation

We present a proof-of-concept implementation for our approach to creating a cross-architecture embedding scheme for function basic blocks. We describe the implementation of our tool for embedding these basic blocks in a semantically meaningful manner. This tool accepts a pair of basic blocks as input and produces a similarity score.

## 4.1 Dataset Generation

The source code we used for our dataset consisted of Arduino core libraries [12], the Arduino Cryptography Library [13], and GNU's coreutils programs [14]. In order to compile this source code for different architectures, we used the LLVM compiler. This compiler consists of several front-ends that compile source code into intermediate representation (IR) and several back-ends, each of which compiles LLVM IR into a different CPU architecture [2].

In order to gather labeled basic blocks from these compiled programs, we utilized the modification for the LLVM compiler provided by the creators of InnerEye [2]. This code altered the AsmPrinter class of the LLVM compiler backend [10]. This class handles the compilation of LLVM intermediate representation into a specified assembly language, and the modified version also assigns a unique label to each basic block reflecting which IR block it was compiled from. Assembly language blocks compiled from the same IR block for different architectures will have the same label.

Since only the AsmPrinter class was modified, these basic block labels will only be

applied when the LLVM IR is compiled to assembly (.s) files, as opposed to object (.o) files. Assembly files are human-readable text files containing assembly language instructions, while object files are compiled binaries.

## 4.2 IR Lifting and Labeling

Our tool lifts assembly language instructions into an IR to minimize noise that may result from idiosyncrasies in each architecture's instructions. LLVM IR cannot be used for this purpose, however, because the LLVM compiler does not possess the capability to lift compiled code back into IR. Instead, we use Ghidra's built-in IR lifter to convert assembly language instructions into P-code. Ghidra was selected because of its popularity with the reverse engineering community as well as its API support for gathering basic blocks from an executable file and lifting them into an IR.

One limitation of using Ghidra is that it cannot accept assembly files or lift instructions in these files into P-code. As mentioned previously, the LLVM compiler can only produce labeled basic blocks in assembly file format, which is plain text instead of machine-readable. To address this limitation, we also compile the source code for our dataset into object files using LLVM. Therefore, we must compile each source code file we wish to add to the dataset into four inputs: an ARM assembly file, an ARM object file, an x86 assembly file, and an x86 object file. We then use Ghidra's scripting support to align the basic block labels in the assembly file with the P-code representation of the basic blocks in the object file.

First, we read in the assembly file to collect each basic block's label and opcodes. We then use the Ghidra API's FunctionManager class to iterate through every function in the object file [15]. This class allows access to the Function objects that represent every function in the object file. By cross-referencing a Function object with the Ghidra API's SymbolTable object associated with the file, we can find the symbols associated with each

function, including the function's name [16]. Because each basic block label in the assembly file contains the name of the function to which the basic block belongs, we can check the labeled basic blocks against only the object file basic blocks that belong to the function with the same name.

We then gather the basic blocks for each function from the object file. We accomplish this using the Ghidra API's BasicBlockModel class, which allows us to access the basic blocks associated with a range of memory addresses [17]. A function's basic blocks can be gathered from this model by querying the appropriate Function object for its address range and submitting this range to the BasicBlockModel. This will return an array of CodeBlock objects, which represent the function's basic blocks [18]. We can then use these CodeBlock instances to iterate through the instructions of each basic block and gather their opcodes.

We check labeled basic blocks against object file basic blocks by comparing the blocks' opcodes. However, this is only possible after some preprocessing of the labeled data. One problem is that Ghidra disassembles the object file instructions of any x86 binary using the Intel syntax instead of the AT&T syntax. These are two different ways to display x86 assembly instructions. If the assembly file containing the basic block labels is not compiled with explicit instructions to output opcodes using the Intel syntax, then the labeled basic block opcodes will not match the object file's basic block opcodes purely due to syntactical differences.

Another discrepancy that must be addressed is that very rarely, the labeled basic block from the assembly file will use a different opcode than Ghidra shows for the object file. These different opcodes always have the same semantic meaning, but the syntactic difference can disrupt opcode comparisons. For example, some labeled x86 basic blocks end with the instruction, "je," while its corresponding basic block in the object file ends with the instruction, "jz." Both of these instructions mean to check the zero flag and jump to a new address if the flag is set. However, the x86 architecture includes them both to improve readability [19]. This can be fixed by manually changing every instance of the "je"

instruction in the labeled basic block to ”jz” so that both blocks match syntactically.

Once we have gathered the labeled basic blocks from the assembly file, preprocessed these blocks' opcodes, and gathered the opcodes of the object file's basic blocks, we are ready to begin aligning the blocks. For each function, we iterate through the function's basic blocks and match each block's opcodes against the opcodes of each labeled basic block belonging to the corresponding function in the assembly file. Upon matching an object file's basic block with a labeled basic block, we convert the object file block's instructions into P-code and assign the label to this P-code block.

We first attempt to find all exact matches. We match the opcodes of each function basic block against the opcodes of each labeled basic block from the same function. If a basic block's opcodes perfectly match a labeled block's opcodes, that block's CodeBlock instance is used to convert all of the block's instructions to P-code. The CodeBlock instance allows us to access the Instruction object associated with each instruction in the block. The Instruction class includes the getPcode() function, which lifts the given instruction into P-code operations [20]. This block of P-code operations is assigned the matching label, and both the function block and the labeled block are removed from further comparison.

After all exact matches are found, the unmatched labeled basic blocks are compared with the unmatched function basic blocks to find any labeled basic blocks that are subsets of function basic blocks. For each labeled basic block, we convert every function basic block's opcode list to a list of n-grams, where n is the length of the labeled basic block's opcode list. For example, if the labeled basic block consisted of the opcodes [”mov”, ”add”, ”jmp”] and the function basic block consisted of the opcodes [”sub”, ”mov”, ”add”, ”jmp”], the function basic block would be converted into the following list of 3-grams: [[”sub”, ”mov”, ”add”], [”mov”, ”add”, ”jmp”]]. The n-gram list is then checked to see if it contains a sequence that matches the labeled basic block's opcodes. In this case, the second entry in the n-gram list matches the labeled basic block's opcodes.

In order to discover the closest possible match, this check is performed with every

function block, even after a match is found. For each function block that has a match with the labeled block, its opcode list is converted into a 2-gram list, the labeled basic block's opcode list is converted into a 2-gram list, and the Jaccard distance of these lists is calculated. The basic block label is assigned to the function block with the highest Jaccard distance. The P-code for this block is extracted in the same manner described above, but only the P-code for the instructions corresponding to the instructions in the labeled basic block are gathered. Because function blocks with no exact match may contain multiple labeled blocks, function blocks that are matched in this way are not removed from comparison.

The input for this IR lifting process is a compiled program's assembly file and object file for the same architecture. The output is a file containing labeled P-code blocks. This process must be performed independently for the program's assembly file and output file for the other architecture. After the basic blocks from both architectures have been lifted in this way, we can generate labeled basic block pairs. This is a simple matter of reading in both output files, bundling the P-code blocks with the same label together, and saving the output to a new file.

In order to better train the neural network, we generate negative pairs, or pairs in which the ARM basic block does not match the x86 basic block. These pairs are labeled with a 0, while the matching pairs are labeled with a 1. In order to generate a negative pair, two pairs are randomly selected from the list of positive pairs. The ARM block from the first pair is matched with the x86 block from the second pair. In order to ensure that we do not accidentally group two similar blocks and label them as dissimilar, we calculate the Jaccard distance between both ARM blocks. If the distance is less than 0.5, we can conclude that the two block pairs are truly dissimilar. After the negative pairs are added to the dataset, we are ready to train the neural network.

## 4.3 Neural Network

We implemented the neural network using the Keras library with a Tensorflow backend [21]. This library was chosen because of its modularity as well as for its abundance of documentation.

We designed the neural network to be a Siamese network, meaning that it accepts an ARM basic block lifted into P-code and an x86 basic block lifted into P-code as input and outputs a similarity score. The maximum score that can be assigned is 1, for identical blocks, while the minimum score that can be assigned is 0, for completely dissimilar blocks.

Because of this design, the first layers in the network are two Input layers [22]. These layers each encapsulate a basic block submitted to the network. The next layer is a Masking layer [23]. Basic blocks must be padded to uniform length to be passed to an Input layer, and the Masking layer ensures that these padding values do not affect the training of the neural network or the network's representation of the basic blocks. Next is an LSTM layer [24]. This layer accepts a basic block as input and learns to represent it in a semantically meaningful manner. We add a second LSTM layer to further learn from information generated by the first LSTM. This layer outputs a semantically meaningful numeric vector for each basic block.

During training, each basic block in a pair will be embedded by LSTM layers with the same weights. We then calculate a similarity score with these vectors to inform the training of the LSTM layers. To accomplish this, we apply Equation 3.1 to the Euclidean distance between the two vectors. First, the difference between the two vectors is calculated. This is performed in its own Subtract layer due to difficulties passing both vectors to the next layer [25].

Finally, these differences are used to calculate the Euclidean distance between the two vectors. Equation 3.1 is applied to this distance to produce the similarity score. A Lambda layer is used for this calculation, which is a Keras layer that allows custom operations to be performed on data in a neural network [26]. Shown in Figure 4.1 is an overview of the

model's structure.

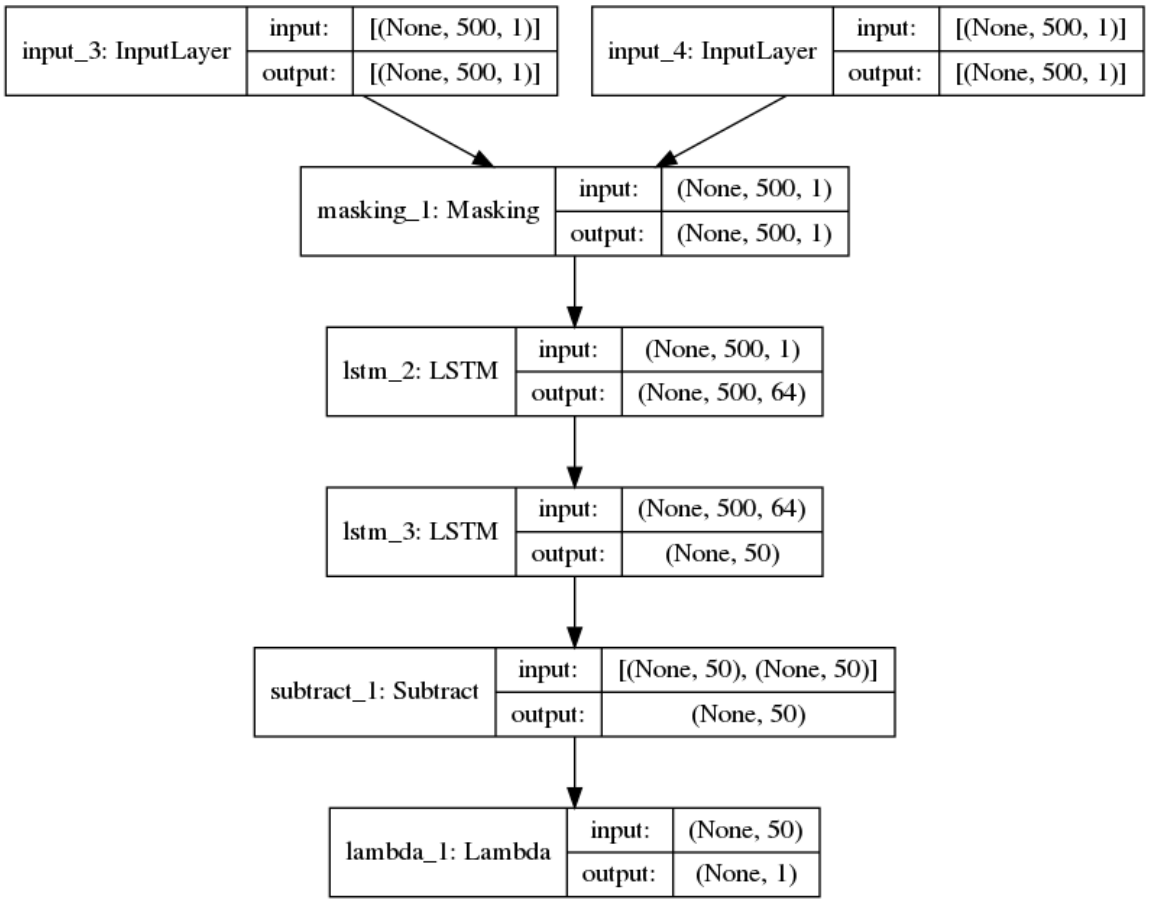


Figure 4.1: Siamese Neural Network Model Overview

The input and output data associated with each layer in Figure 4.1 shows the dimensionality of the layer's input and output, with the word, "None," at the beginning of each input and output dimension signifying that multiple arrays of this dimension will be passed to the layer during training. For example, each Input layer is passed a two-dimensional array representing a P-code basic block. This block is padded to a length of 500, and each instruction must be stored in its own array. This conveys to the LSTM that the instructions should be read as a sequence instead of all at once. Because the Input layer does not perform any processing on the basic block, the output dimension is the same as the input dimension.

Similarly, the Masking layer takes the basic block as an input and ensures that the

padding values do not affect the training of the model [23]. However, this layer does not alter the block data itself. The two LSTM layers accept this basic block as an input and represent the block as a numeric vector of length 50, which is returned as output. The Subtract layer calculates the difference between the ARM block's representation vector and the x86 block's representation vector. It returns this difference as a vector of length 50. Finally, the Lambda layer calculates a similarity score using this difference. It returns a single value between 0 and 1 reflecting the similarity between the two blocks.

# Evaluation

We collected a dataset of basic block pairs for our tool. We set aside a portion of this dataset for testing.

## 5.1 Dataset

As mentioned previously, to create the dataset for training and testing our neural network, we used a modified compiler to label basic blocks during compilation. We then compiled source code from Arduino core libraries [12], the Arduino Cryptography Library [13], and GNU's coreutils library [14].

The Arduino core libraries are C++ files included with the Arduino Integrated Development Environment (IDE). These libraries include dependencies required for any Arduino program to function. Also included are libraries governing the operation of Arduino hardware, such as the USB port, Ethernet port, LCD screen, and servos.

The Arduino Cryptography Library enables Arduino devices to perform a variety of cryptography operations. These operations include encryption and decryption, cryptographic hashing, and random number generation. This library contains a wide variety of algorithms and configuration options for these tasks.

GNU's coreutils library includes the code for a large number of programs that are typically found as command-line tools in most GNU systems. These programs include the command-line tools, "chmod," "chown," "cat," "echo," "ls," "rm," "touch," and many others. The programs in this library cover a wide variety of applications, but were all

included because they were considered to be necessary tools for interacting with the GNU operating system.

We compiled the source code from these libraries into both ARM and x86 assembly in order to obtain labeled basic blocks. However, the labeled basic blocks could not be read by Ghidra. In order to lift the binary instructions to Ghidra's P-code intermediate representation, we also compiled these source code files into ordinary ARM and x86 binaries that were readable by Ghidra. We associated the P-code translations of the basic blocks in these binaries with the corresponding labels from the assembly files. We then grouped the P-code representations of the ARM and x86 blocks with the same label into pairs and labeled the pairs with a 1, signifying that each pair contained matching basic blocks. We collected 27,828 positive pairs in this manner. We also generated an equal number of negative pairs by matching the ARM block of a randomly selected pair with the x86 block of another randomly selected pair. These pairs were labeled with a 0.

Shown in Table 5.1 are several characteristics of each library in this dataset. These characteristics are the number of source code files in the library, the number of functions in the library, the number of basic block pairs gathered from the library, and the average number of P-code instructions in each ARM and x86 block in the library.

Table 5.1: Dataset Features by Library

| | **Arduino Core Library** | **Arduino Cryptography Library** | **Coreutils** |
|---|---|---|---|
| **Files** | 132 | 33 | 102 |
| **Functions** | 2224 | 405 | 1175 |
| **Basic Block Pairs** | 8125 | 1632 | 18069 |
| **Average ARM Block Length (P-code)** | 41 | 71 | 31 |
| **Average x86 Block Length (P-code)** | 30 | 62 | 22 |

The Arduino Core Library and the Coreutils library have comparable numbers of source code files, but the Arduino Cryptography Library has significantly fewer files than the other two libraries. This library makes the smallest contribution to the dataset. Interestingly, the Arduino Core Library has approximately twice as many functions as the Coreutils library, but the Coreutils library contributes well over twice as many basic block

pairs to the dataset as the Arduino Core Library. This is likely because the functions in the Coreutils library interact with the operating system to gather data and perform operations on this data, while the Arduino Core Library functions largely perform low-level operations on hardware peripherals. Function complexity expresses itself as a high number of basic blocks: the more tasks a function must complete and the more varied the function's input data can be, the more contingencies the function must address. Addressing contingencies involves checking conditions to determine what course of action to perform next, which generates a large number of basic blocks.

Additionally, many of the Coreutils functions must change their functionality based on user input, such as the arguments passed into command-line tools. The Arduino Core Library functions operate at too low a level to deal with user input. This added complexity would result in functions from the Coreutils library having more basic blocks analyzing the user input and handling each possible variation in how to operate.

The Coreutils library contributes the largest amount of basic block pairs to the dataset, but the blocks from this dataset are the smallest on average. In contrast, the Arduino Cryptography Library contributes the fewest basic blocks, but they are the largest on average. This is because each Coreutils function performs many tasks, but none of these individual tasks are particularly computationally intensive. On the other hand, the each function in the Arduino Cryptography Library performs a small number of computationally intensive tasks.

Consider an example program from each library. The Coreutils "ls" program must analyze any command line arguments, gather the appropriate file names, order the file names as specified, and print the file names to the command line [27]. These steps may require frequent condition checking, thus generating a high number of basic blocks, but they do not require many operations on the data at each step, thus generating basic blocks with few instructions. An encryption function from the Arduino Cryptography Library must accept data and perform mathematical operations on it to encrypt it. There are a

large number of these operations, and they are performed with little variation. Other than loop checking, there are not many conditions that must be checked. This generates a small number of blocks containing a large number of instructions.

## 5.2 Effectiveness

We trained our Siamese neural network using the ten-fold training method. In this method, the dataset is split into ten equal groups, or folds. Nine of the folds are used to train the network, while one fold is kept aside for testing. Training is performed ten times, keeping a different fold aside for testing each time.

In addition to setting aside 10% of the dataset for testing, we set aside another 10% of the dataset as validation data. Validation data is not used to train the neural network, but is instead used to evaluate the performance of the network on non-training data at the end of each training epoch [28]. This helps to ensure that the neural network is not being trained to over-fit or under-fit the training data.

We evaluated the performance of the network during training using loss and accuracy metrics. The loss function is a function that the network is being trained to minimize [29]. The lower the loss score, the better the network's performance. At the end of each training epoch, the loss function is computed for both the training data and the validation data. Shown in Figure 5.1 is the plot of the loss calculations for every epoch in the training of the best-performing fold. This plot was generated with scikit-learn [30].

In Figure 5.1, the training and validation loss scores remain close to one another, suggesting that the network is neither over-fitting nor under-fitting the data. The loss scores steadily decrease as the network is trained. Increasing the number of training epochs would likely lower the network's loss further, but this was infeasible due to the large time cost of training the network.

We also evaluated the performance of the network by calculating the training and

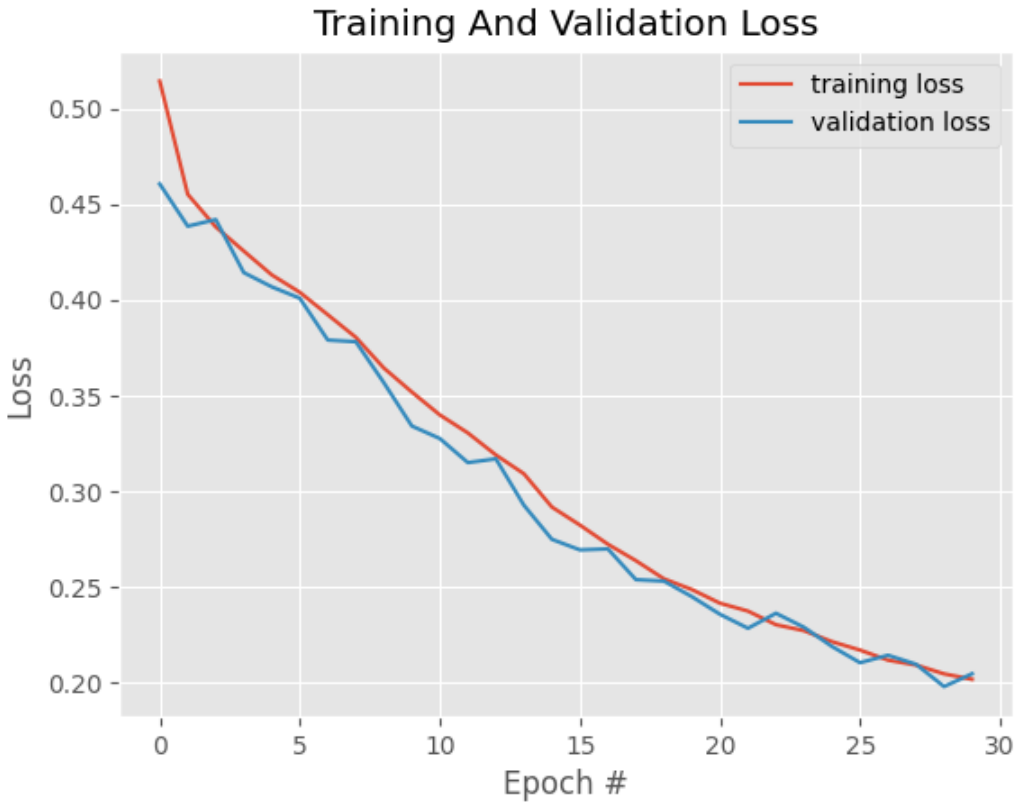


Figure 5.1: Neural Network Loss Curve

validation accuracy at the end of each epoch. The accuracy metric involves determining how close the network's similarity score prediction for each basic block pair was to the ground truth label. The better the network performs, the higher the accuracy score will be. As with the loss metric, we calculate the accuracy metric for the training and validation data in order to ensure that the network is not over-fitting or under-fitting the data. Shown in Figure 5.2 is the plot of the accuracy calculations by epoch in the training of the best-performing fold. This plot was generated with scikit-learn [30].

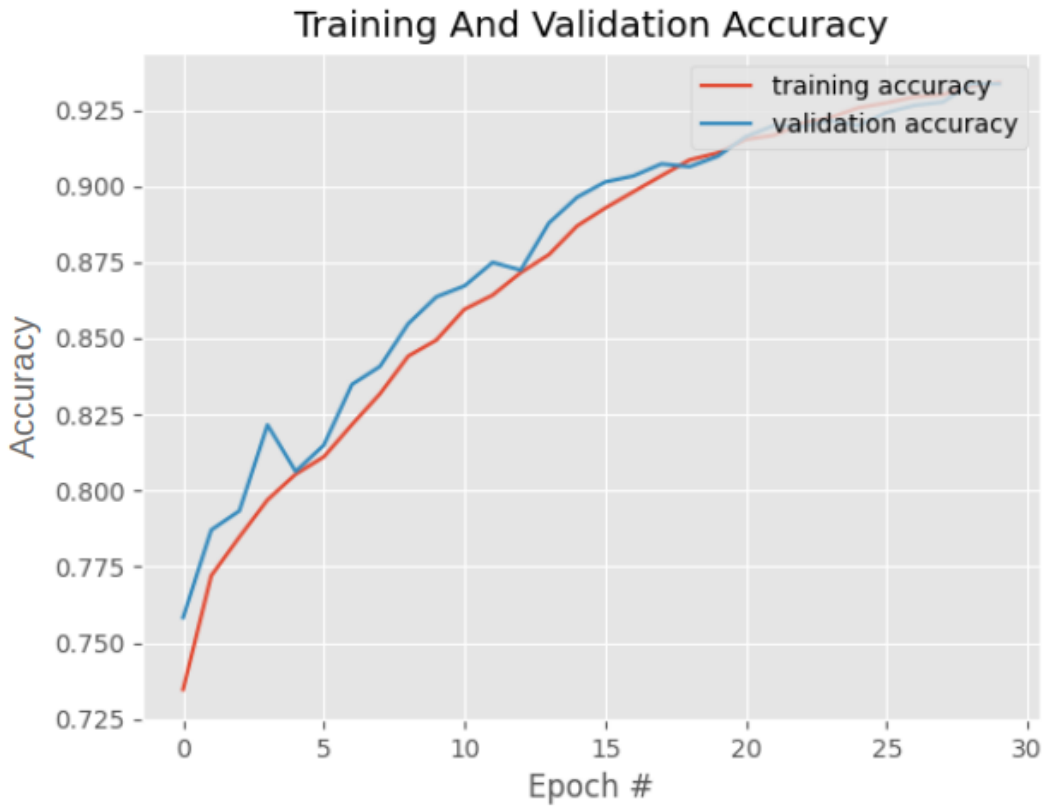


Figure 5.2: Neural Network Accuracy Curve

In Figure 5.2, the training and validation accuracy scores remain close to one another, suggesting that the network is neither over-fitting nor under-fitting the data. The accuracy scores increase as the network is trained, exceeding 93% for both datasets. Further training

the model may further increase the accuracy, but this was infeasible due to the large training time of each network.

We tested each trained network by submitting each basic block pair in the test dataset as input and comparing the network's output to the ground truth label. Because the network can output a similarity score with any value between 0 and 1, a threshold must be established to determine which prediction values count as a negative prediction and which values count as a positive prediction. A Receiver Operating Characteristic (ROC) curve is a graphical representation of the true positive rate and false positive rate of multiple thresholds. Each point on the curve represents a different threshold, while the point's position along the Y axis represents the true positive rate at that threshold and its position on the X axis represents its false positive rate. The overall quality of the network is judged by the area under the curve (AUC) of this graph. The closer the AUC is to 100%, the better the network [31]. Figure 5.3 shows the ROC curve of the best-performing neural network. This image was generated with scikit-learn [30].

Figure 5.3 shows an AUC of 98.03%. While this is indicative of good performance, showing the network's accuracy using multiple thresholds simultaneously is a highly abstracted metric. In order to show the performance of this model more concretely, we have performed the same test using a threshold of 0.5. A similarity score of less than 0.5 is considered a prediction of 0, while a score of greater than or equal to 0.5 is considered a prediction of 1. The count of true positives, true negatives, false positives, and false negatives for this threshold is shown in Figure 5.4. This image was generated with scikit-learn [30].

In Figure 5.4, the number of true negatives is shown in the top left square, the number of false negatives is shown in the top right square, the number of false positives is shown in the bottom left square, and the number of true positives is shown in the bottom right square. The neural network has a detection rate of 93.1% and a false positive rate of 6.5%.

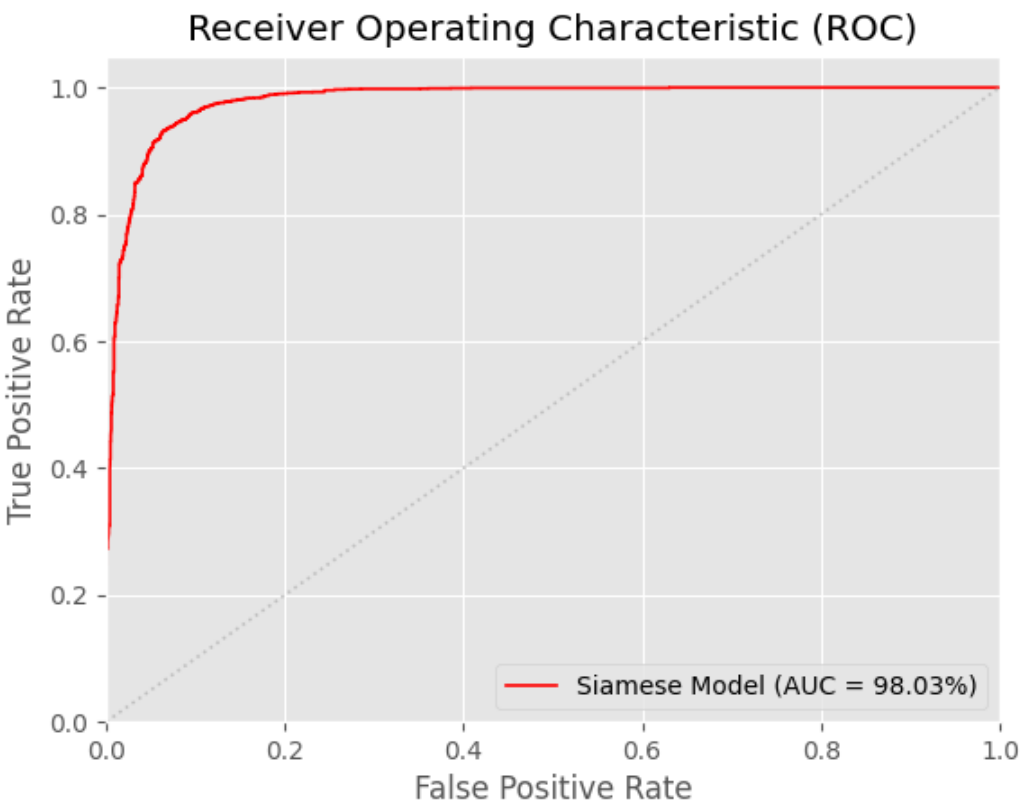


Figure 5.3: Neural Network ROC Curve

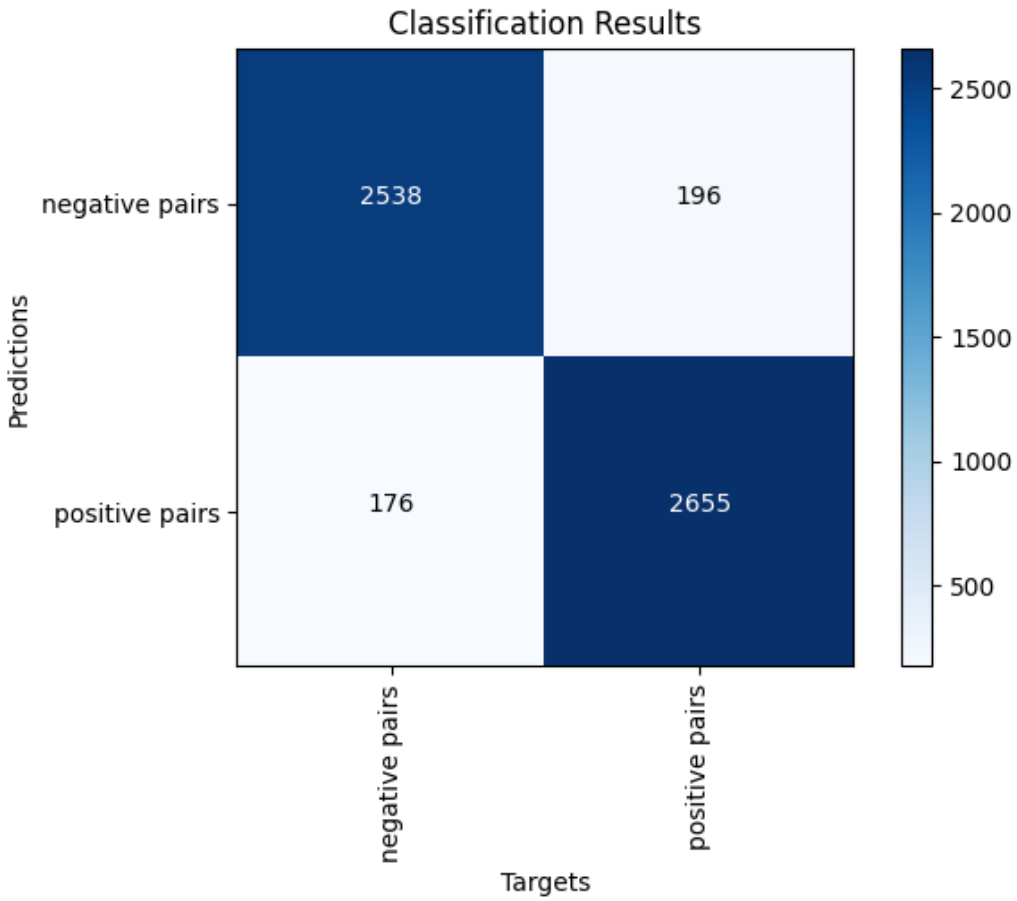


Figure 5.4: Neural Network Classification Matrix

## 5.3 Performance

The ten-fold training took place on a machine running Ubuntu 18.04 with 16 GB of RAM. Each neural network was trained for 30 epochs. Training took approximately 22 hours, or 2.2 hours per neural network.

The neural network trains by updating the weights associated with various internal parameters. These parameters and their weights are used to transform input as it propagates through the network. In our case, these parameters determine how basic blocks are transformed into numeric vectors. In general, a neural network can perform more precise calculations if it has more parameters. Shown in Figure 5.5 is a system overview of the neural network that shows each layer of the network in order as well as the shape of the layer's output and the number of parameters associated with the layer.

| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_1 (InputLayer) | [(None, 500, 1)] | 0 | |
| input_2 (InputLayer) | [(None, 500, 1)] | 0 | |
| masking (Masking) | (None, 500, 1) | 0 | input_1[0][0]<br>input_2[0][0] |
| lstm (LSTM) | (None, 500, 64) | 16896 | masking[0][0]<br>masking[1][0] |
| lstm_1 (LSTM) | (None, 50) | 23000 | lstm[0][0]<br>lstm[1][0] |
| subtract (Subtract) | (None, 50) | 0 | lstm_1[0][0]<br>lstm_1[1][0] |
| lambda (Lambda) | (None, 1) | 0 | subtract[0][0] |

Total params: 39,896
Trainable params: 39,896
Non-trainable params: 0

Figure 5.5: Neural Network System Parameter Overview

This network contains 39,896 parameters that can be trained. Only the LSTM layers contain these trainable parameters because the other layers should not be updated with

model training. The Input layers only accept basic blocks and do not perform calculations on them. The Masking layer filters out the padding so that it does not affect the block's embedding. The Subtract and Lambda layers exist to transform block embeddings into similarity scores. This process should not be updated over the course of training. A network of this size takes up 520.8 kB of disk space.

# Discussion

We have shown that our system can perform cross-architecture basic block similarity detection with a high degree of accuracy. We were able to demonstrate this at a high level using a large number of possible thresholds and at a low level using the threshold of 0.5. This threshold was arbitrarily chosen; examining the data further could show that a threshold could be chosen that increases the system's accuracy.

One possible source of future work is further tuning the hyperparameters of our neural network. It is possible that experimenting with different values for the number of training epochs, the size of the batches of block pairs submitted to the network at once for training, the standard block length that every basic block must be padded or truncated to reach, and the length of the output vector of each LSTM layer could cause the network to produce even better results. However, due to the complexity of experimenting with all of these variables simultaneously and the long training time for this network, we did not tune these hyperparameters extensively.

Despite our system's high accuracy, it remains limited in its utility while it can only detect basic block pairs. The execution flow of a function is a valuable source of information for code similarity detection. While it is out of the scope of this thesis, an important step of future work is to build on this system to enable it to embed the control flow information of functions. If this can be accomplished without sacrificing our system's ability to embed individual basic blocks, our system will become an incredibly useful tool.

# Conclusion

In this thesis, we have outlined our cross-architecture binary code similarity detection tool. We have shown that our tool can detect semantically similar basic blocks with a high degree of accuracy. We have also identified future work to further improve system accuracy by refining the network's settings and to expand the system's capability to include function-level control flow information.

# Bibliography


[1] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 363–376, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[3] John L Hennessy, David A Patterson, and Krste Asanović. *Computer architecture : a quantitative approach.* Morgan Kaufmann series in computer architecture and design. Morgan Kaufmann/Elsevier, 2012.

[4] P-code Reference Manual. https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/pcoderef.html.

[5] K. R. Chowdhary. *Natural Language Processing*. Springer India, New Delhi, 2020.

[6] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(4):694–707, 2016.

[7] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993.

[8] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*, 2016.

[9] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 480–491, New York, NY, USA, 2016. Association for Computing Machinery.

[10] AsmPrinter.cpp. https://github.com/nmt4binaries/nmt4binaries.github.io/blob/master/download/AsmPrinter.cpp.

[11] Jim Ledin. *Modern Processor Architectures and Instruction Sets*. Packt Publishing, 2020.

[12] Arduino Downloads. https://www.arduino.cc/en/software.

[13] Arduino Cryptography Library. https://github.com/rweather/arduinolibs.

[14] Coreutils - GNU Core Utilities. https://www.gnu.org/software/coreutils.

[15] FunctionManager API Documentation. https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/FunctionManager.html.

[16] SymbolTable API Documentation. https://ghidra.re/ghidra_docs/api/ghidra/program/model/symbol/SymbolTable.html.

[17] BasicBlockModel API Documentation. https://ghidra.re/ghidra_docs/api/ghidra/program/model/block/BasicBlockModel.html.

[18] CodeBlock API Documentation. https://ghidra.re/ghidra_docs/api/ghidra/program/model/block/CodeBlock.html.

[19] Conditional Jumps. https://riptutorial.com/x86/example/20470/conditional-jumps.

[20] Instruction API Documentation. https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Instruction.html.

[21] François Chollet et al. Keras. https://keras.io, 2015.

[22] Input Object. https://keras.io/api/layers/core_layers/input.

[23] Masking Layer. https://keras.io/api/layers/core_layers/masking.

[24] LSTM Layer. https://keras.io/api/layers/recurrent_layers/lstm.

[25] Subtract Layer. https://keras.io/api/layers/merging_layers/subtract.

[26] Lambda Layer. https://keras.io/api/layers/core_layers/lambda.

[27] ls(1) - Linux man page. https://linux.die.net/man/1/ls.

[28] Train a Keras Model. https://keras.rstudio.com/reference/fit.html.

[29] Losses. https://keras.io/api/losses/.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[31] Wanhua Su, Yan Yuan, and Mu Zhu. A Relationship between the Average Precision and the Area Under the ROC Curve. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15, page 349–352, New York, NY, USA, 2015. Association for Computing Machinery.