Cross Architecture Function Matching

A Major Qualifying Project Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In Partial Fulfillment of the Requirements for the

Degree in Bachelor of Science

In

Computer Science

On October 16, 2020

By: Samantha Woodland

WPI Advisors: Professor Robert Walls, Professor Lorenzo DeCarli

Sponsor: The MITRE Corporation

Approved for Public Release;
Distribution Unlimited. Case Number 20-3267
© 2021 The MITRE Corporation. ALL RIGHTS RESERVED.

Abstract

Malware almost always comes without the source code; therefore, it is necessary for reverse engineers to examine malware binaries at the assembly level. Malware authors frequently re-use pieces of code, therefore new malware can often overlap with other already analyzed binaries. Function matching assists in binary diffing in that it reduces the manual labor required to examine each file and assists in identifying the differences between the two. If functions that have been previously analyzed can be identified in new malware, those results are helpful in increasing the speed and accuracy of an analyst's reverse engineering. However, function matching becomes more difficult when matching across different architectures. Although the binaries may have been compiled from the same source code, the machine instructions cannot be compared directly, due to different instruction sets, calling conventions, register sets, etc. among different architectures.

Our matcher, the PCodeMatcher, uses PCode, an intermediate representation language developed by the NSA for use in Ghidra, their reverse engineering framework. PCode is used by Ghidra to decompile machine instructions into C code independent of architecture. Using a fuzzy string-matching technique, we were able to match 70% of functions in our cross-architecture sample set. We believe that, with further development, function matching using PCode could be a valuable tool for analyzing binaries in reverse engineering.

Acknowledgements

Special thanks to the following for all their support regarding this project:

The MITRE Corporation for the sponsorship of this project:
Carlos Cheung, Pete Brown, Peter Lucia, Corre Steele

Professor Robert Walls, Professor Lorenzo DeCarli

Table of Contents

Abstract i

Acknowledgements ii

Table of Contents iii

List of Figures v

List of Tables v

Chapter 1: Introduction 1

Chapter 2: Background 3
- 2.1 Computer Architectures and Compilers 3
- 2.2 Binary Analysis for Reverse Engineering 4
- 2.3 Ghidra 5
  - 2.3.1 PCode 6
- 2.4 Function Matching Tools 6

Chapter 3: System Design 8
- 3.1 PCode Translation 9
- 3.2 Fuzzy String Matcher 9
- 3.3 Similarity Threshold 10

Chapter 4: Methodology 12
- 4.1 Test Binaries 12
- 4.2 Ground Truth 13

Chapter 5: Results and Discussion 14
- 5.1 Data Plots for PCodeMatcher 14
- 5.2 Similarity Threshold 15
- 5.3 PCodeMatcher 16
- 5.4 Comparing against Industry Tools 17
  - 5.4.1 BinDiff 18
  - 5.4.2 Ghidra's Version Tracker 19

Chapter 6: Conclusions 21

6.1 Future Work 21
6.2 Lessons Learned 21
References 23

List of Figures

Figure 1: Source code used to generate assembly instructions in Figure 2 3

Figure 2: Source code compiled for ARM (left) and x86 (right).. 4

Figure 3: x86 Assembly instructions compared with PCode and C Code.................................... 6

Figure 4: Flow chart for the system 8

Figure 5: PCode Instruction.. 9

Figure 6: Normalized PCode strings for comparison 10

Figure 7: Levenshtein similarity ratio formula 10

Figure 8: PCodeMatcher results displayed in a ROC curve 14

Figure 9: PCodeMatcher results displayed in a PR curve... 15

Figure 10: Similarity threshold results displayed in confusion matrices.................................... 15

Figure 11: PCodeMatcher results displayed in a confusion matrix 16

Figure 12: Results from different function sizes displayed in confusion matrices...................... 17

Figure 13: Results from all three matchers displayed in a PR curve 18

Figure 14: BinDiff results displayed in a confusion matrix.. 18

Figure 15: Results from different function sizes displayed in confusion matrices...................... 19

Figure 16: Version Tracker results displayed in a confusion matrix 19

Figure 17: Results from different functions sizes displayed in confusion matrices 20

List of Tables

Table 1: Summary of test binaries 12

Chapter 1: Introduction

Binary analysis has many uses such as revealing flaws and vulnerabilities in code that can be revised to make a program more secure and less vulnerable. It can also aid in computer security by examining malicious code to understand its behavior to build better defenses against that malware [2]. This can be done through static analysis, by examining the binary directly in a non-runtime environment, or dynamic analysis, testing the binary during execution [15]. This project will focus on static analysis. Manual reverse engineering can take much time and effort on the part of the reverse engineer because it requires reading through all the assembly instructions to figure out what program does. If the binary contains thousands of functions, this process becomes exhausting.

Function matching tools can assist in this process by reducing the manual labor required to examine each file. Examining one binary and comparing that file to another binary whose behavior is undetermined can speed up the process of examining multiple files. This can be done using a function matching tool which identifies the similarities and differences in functions between the two binaries. Malware authors frequently re-use pieces of code, such as functions from libraries, therefore, malware can have pieces of code in common with other already analyzed binaries [13]. If functions that have been previously analyzed can be identified in new malware, those results are helpful in increasing the speed and accuracy of the reverse engineering. Common function matching tools used in industry include BinDiff and Ghidra's Version Tracker [6, 11].

A significant challenge is that the same source code can have multiple binary representations. The reason for this is that the source code must be reduced and translated according to the processor's instruction set during the compilation process. There are multiple different compilers, compiler optimization options, and target architectures making matching functions across binaries a complicated process [1]. Comparing two binaries of different architectures becomes more complicated because they will have different instruction sets, calling conventions, register sets, etc. Furthermore, individual assembly instructions from different architectures often cannot be compared directly due to the slightly different behavior of different architectures (side effects, instruction availability etc.)

This project investigates this problem using Ghidra, a reverse engineering framework developed by the NSA, and its intermediate representation of assembly code to match functions from binaries of different architectures. PCode is the intermediate representation language used by Ghidra to decompile machine instructions from architectures such as x86, MIPS, PowerPC, and ARM. Since PCode is an architecture agnostic representation of assembly instructions, our hypothesis is that it could be useful to build a function matcher that compares PCode instructions to accurately find matches across two binaries.

In this report, we present a background chapter that examines binary analysis for reverse engineering and the need for cross-architecture function matching. Following the background chapter, our system design chapter outlines the design and behavior of our function matcher, the PCodeMatcher. In our methodology chapter, we cover our testing methods and test dataset for the

PCodeMatcher. In the Results and Discussion chapter, we discuss the results of the PCodeMatcher and compare those results to industry tools such as BinDiff and Ghidra's Version Tracker. Overall. the PCodeMatcher was able to correctly match 70% of cross-architecture function pairs compiled from the same source to BinDiff's 33% and Ghidra Version Tracker's 23%.

Chapter 2: Background

In this chapter, we begin with a discussion of different architectures and compilers as well as an overview on cross-compiling. Next, we move into binary analysis for reverse engineering and an explanation of why function matching is important and how problems arise when function matching across architectures. Lastly, we cover Ghidra, a reverse engineering tool, and its intermediate representation language PCode, and other industry tools used in this project.

2.1 Computer Architectures and Compilers

An architecture is a set of rules and methods used to describe the functionality, organization, and implementation of computer systems. Computer architecture deals with balancing the performance, reliability, efficiency, and cost of a computer system [7]. An instruction set is a group of commands for the CPU in machine language [3]. Instruction sets between architectures may be similar or completely different. There are multiple different types of architectures, but this paper will focus primarily on CISC and RISC architectures, specifically ARM, MIPS, PowerPC and x86.

CISC stands for Complex Instruction Set Computer and includes architectures such as x86. RISC stands for Reduced Instruction Set Computer and includes architectures such as ARM and MIPS. CISC architectures have large complex instruction sets that include single commands capable of performing multi-step operations. RISC architectures have smaller reduced instruction sets consisting of many simple instructions that complete very small operations [3].

```
int func(int num) {
    if (num == 1) {
        return num;
    } else if(num > 1) {
        return 0;
    } else {
        return -1;
    }
}
```

Figure 1: Source code used to generate assembly instructions in Figure 2

```
str     fp, [sp, #-4]!
add     fp, sp, #0
sub     sp, sp, #12
str     r0, [fp, #-8]
ldr     r3, [fp, #-8]
cmp     r3, #1
bne     .L2
ldr     r3, [fp, #-8]
b       .L3
ldr     r3, [fp, #-8]
cmp     r3, #1
ble     .L4
mov     r3, #0
b       .L3
mvn     r3, #0
mov     r0, r3
add     sp, fp, #0
ldr     fp, [sp], #4
bx      lr
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
cmp     DWORD PTR [rbp-4], 1
jne     .L2
mov     eax, DWORD PTR [rbp-4]
jmp     .L3
.L2:
cmp     DWORD PTR [rbp-4], 1
jle     .L4
mov     eax, 0
jmp     .L3
.L4:
mov     eax, -1
.L3:
pop     rbp
ret
```

Figure 2: Source code compiled for ARM (left) and x86 (right)

Compilers are programs that translate source code into machine executable code. Compilers are important to consider because different compilers will have different effects on the final binary executable, a computer-readable file that is used to execute programs, such as different optimization options, algorithms, requirements, and language constraints, which will all influence the final executable [5]. Both the compiler and the target architecture are important to consider when analyzing binaries. Different compilers and different architectures will make a difference with the final output.

For this research, it is also important to consider cross-compiling. If we consider the *host computer* to be the computer we are compiling source code on and the *target computer* to be the computer we are running the final executable on, then a *native compiler* is one where the target and host are the same architecture and a *cross-compiler* is one where the target and host are different architectures. A *toolchain* is a set of compilers, linker, libraries, and any other tools needed to generate an executable. A linker is a computer program that takes one or more object files generated by a compiler and combines them into one executable file while a library is a collection of precompiled related routines that programs can use [5]. Most systems already have the correct toolchain for compilation for their own architecture but when cross-compilation is desired, it becomes necessary to install other toolchains for the correct architecture. We have used Dockcross, an open source toolchain for cross-compilation using Linux as a host system, to implement cross-compilation for this project [8].

2.2 Binary Analysis for Reverse Engineering

Binaries can contain multiple sections including executable code and data, such as initialized variables and constants. The main executable portion of the binary is generally known

as the text section which is made up of different functions that are included in the program. This project will examine functions within the text section of binaries and attempt to match them.

Conceptually, reverse engineering is the process of taking a man-made object apart to determine what purpose it serves. Reverse engineering is the process of examination; the system being investigated is not changed at all. For the purposes of cybersecurity, this concept is applied to determine the behavior and safety of binary executable files.

Binary analysis in cybersecurity can have many uses, including the identification of relationships between different malwares [14]. Reverse engineering a binary can reveal flaws or vulnerabilities in the code such as buffer overflows, memory corruption and other bugs [10]. When those flaws are revised, the program will be more secure and less vulnerable to attacks. When a piece of malicious code is examined, its behavior and structure can be understood, and better defenses can be constructed to protect computer systems against that malware [2]. Binary analysis can be done through static or dynamic analysis. Static analysis is the process of analyzing the internal structure of the binary through examination of the source code, byte code, or binaries to determine behavior or identification of security vulnerabilities. Dynamic analysis is where the binary is examined from the outside during execution [15]. This project will focus on static analysis only.

Binary diffing is a step in the reverse engineering process in which two files are examined to identify the differences in code [4]. It consists of comparing the semantic and syntactic differences between the two binaries. Automated function matching assists in binary diffing in that it reduces the manual labor required to examine each file and assists in identifying the differences between the two [2]. Malware authors frequently re-use pieces of code, such as functions from libraries, etc., therefore, malware can often overlap with other already analyzed binaries. If functions that have been previously analyzed can be identified in new malware, those results are helpful in increasing the speed and accuracy of the reverse engineering. Therefore, when the same source code has been compiled for different architectures, this can impede the binary diffing process because each binary often must be analyzed and compared manually because common binary diffing tools are of limited use when comparing binaries across architectures.

2.3 Ghidra

Ghidra is an open source software reverse engineering framework developed and maintained by the National Security Agency's (NSA) Research Directorate in support of the Cybersecurity mission [11]. Ghidra can be used to solve a variety of problems involving analysis of malicious code as well as identifying potential vulnerabilities in networks and systems. It includes a multitude of software analysis tools that allow users to analyze binaries compiled for different architectures on a variety of different platforms including Windows, MacOS and Linux. Capabilities include decompilation, scripting, disassembly, assembly, and graphing among other features. Ghidra supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes. Ghidra can also be extended through user-developed plug-in components and scripts using Java or Python.

2.3.1 PCode

PCode is an *intermediate representation language* (IRL) designed for reverse engineering applications. Intermediate representation is a representation of a program in between the source and target languages [9]. PCode can be interpreted by software to emulate behavior of different processors. PCode's IRL subtype is *register transfer language* (RTL), a type of IRL that specifies the semantics of machine instructions. In this context, semantics describe how an instruction of an architecture manipulates data in registers and memory. All data is manipulated explicitly, meaning instructions have no indirect effects. Individual PCode operations are designed to mirror typical processor tasks by translating individual processor instructions into a sequence of PCode operations. The set of PCode operations, known as opcodes, form a set of the arithmetic and logical actions performed by general purpose processors.

x86 Assembly Instruction	PCode	C Code

`00100774 e8 67 fe ff CALL printf`	`RSP = INT_SUB RSP, 8:8`
	`STORE ram(RSP), 0x100779:8`
	`CALL *[ram]0x1005e0:8`
		`printf("%d\n",local_18)`

Figure 3: x86 Assembly instructions compared with PCode and C Code

The core concepts of PCode are address space, varnode, and PCode operation. The address space for PCode is a generalization of RAM. It is an indexed sequence of bytes that can be read and written by the PCode operations. Varnodes are generalizations of either registers or memory locations and are contiguous chunks of bytes that have no type, although PCode operations can force three different type interpretations: integer, boolean or floating-point. A PCode operation is like a machine instruction. All PCode operations have a common structure which is one or more varnodes as input and ideally a single output varnode, although some operations such as the BRANCH operation does not support output. The action of the operation is determined by its opcode.

PCode abstracts the complexities of different architectures, allowing for a common instruction set for reverse engineers to work with. However, PCode is not completely independent of the original target architecture. The original target architecture's instruction side effects are carried over into the PCode representation which complicates the matching. For example, arithmetic instructions in many architectures set architecture-dependent flags which record things like register overflow, equality with zero, and if the results are positive or negative.

2.4 Function Matching Tools

There are a multitude of different tools for function matching. One of the most common tools is BinDiff, which uses multiple techniques to determine the similarity of two files [6]. These techniques include different algorithms such as hash matching, name hash matching, MD index

matching, prime signature matching, call sequence matching, string references, and address sequence among other techniques [6]. Similarity is reported as a number describing how similar the two functions are. BinDiff is primarily a plugin for the binary analysis tool IDAPro but can also be used with other reverse engineering tools, such as Ghidra [6].

Ghidra also comes with its own function matching tool, known as the Ghidra Version Tracking Tool [11]. The tool is designed to identify differences between an old and new version of the same binary. The tool leverages multiple matching algorithms including data matching, symbol name matching, function byte matching, function mnemonics matching, and function instructions matching. Like BinDiff, the Version Tracking Tool also gives a similarity score for each match.

Chapter 3: System Design

The PCodeMatcher proposes using intermediate representation to solve the problem of binary diffing across architectures. Our hypothesis is that using intermediate representation allows for cross-architecture matching between binaries, decreasing the manual work required of reverse engineers leading to faster analysis of binaries. To match functions from each binary, we lifted the functions to PCode. This allows for functions to be matched based on their sequence of PCode operations. Using an intermediate representation language such as PCode allows for functions to be matched regardless of the target architecture.

Figure 4 shows a system design that compares two binaries. The system extracts functions from each binary and lifts them to PCode. The PCode representations of each function are then ingested by the matcher and we are then provided with matches between the two binaries.

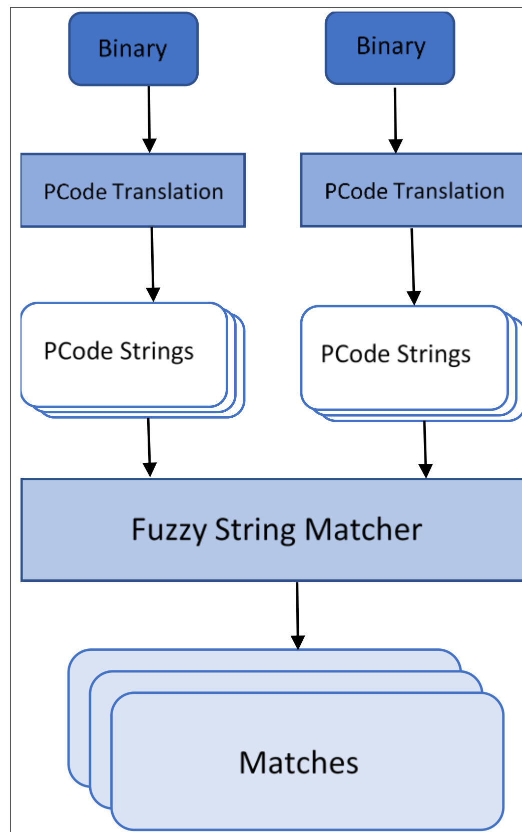

Figure 4: Flow chart for the system

To match functions, fuzzy string matching (also called approximate string matching) was used. We discuss the details of the fuzzy string matcher in Section 3.2. This allowed for matches to be found despite some differences between the PCode strings that would complicate exact matching.

3.1 PCode Translation

We can acquire the functions within a given binary through Ghidra's ability to iterate through all the functions. As the functions are extracted from each binary, they are ingested by the PCode translator which translates the processor instructions into a sequence of PCode instructions. PCode generation is handled by the decompiler module, which we invoke first. Once the function has been passed in, the first step is to decompile the function to get the high-level function structure associated with the decompilation results. If there has been an error with decompilation, such as a timeout, the high-level function structure will return null and we cannot proceed with PCode translation for that function. In that case, that function is then considered to not match anything according to our matcher, and it is not passed on to the fuzzy string matcher to be compared with the functions in the other binary. After decompilation, the high-level function structure is then broken down into its basic blocks which can then be iterated through to get the PCode instructions.

It is at this stage, the PCodeMatcher performs normalization. The first step taken in the normalization process is to filter out two specific PCode opcodes: INDIRECT and MULTIEQUAL. They are used for generating Single Static Assignment form [12]. The MULTIEQUAL operation represents a copy from one or more possible branches while the INDIRECT operation is a placeholder for possible indirect effects [12]. Both opcodes are generated as side effects from the real control flow instructions. Since they are not mappable to any CPU instruction, they were not included in the PCode strings used for comparison by the fuzzy string matcher.

Once those instruction are filtered out, the rest of the PCode instructions are formatted into a string to be passed on to the fuzzy string matcher. The format of each instruction is broken up into the inputs, the PCode opcode and the output. An example of a formatted PCode instruction is shown below in Figure 5.

```
register INT_EQUAL register const
```

Figure 5: PCode Instruction

In Figure 5, we can see an example of the integer equality operator INT_EQUAL. To the right of the opcode are the input varnodes. To the left of the opcode is the output varnode. The purpose of this instruction is to compare the integer stored in a register to a constant integer. The output is stored in a register.

3.2 Fuzzy String Matcher

A fuzzy string matcher, FuzzyWuzzy [16], is used to calculate a similarity score out of one hundred between the PCode representations of pairs of the functions within the two binaries. This allows for deviations between the PCode strings without disqualification. Where most string matchers require two strings to be the same without any disparity, the fuzzy matcher will look to find similarity rather than exact equality. The matcher will look at two strings with a few discrepancies and see that, while they may not match exactly, they could be close enough to each

other to be considered a match while an exact string matcher would determine that the two strings would not be a match based on their differences.

The matcher we used calculates the Levenshtein similarity ratio based on the Levenshtein distance as a measure of similarity between two strings. The Levenshtein distance is defined as the minimum number of single character changes (i.e. substitutions, insertions, or deletions) required to change one sequence into another. The Levenshtein similarity ratio is calculated by subtracting the Levenshtein distance for two sequences from the sum of the lengths of those sequences and then dividing by the sum of the lengths of the two sequences being compared. Figure 6 shows normalized PCode instructions from two functions. Figure 7 shows the formula for the Levenshtein similarity ratio. Using the formula, the calculated similarity score for the strings in Figure 6 would be sixty-three.

String A:	**String B:**
register COPY const unique INT_ADD register const unique INT_ADD register const stack COPY const stack COPY const stack COPY const stack COPY const	stack COPY const unique LOAD const ram unique INT_ADD ram const unique LOAD const unique register INT_ADD register const stack COPY const register INT_ADD register const

Figure 6: Normalized PCode strings for comparison

$$\frac{(|a| + |b|) - lev_{a,b}(i,j)}{|a| + |b|}$$

Figure 7: Levenshtein similarity ratio formula

Since the function strings can be large and are not likely to match one hundred percent based on the discrepancies with side effects across architectures and the addresses within each binary, it becomes necessary to establish a threshold for matching. This will set a boundary where any similarity score above a certain number will be considered a match and anything below will not. When we allow for partial matching, there will be more of an opportunity for a true match to be found by the reverse engineer and lessen the possibility of false negatives.

3.3 Similarity Threshold

Similarity is defined as how much two functions resembled each other. In the case of our PCodeMatcher, similarity is calculated with the Levenshtein similarity ratio and represented as a score between zero and one hundred. This is on par with BinDiff as well as Ghidra's Version Tracker, both of which report similarity as a score between zero and one hundred. Because of the slight variations within the PCode due to the lift from architecture specific instructions, getting a score of one hundred from the fuzzy string matcher for two functions lifted from different

architectures is unlikely. Still, we cannot dismiss a potential match for not getting a perfect score. Therefore, a similarity threshold is needed to set a minimum similarity that two functions need to achieve to be considered a match. The results section below will discuss the derivation of an ideal threshold value.

Chapter 4: Methodology

In this chapter, we go over the binaries used to test the PCodeMatcher as well as the industry tools we compared our results to. We also describe what we use as ground truth to compare our results against.

4.1 Test Binaries

To test the PCodeMatcher's capabilities, we gathered a variety of different binaries to compare. Ghidra supports a wide variety of architectures, but we decided to focus on several of the most common architectures. We manually cross-compiled binaries for ARM, MIPS, x86 and PowerPC. To start, we compiled small self-written C programs that included if-else statements, loops, case statements and other simple operations to understand the way that Ghidra handled lifting the assembly code from various architectures into PCode in these situations. This also allowed us control over what we saw with the PCode as well as how small changes in the programming might affect the matching results.

Size (KB)	Number of Functions	Filename	Source	Architecture
9-73	18-22	Self-written programs	Self-compiled	ARM, MIPS, x86, PowerPC
990-1,427	3180-3575	BusyBox 1.32.0	BusyBox	ARM, MIPS, x86, PowerPC
431-626	163-305	zlib 1.2.11	zlib	ARM, MIPS, x86, PowerPC
364-467	321-558	libmicrohttpd-0.9.70	GNU	ARM, MIPS, x86, PowerPC
1,296-1,611	879-1439	gnuchess-6.27	GNU	ARM, MIPS, x86, PowerPC
3,296-4,614	2782-5049	bash-5.0	GNU	ARM, MIPS, x86, PowerPC
758-1,061	665-1068	grep-3.5	GNU	ARM, MIPS, x86, PowerPC
723-1,114	809-1789	readline	GNU	ARM, MIPS, x86, PowerPC

Table 1: Summary of test binaries

Table 1 displays all the test binaries that we used for this project. There were some criteria that the test data needed to meet to be included for testing. We needed existing binaries that were much larger than the self-written programs and we needed the functions in those binaries to be of varied sizes and operations. This was to ensure that we tested the PCodeMatcher with binaries that displayed a variety of different behaviors that would translate to a variety of different PCode opcode sequences.

We used these test binaries to test and evaluate the PCodeMatcher as well as BinDiff and Ghidra's Version Tracker to compare the results of the PCodeMatcher against common industry tools. Each set of source code was compiled into each architecture and comparisons were made between each pair of architectures (always comparing within the same source code). For example, BusyBox compiled for ARM was only compared against BusyBox compiled for the other three architectures, readline compiled for one architecture was only compared against the readline compiled for the other three architectures and so on. This resulted in six different architecture to architecture comparisons for each binary. This process was completed for the PCodeMatcher, BinDiff and Ghidra's Version Tracker. The results of these comparisons can be found in the Results and Discussion chapter.

4.2 Ground Truth

To determine the accuracy of our PCodeMatcher, we needed to establish a ground truth to compare our results to. In binary matching, there could be multiple definitions for a true match between two binaries. For example, a score of one hundred for a binary comparison between two functions could be considered a true match. Reverse engineers trying to accomplish different purposes will have different definitions of a true match. For this project, we define a true match to be two functions that have the same name as recorded by the symbols compiled into the binary. We do not take possible duplication of functions into account.

Chapter 5: Results and Discussion

In this section, we provide and discuss our results. We provide our determination for the similarity threshold as well as a breakdown of results by function size for the PCodeMatcher, BinDiff, and Ghidra's Version Tracker. We also provide a comparison of results from all three matchers.

5.1 Data Plots for PCodeMatcher

The performance of the PCodeMatcher is displayed in the Receiver Operator Characteristic (ROC) curve shown in Figure 8 below.

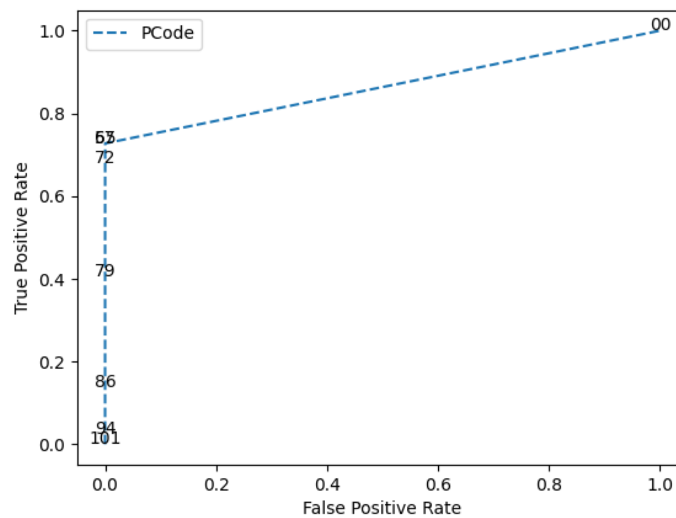

Figure 8: PCodeMatcher results displayed in a ROC curve

In a ROC curve, a true positive rate equal to the false positive rate will give a straight line along the diagonal. Figure 8 demonstrates that our ROC curve veers towards the upper left-hand corner and then heads into the right-hand corner. Note that data was not collected for matches of less than 70 out of 100, causing the straight line to the top-right. ROC curves are more appropriate for datasets that are balanced among each class. Our data set is very imbalanced because for every function in one binary, there is only one true match in the other binary being compared and every other function in that binary that is not the true match is classified as a false match. This means that our data is dominated by true negatives and Precision-Recall (PR) curves are more appropriate for interpreting our data. Figure 9 below displays the results from the PCodeMatcher in a PR curve.

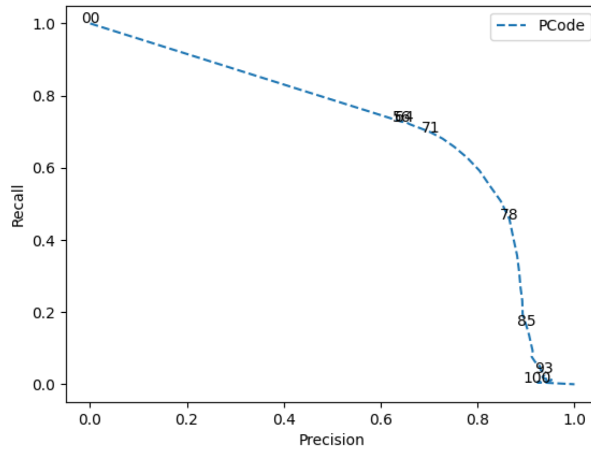

Figure 9: PCodeMatcher results displayed in a PR curve

In Figure 9 we can see that the PR curve generated by the PCodeMatcher results is not obscured by the high number of true negative results. From Figure 9, we can see that the optimal threshold for our matcher balancing both precision and recall would be in the range of seventy – eighty as the values within this range span the top right corner of the PR curve.

5.2 Similarity Threshold

To determine the optimal minimum threshold for the PCodeMatcher, we tested the implementations with three different similarity thresholds based on our observation of the PR curve. We needed a threshold that was high enough as to not lead to false positives and low enough where true matches would not be eliminated for not meeting the threshold. The minimum similarities we tested were seventy, seventy-five and eighty. Figure 10 displays the results of each threshold when comparing all binaries in confusion matrices.

Threshold of 70:

		Actual Values	
		Match	No Match
Predicted Values	Match	15238	6456
	No	6569	39145753

Threshold of 75

		Actual Values	
		Match	No Match
Predicted Values	Match	12038	2566
	No	9769	39149643

Threshold of 80

		Actual Values	
		Match	No Match
Predicted Values	Match	6766	871
	No	15041	39151338

Figure 10: Similarity threshold results displayed in confusion matrices

After examination of the results from all three similarity thresholds, we conclude that a minimum threshold of seventy is the best of these values to optimize for true positive rate. First, we can see that the similarity threshold of eighty should not be used due to its true positive rate of 31%. From analysis of the confusion matrices of the remaining similarity thresholds, we can see that the minimum threshold of seventy has the highest true positive rate of 70% compared with a true positive rate of 55%. The minimum threshold of seventy does have a higher false positive rate of .02% compared with a false positive rate of .006% for a threshold of seventy-five. Higher thresholds increase the precision, meaning more of the matches they find are indeed true matches, but this comes at the cost of a lower recall rate as shown.

5.3 PCodeMatcher

Figure 11 shows the results of the PCodeMatcher for all binary comparisons. A threshold of seventy was used. From these results, we can see that at this threshold the PCodeMatcher has a sensitivity of 70%, also called the true positive rate, meaning that it is able to correctly match 70% of truly matching functions compared. The PCodeMatcher also has a specificity of 99% which is also called the true negative rate and means that it can correctly identify true negatives 99% of the time.

		Actual Values	
		Match	No Match
Predicted Values	Match	15238	6456
	No Match	6569	39145753

Figure 11: PCodeMatcher results displayed in a confusion matrix

Figure 12 displays results broken down by function size for the PCodeMatcher. A similarity threshold of seventy was used to generate these results. The sizes we used were eighty to five hundred bytes, five hundred to one thousand bytes and over one thousand bytes. We ignored functions that were below eighty bytes because these functions only contain about twenty instructions or less. When taking a binary that is already analyzed and comparing that to another unknown binary, attempting to match functions under 80 bytes would not give us much insight about the unknown binary.

80-500 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	10658	5490
	No	4880	22300241

500-1000 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	1623	246
	No	448	694708

>1000 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	1559	209
	No	642	489865

Figure 12: Results from different function sizes displayed in confusion matrices

After evaluation of the results for each size range, we can say that the PCodeMatcher performed slightly better for functions that are between five hundred and one thousand bytes in size for our sample binaries. The PCodeMatcher was able to match 78% of functions correctly within that size range. The PCodeMatcher matched 70% of functions greater than one thousand bytes correctly and 68% of functions between eighty and five hundred bytes correctly. The PCodeMatcher uses functions lifted to PCode, essentially a pattern of PCode instructions to match functions between binaries. Functions that are between five hundred and one thousand bytes could be more easily matched by the PCodeMatcher since the pattern of PCode instructions produced from these functions can be distinguished from other functions by the fuzzy string matcher. Functions that are between eighty and five hundred bytes are smaller and therefore the pattern of PCode instructions generated from these functions will have enough similarity to throw off the fuzzy string matcher while functions that are over one thousand bytes will have more differences that will also throw off the fuzzy string matcher. Future study in this area would be necessary to definitively confirm this speculation.

5.4 Comparing against Industry Tools

Comparing the results of the PCodeMatcher to industry tools is also important. Testing against widely accepted and used industry tools will help determine the real-world usefulness of our matcher. The industry tools we compared the PCodeMatcher against are BinDiff and Ghidra's Version Tracker [6, 11]. Figure 13 shows the comparisons of the PR curves generated from the results from all three matching tools. Here we can see that the PCodeMatcher has higher recall than BinDiff and the Version Tracker for many thresholds and has higher precision than Version Tracker for many thresholds.

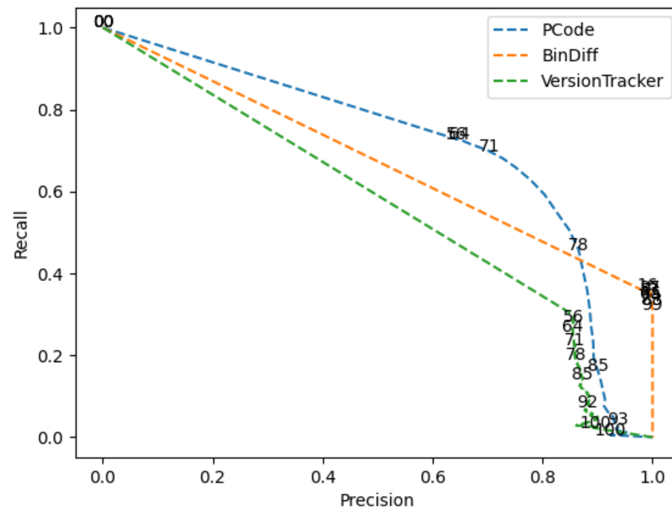

Figure 13: Results from all three matchers displayed in a PR curve

5.4.1 BinDiff

The results from BinDiff are shown in the confusion matrix in Figure 14. The data presented here was obtained with BinDiff 6 using the default configuration and a match threshold of 70.

		Actual Values	
		Match	No Match
Predicted Values	Match	7309	12
	No Match	14498	39152197

Figure 14: BinDiff results displayed in a confusion matrix

From the results shown in Figure 14, we can see that the BinDiff has a true positive rate of 33%. BinDiff also has a true negative rate of 99%. BinDiff had very few false positives, meaning that of the functions it matched, it did so accurately for the most part. However, BinDiff had many false negatives, managing to only match around 33% of functions that were truly matches. In comparison, the PCodeMatcher performed much better at 70% of functions matched accurately. Figure 15 shows the results from BinDiff broken down by function size.

80-500 Bytes		Actual Values	
		Match	No Match
Predicted Values	Match	5234	8
	No	10304	22305723

500-1000 Bytes		Actual Values	
		Match	No Match
Predicted Values	Match	767	0
	No	1304	694954

>1000 Bytes		Actual Values	
		Match	No Match
Predicted Values	Match	747	0
	No	1454	490074

Figure 15: Results from different function sizes displayed in confusion matrices

After evaluation of the results, BinDiff performed slightly better for functions that are between five hundred and one thousand bytes in size within our sample binaries, similar to the PCodeMatcher. BinDiff was able to match 37% of functions correctly within that size range. BinDiff matched 33% of functions greater than one thousand bytes correctly and 33% of functions between eighty and five hundred bytes correctly.

5.4.2 Ghidra's Version Tracker

Ghidra's Version Tracker has multiple different matching algorithms it can use to find matches within binaries. The Version Tracker assigns a score to each match within a range of 0.0 to 1.0. First, we ran the Exact Data Match algorithm and accepted all the matches above the threshold 0.7. Next, we ran the rest of the matching algorithms excluding those that compared symbols because we wanted to see results of direct examination of the functions. Since there were multiple matching algorithms, there were duplicate matches with different similarity scores because they were found by different matching algorithms. In these cases, we kept the match with the highest similarity score and ignored the others. The results for matches above a similarity of 0.7 are shown in Figure 16 below.

		Actual Values	
		Match	No Match
Predicted Values	Match	5019	836
	No Match	16788	39151373

Figure 16: Version Tracker results displayed in a confusion matrix

From these results shown in Figure 16, we can see that the Ghidra's Version Tracker has a true positive rate of 23%. The Version Tracker also has a true negative rate of 99%. The Version Tracker had more false positives than BinDiff, meaning that it was less accurate at matching functions correctly. The Version Tracker also had many false negatives, managing to only match around 23% of functions. In comparison, the PCodeMatcher performed much better at 70% of functions matched accurately. Figure 17 shows the results from the Version Tracker broken down by function size.

80-500 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	3277	498
	No	12261	22305233

500-1000 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	606	55
	No	1465	694899

>1000 Bytes

		Actual Values	
		Match	No Match
Predicted Values	Match	571	99
	No	1630	489975

Figure 17: Results from different functions sizes displayed in confusion matrices

After evaluation of the results, as was the trend with other matchers, the Version Tracker performed slightly better for functions that are between five hundred and one thousand bytes in size within our test binaries. The Version Tracker was able to match 29% of functions correctly within that size range. It matched 25% of functions greater than one thousand bytes correctly and 21% of functions between eighty and five hundred bytes correctly.

Chapter 6: Conclusions

The scope of the project was to attempt to solve the problem of cross-architecture comparisons of binaries using PCode, an architecture agnostic representation of assembly language. When compared to industry tools such as BinDiff and Ghidra's own matching tool, Version Tracker, the PCodeMatcher showed significantly higher recall while maintaining similar precision to Version Tracker for many thresholds.

With the given test sets, the fuzzy matcher performed well, matching most of the largest functions in each binary comparison. It did not perform as well with many of smaller functions within each binary where there was much more similarity between functions, meaning there is room for further improvement in our system.

6.1 Future Work

We have identified several areas for improvement within the project. One improvement that could be made is the implementation of other matching techniques using PCode. This project used fuzzy string matching which had some complications due to the requirement of string normalization before comparison and the arbitrary encoding lengths introduced by opcode and varnode labels. This led us to believe that in the future, other matchers could be implemented to avoid these problems. One example of another matcher is to use Abstract Syntax Trees (AST). Ghidra already has the capability to build ASTs using PCode which would allow for graph matching. Graph matching would allow for more accurate matching since it would eliminate the complications of string matching. Further normalization, such as removing variations in the PCode due to the lift from architecture specific instructions, could also improve results from the fuzzy string matcher.

Another area of improvement for this project is the time optimization of the PCodeMatcher. Currently the PCodeMatcher runs comparisons of every function within Binary A against every function within Binary B to find potential matches, taking up hours of compute time to run two large binaries with upwards of a thousand functions each. There was a line of inquiry into parallelization, but it was determined that this was not within the scope of the current project and would be left for further research. Another area for optimization about time is using preprocessing techniques before running the functions through the matcher. If there was a way to narrow down the list of functions into groups that are only partially similar to each other in either size or behavior, the matcher would not have such a large list of functions to match to each other and therefore finding matches for each function would be less time consuming.

6.2 Lessons Learned

Throughout this project, we learned many lessons. This project was somewhat intimidating at the beginning, knowing almost nothing about reverse engineering or binary analysis. But through research and problem solving, we learned a lot about reverse engineering. Talking to

experts was also a great resource that we utilized many times throughout the completion of this project. Reaching out for help instead of spending significant amounts of time on problems was also a good lesson to learn. Many times, other people will have the answers you need, and it is not necessary to struggle through problems on your own.

The completion of this project led to much more knowledge in computer science topics such as binary diffing and reverse engineering techniques but it also led to project management skills and learning to set and manage expectations from sponsors and advisors.

References

[1] C. Karamitas, & A. Kehagias. (2018). Efficient features for function matching between binary executables. Paper presented at the *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER),* 335-345. doi:10.1109/SANER.2018.8330221

[2] Lageman, N., Kilmer, E. D., Walls, R. J., & McDaniel, P. D. (2017). BinDNN: Resilient function matching using deep learning. Paper presented at the *Security and Privacy in Communication Networks,* 517-537.

[3] listDiff. (2019). What is RISC and CISC architecture and their differences? Retrieved from https://www.listdifferences.com/risc-and-cisc-architecture-their-differences/

[4] Rodriguez, A. (2015). The importance of language, binary diffing and other "one day" stories. Retrieved from https://www.incibe-cert.es/en/blog/importance-of-language-binary-diffing-and-other-stories-of-1day

[5] The Editors of Encyclopaedia Britannica. (2015). Compiler. Retrieved from https://www.britannica.com/technology/compiler

[6] zynamics, "Zynamics BinDiff," zynamics.com, [Online]. Available: https://www.zynamics.com/bindiff.html.

[7] S. G. Shiva, "Advanced Computer Architectures," Taylor & Francis Group, Boca Raton, FL, 2006.

[8] steeve, "Dockcross," GitHub. Retrieved October 16, 2020, from https://github.com/dockcross/dockcross

[9] Toal, R. (n.d.). Intermediate Representations. Retrieved October 06, 2020, from https://cs.lmu.edu/~ray/notes/ir/

[10] Buffer Overflow. (n.d.). Retrieved October 08, 2020, from https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[11] National Security Agency. (n.d.). Ghidra. Retrieved October 08, 2020, from https://ghidra-sre.org/

[12] PCode Reference Manual. (n.d.). Retrieved October 09, 2020, from https://ghidra.re/courses/languages/html/pcoderef.html

[13] T. Bradley, "Report: Average of 82,000 new malware threats per day in 2013," PCWorld.com Retrieved October 16, 2020, from https://www.pcworld.com/article/2109210/report-average-of-82-000-new-malware-threats-per-day-in-2013.html

[14] Karim, Md & Walenstein, Andrew & Lakhotia, Arun & Parida, Laxmi. (2005). Malware phylogeny generation using permutations of code. Journal in Computer Virology. 1. 13-23. 10.1007/s11416-005-0002-9.

[15] DuPaul, Neil, 2020. *Static Testing Vs. Dynamic Testing | Veracode*. Veracode. Available at: https://www.veracode.com/blog/secure-development/static-testing-vs-dynamic-testing Accessed 16 October 2020

[16] "Xdrop/Fuzzywuzzy." *GitHub*, 14 Oct. 2020, github.com/xdrop/fuzzywuzzy. Accessed 16 Oct. 2020.