

Symbolic Execution for 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 12, 2017

By: Samantha Comeau

WPI Advisor: Professor Robert Walls

Sponsor: The MITRE Corporation

**Approved for Public Release;
Distribution Unlimited. Case Number 17-3975
©2018 The MITRE Corporation. ALL RIGHTS RESERVED.**

Abstract

The idea for this project was to use symbolic execution to create an architecture-agnostic representation of a function to use for function matching. We were motivated because reverse engineering is a difficult job to do because of the many limitations of static analysis. Simply looking at assembly code and attempting to determine what a function does based on different inputs is difficult and time consuming. Using symbolic execution, a reverse engineer no longer must read and interpret assembly language and can rather focus on the core behavior of the function by interpreting the symbolic analysis results. Symbolic execution is a dynamic analysis method that provides the reverse engineer with a better understanding of what a binary does during run-time. Using symbolic execution for a function matcher allows for functions to be matched based on how they react to symbolic variables such as parameters, memory reads and more.

Our matcher uses symbolic constraints to match functions cross architecture. This project uses *angr* to extract symbolic information about the program and generate an architecture agnostic representation of a function. We then use this representation to match functions from different binaries. We evaluated our work using a variety of test binaries compiled for different architectures and the same binary from multiple versions of a project. Our matcher gave us upwards of 87% of functions matched when it came to ideal functions for this matcher type. Ideal functions for this matcher are functions whose control flow relies on run time data. Where we could match functions cross architecture, the leading tool for function matching, BinDiff, could not. Our proposed matcher attempts to solve the problem of cross architecture comparisons of binaries while also allowing full code coverage.

Acknowledgements

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

The MITRE Corporation for the sponsorship of the project

Peter Lucia

Peter Brown, Eric Cheng, Carlos Cheung, Karen Lamb, Hiroshi Fuji, Corre Steele

Professor Robert Walls

Professor Alexander Wyglinski

Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
Table of Contents.....	iv
List of Figures	vi
1. Introduction	1
2. Background	4
2.1 Computer Architectures and Compilers	4
2.2 Function Matching	7
2.3 Binary Analysis Techniques.....	8
2.4 <i>angr</i> 's Symbolic Execution Engine	9
2.5 Related Work	13
3. System Design	16
3.1 Overview	16
3.2 Constraint Extraction	17
3.3 Matcher Overview	21
3.4 Symbolic Constraint Value Matcher	23
3.5 Symbolic Constraint Fuzzy Value Matcher.....	25
3.6 Symbolic Constraint String Matcher	26
4. Methodology.....	27
4.1 Datasets	27
4.2 Ground Truth and Minimum Similarity.....	28
4.3 Environment.....	29
5. Results and Discussion	30
5.1 Results.....	30
5.2 Evaluation	32
5.3 Comparing Symbolic Matchers	36
5.4 <i>angr</i> 's Symbolic Execution Engine	36
6. Conclusions	38
6.1 Matchers	38
6.2 Recommendations for Future Study and Improvement Measurement.....	38

6.3 Lessons Learned	39
References	40
Appendix A	43
Appendix B	44
Appendix C	45
Appendix D	47
Appendix E	49
Appendix F	50
Appendix G	51
Appendix H	52
Appendix I	53
Appendix J	54

List of Figures

Figure 1. Source code used to generate the instructions shown in Figure 2.....	5
Figure 2. Source code in Figure 1 compiled for x86 on the left and ARM on the right.	5
Figure 3. Compiler flow diagram that demonstrates the transition from source code to assembly code through the compiler. Based on GCC’s functionality.....	6
Figure 5. This is a sample program. The parameter ‘a’ will be made symbolic at the beginning of the function analysis and then later ‘final_value’ will also be symbolic because of its dependency on a symbolic variable.	9
Figure 6. The data dependency tree for ‘final_value’.....	10
Figure 7. Initializing a project in angr.....	10
Figure 8. A CFG demonstrating relationships between basic blocks.....	12
Figure 9. Constraints in angr are stored as abstract syntax trees.....	12
Figure 10. This shows a sample program state for the values stored in each register and whether the value is currently symbolic. This is an x86 program.	13
Figure 11. Overall flow chart of our proposed matching and extraction techniques.....	16
Figure 12. Constraint extraction design for our system.	17
Figure 13. Demonstrates how the extractor stores the results in the graph database. After symbolic execution, each function has different possible paths of execution with their constraints stored on the path.....	19
Figure 14. A demonstration of 3 different possible paths and the constraints that live on those paths. This is from a binary compiled for x86.....	20
Figure 15. Example of an error that can occur in angr when the constraint solving does not concretize properly.....	21
Figure 16. A demonstration of the output from extraction. Depending on the function there can be more or fewer paths and depending on the path there could be more or fewer constraints.....	22
Figure 17. Flow graph for all matchers. The only place where matchers differ is within the Constraint Matching methodology.....	23
Figure 18. An example of a constraint matrix.....	24
Figure 19. On the left is x86 and on the right is the same source code compiled for arm. As shown, arm and x86 order their variable and constants differently. This is the final possible path’s constraints.	25
Figure 20. An example of the same constraint matrix as in Figure 18, however this one uses fuzzy matching.	26
Figure 21. Summary of selected test data. A variety of architectures and code needed to be tested so self-written binaries were used to test cases like looping, recursion and more and the large binaries were used to test the usability for production.	27
Figure 22. This figure shows the composition of a binary and the categorization of the functions by percentage of the binary.	30
Figure 23. The percentage of functions of each type with constraints. Generated using our test data....	31
Figure 24. This shows the number of times angr threw errors on constraints during extraction for some of our test binaries.....	32
Figure 25. The confusion matrix for the pairwise function comparisons for functions with constraints only.....	33

Figure 26. A subset of some of the test data to show how the distribution of functions that were matched with our class of matchers in comparison to how many functions there were in total..... 33

Figure 27. The confusion matrix for all possible function comparisons. 34

Figure 28. BinDiff in comparison to our class of matchers for cross architecture comparisons on functions with a change in code flow due to run time data: the ideal functions..... 35

Figure 29. Summary of some of the places where the fuzzy matcher performed better than the value matcher. 36

Figure 30. Unsupported operation error in angr. 37

Figure 31. Here the Control Flow Graph shows all debugging information. From top to bottom it provides the basic block's address and name of the container function and offset, then it moves to the opcodes for the architecture, then the intermediate representation statement block, then the information on the return target and more. 43

Figure 32. This is a sample matrix that would be used for matrix scoring using the Hungarian algorithm. 45

1. Introduction

For cyber security, analyzing binaries is an important component to malware analysis, software analysis and many other types of analysis. Reverse engineers spend a lot of time reading assembly language and attempting to infer what a binary does, whether it be for malware analysis or something as simple as bug detection. To save time, developers have written function matchers such as BinDiff [1]. Function matching is the concept of taking two functions from different binaries and attempting to determine how similar the functions are with confidence. Some reasons that two separate binaries would share the same functions include: upgrading versions, shared imported library functions, and more. Ultimately, this saves the reverse engineer time because they can first run automated analysis before needing to manually analyze functions to determine which are harmful.

This problem is known as binary code similarity detection [2]. Using function matchers to solve this problem can save a reverse engineer time because they then avoid needing to re-analyze the same function twice [3] and they can also use this to find functions that did not match and then manually analyze the interesting new functions. Another advantage to knowing what functions from different binaries match is potentially knowing who the code was written by if they have the same coding style or reuse the same exact functions. This aids malware reverse engineers in finding similar binaries and using social engineering to analyze the malware author. Automated function matching has the potential to save the human reverse engineer days or weeks of ground work.

There are many challenges in developing robust function matchers. Simply attempting to compare the instructions from one function to the instructions from another is typically inaccurate. This comes from differences in architecture instruction sets, choice of compiler and compiler options, such as optimization levels. This creates a growing need for functions to be matched using different techniques so they can cover all cases and classes of functions and binaries. With the expansion of the Internet of Things, reverse engineers have the need to analyze many different architectures including, x86, ARM and MIPS. These architectures are incredibly common for IOT processors [4]. Many new types of function matchers have been released to solve some of these function matching problems as the computer science community evolves to explore new areas. These new areas include deep learning [3], graph matching and more.

This project aims to create a function matcher using symbolic execution. This approach is attractive for a few key reasons. This matcher attempts to solve the problem of cross architecture comparison and allows full code coverage. Symbolic execution allows us to generate a model of a function's code flow during execution. This model is consistent across different architectures regardless of the underlying instructions. Therefore, even if the

instructions are completely different the basic logic of the program is the same. Allowing cross architecture matching allows for reverse engineers to get more out of their matchers. For reverse engineers to stay current with new firmware's, programs and other developments, the need for cross architecture matching techniques grows.

Contributions of MQP

For the Major Qualifying Project, we make three main contributions.

Generate an architecture-agnostic representation of a function using constraints derived from symbolic execution. To generate an architecture-agnostic representation of a function we had to first determine all the different analysis possibilities that are presented when using symbolic execution, which led us to use path constraints. We need to first run symbolic execution on a binary for every individual function until we reach the end of the function. We then extract the constraints that were placed on every possible path through the function and use this information to generate a representation of the function.

Develop a novel algorithm to match functions based on the symbolic representation. Our contribution of the function matcher was accomplished through developing an algorithm that can be used to match our symbolic representations of functions. This involves first generating the symbolic representation of each of the functions contained in two binaries. We can then pairwise match the functions from the first binary to the functions from the second using our algorithm. This involves looping through the paths and finding which paths are the same as each other based on what constraints they have. Ultimately, we have automated this process to the point where, the reverse engineer just needs to run the extraction script and then the matcher script on the binaries they would like to match and then analyze the matcher results.

Evaluate the effectiveness and usability of the function matcher. Our final contribution included testing the usability of symbolic execution for function matching and the effectiveness of our implemented algorithm. We first found and generated test data to represent a wide variety of different situations that would affect our matcher including looping situations, recursive functions and a variety of binary sizes. These were then used for accuracy and efficiency tests on both our extractor and class of matchers.

Novelty

To the best of our knowledge, we are the first to propose a symbolic function matcher. However, there is one similar function matcher that matches Abstract Syntax Trees (AST). There are similarities between our proposed matching technique and this AST matcher. Our proposed matcher relies on symbolic constraints which are stored as Abstract Syntax Trees and the AST matcher also matches ASTs. The difference however is in the algorithm we developed

to match these trees. We match based on values whereas the AST matcher uses hashing and graph matching techniques. The methods they use could complement our class of matchers.

Results

The results of this symbolic execution function matcher show a strong reliance on the type of function that is being matched. It's important to consider the different types of functions because these symbolic matchers can only do well on certain types of functions. The ideal function type for this function matcher is a function that relies on run time data to affect code flow. Overall, for these ideal functions with constraints, the fuzzy matcher matched 87% of the functions and the value matcher matched 82% of the functions.

2. Background

2.1 Computer Architectures and Compilers

The specific code contained in a binary executable depends on myriad factors, including choice of compiler, optimization level, and target architecture. In other words, the same source code will produce very different end binaries based on these factors. These *binaries* are the machine-readable code that comes from compiling source code written in languages like C. Compilers are important because they generate the machine-readable code from source code, and based on the specified options, this will generate different binaries for the same source code. These concepts are imperative to understand for reverse engineers wishing to accurately analyze code.

Considering the intricacies of binary creation is important because it affects function matching techniques. In general, each function matching technique has pre-conditions that must be satisfied. For example, for many matching techniques, functions must be compiled for the same architecture to be matched. These pre-conditions affect the accuracy and usability of many existing function matching techniques. The proposed class of matching techniques using symbolic execution attempt to match regardless of many common pre-conditions being unsatisfied achieving this requires previous in depth knowledge of the differences between architectures and the methodology behind compilers. To leverage the analysis provided by a symbolic execution engine, understanding of how this analysis was affected by architectures and other differences is integral to utilizing the results.

Architectures

An *architecture* is a set of the rules and methods that are used to support an operating system regarding structure, organization, performance and the implementation of computer systems [5]. Instruction sets between architectures are completely different. There are multiple types of architectures but this project focuses on the control-driven architectures, RISC, reduced instruction set computer, and CISC, complex instruction set computer [6].

Within the categories of RISC and CISC lie the architecture implementations. Some examples of common architectures include ARM, x86, MIPS and more. ARM is a RISC architecture and x86 is a CISC architecture. Comparing these two architectures allows the differences in instruction set and speed to be easily demonstrated.

```

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

```

Figure 1. Source code used to generate the instructions shown in Figure 2.

```

push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x4(%rbp)
cmpl   $0x0,-0x4(%rbp)
jne     400501 <func+0x14>
mov     $0x0,%eax
jmp     400513 <func+0x26>
cmpl   $0x0,-0x4(%rbp)
jle     40050e <func+0x21>
mov     $0x1,%eax
jmp     400513 <func+0x26>
mov     $0xffffffff,%eax
pop     %rbp
retq

```

```

push    {r7}
sub     sp, #12
add     r7, sp, #0
str     r0, [r7, #4]
ldr     r3, [r7, #4]
cmp     r3, #0
bne.n   83ce <func+0x12>
movs    r3, #0
b.n     83dc <func+0x20>
ldr     r3, [r7, #4]
cmp     r3, #0
ble.n   83d8 <func+0x1c>
movs    r3, #1
b.n     83dc <func+0x20>
mov.w   r3, #4294967295 ;
        0xffffffff
mov     r0, r3
adds   r7, #12
mov     sp, r7
ldr.w   r7, [sp], #4
bx     lr

```

Figure 2. Source code in Figure 1 compiled for x86 on the left and ARM on the right.

Depending on the instruction set of the architecture this causes the variables in the program to be stored differently. This difference between architectures creates the need for a generic method of representing a binary regardless of architectures. This concept is defined as an *Intermediate Representation*, which is discussed more in the next section.

Compilers

Compilers are programs that translate source code into machine code. In the context of reverse engineering, understanding compilers and how the source code is transformed is necessary to understand the final output, which is the binary or executable. Generally, the code is translated into machine code and there are variations on how this process is applied when it comes to different architectures and different compilers.

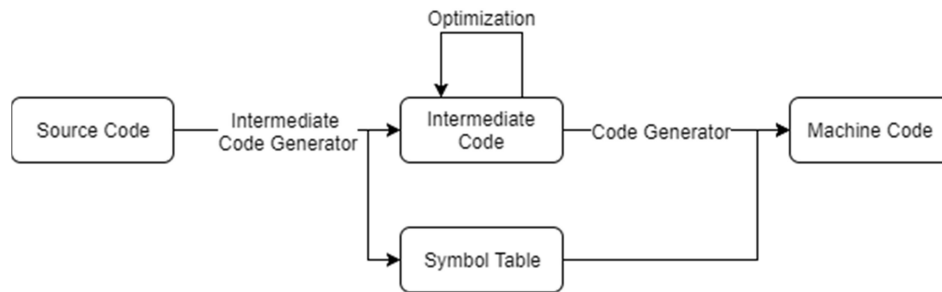


Figure 3. Compiler flow diagram that demonstrates the transition from source code to assembly code through the compiler. Based on GCC's functionality.

Compilers can be configured to also store additional information alongside the basics. One piece that is important for reverse engineering is the symbol table. Symbol tables are binding information about naming schemes in the binary [7]. The symbol tables store information such as the function names in a lookup table. As the compiler is working through the source code, it will add and modify this table continuously [7]. This information is useful to reverse engineers because having access to the function names aides in understanding what a function does. This information aides in static analysis techniques and can also be used to match functions based solely on their name. This information is only available when the compiler is directed to store this information.

Cross compilation is something to consider for this project. It is used to build test binaries and it is also important to understand the differences between cross compiling and regular compiling. Cross compiling is the action of running a compiler on one architecture, which compiles for a different architecture [8]. When regular compilation happens, the correct toolchain is often already installed on that operating system for the architecture it is running. However, when someone wishes to cross compile a program, the toolchain for the desired architecture is typically not installed on the current operating system. There are many different tools to implement cross compilation such as Dockcross [9], crosstools-ng [10], buildroot, and more which implement the proper toolchains for the user. In theory, a binary compiled for ARM

on a machine running ARM and the same binary compiled with the same compiler on a machine running x86 for ARM should be the same.

2.2 Function Matching

A match between functions can have many definitions. Our definition of a function match is a function that is attempting to do the same thing. This could be along the lines of comparing two strings and returning 1 if they are the same, and if the other function does the same thing, then these two functions are a match. There can be varying similarities in function matches such as saying a function has 90% similarity using BinDiff. This field of research is constantly growing through adding new techniques to determine what functions are similar, or fully the same, from different binaries. Some people define a function match as having a long Longest Common Subsequence of instructions or bytes, however others would determine this to be an inaccurate match. There are many methods for matching functions from different binaries. For function matching many researchers have set out to try to find the perfect method of determining if two functions from different binaries are the same.

There is a bounty of tools in the field of binary function matching. One tool, called BinSim, is a tool that aims to match functions based on execution traces. This method uses symbolic execution and dynamic slicing to compare instructions that impact observable behaviors [11]. This tool specifically works on malware samples and crypto ransomware to find matches between binaries.

Another tool, BinDNN, is a tool that uses deep learning and natural language processing to find matches regardless of compiler optimization levels [3]. In many cases, matching functions can rely on architecture or compiler, however this tool matches regardless of compiler optimization. The tool also uses machine learning due to machine learning's known success at building classifiers and its ability to accurately detect malware and network intrusion [3]. A combination these tools allows BinDNN to accurately match functions from different binaries with improved accuracy.

The most common and widely known function matching tool is BinDiff [1]. This tool uses an IDA Pro saved state in order to compare functions from two different binaries. BinDiff uses a variety of techniques to determine the similarity of two functions. The similarity is a number they assign to the two functions and the confidence is how confident in that similarity score they are. Some of the techniques BinDiff uses includes: CFG matching, hash matching, name hash matching, MD index matching, string references, address sequence and many more techniques. Therefore, BinDiff is reputable for rapidly producing true matches between binaries.

Our proposed function matching technique leverages symbolic execution. Using symbolic execution will allow for cross architecture comparisons of functions. This allows for the possibilities of matching with constraints, using system state such as the stack and registers at

different points in the program and more. These concepts are described more in the system design chapter.

2.3 Binary Analysis Techniques

Reverse engineering in general is the concept of taking something apart to determine how the object was originally configured [12]. In industry, this process is used to look at competitor's code and for academia, it is used to get hands on experience with decompilation [13]. For cybersecurity, reverse engineers take executable code, whether it be a binary for a Linux machine or an application for an Android device, and try to reverse engineer the code to determine what it does.

A binary file is a combination of many things. The main executable portion of the binary comes from the source code and includes all the functions from the binary. These functions are further broken down into basic blocks. A *basic block* is any section of code that does not branch. For example, if a function has an `if` statement, that will cause a branch in the code. During a conditional branch, at the end of the first basic block there is a jump from that address to the two other addresses, depending on which path the execution will take next. Binaries are analyzed at a function and basic block level, for different reasons and purposes. Most commonly they are analyzed function to function because this is a simple way to break up code into chunks however in cases where code flow is involved, consideration needs to be given to the interactions between basic blocks for a specific function. For this project, they are analyzed at the function level, but the interactions between basic blocks is important to consider during analysis.

There are many purposes behind reverse engineering machine code for cybersecurity. A purpose behind reverse engineering machine code is to find bugs. Detecting potential buffer overflows, unbounded jumps, memory corruption and other bugs can be detected using reverse engineering. Function matching can aid reverse engineering by allowing common functions like imported functions or trampoline functions to be easily paired together as a match. This can be summarized into the main purposes of version differentiating, patch and exploit management, bug discovery, and increased productivity in general. Function matching will allow reverse engineers to save time, find new or interesting functions, and identify similar security flaws between binaries. Once function matching is done, manual analysis must be applied to the new functions.

One technique of manual analysis is static analysis [14]. This form of analysis involves looking at assembly code through disassembly and attempting to infer what a function does through solely reading and coming to conclusions based on previous knowledge or experience. Another method of analysis is dynamic code analysis. Dynamic code analysis allows the reverse engineer to simulate running a program to see what happens during run time. Symbolic

execution is a mix of static and dynamic analysis that also allows the reverse engineer to see what inputs cause what paths of a program to run.

Symbolic Analysis

Symbolic execution refers to a class of analysis techniques that allow a user to dynamically gather information from a program. This is done through supplying arbitrary values instead of concrete values during dynamic analysis. At the beginning of execution, a symbolic execution engine will make any variables that cannot be determined at run time symbolic, meaning they have no concrete value. Then it will execute through the program stepping until it reaches a conditional branch. A constraint is then added to the symbolic variable and two new paths spawn from this branch. The engine will continue throughout the function this way until it has reached the end of all the possible paths. This analysis provides full code coverage however it is computationally expensive. Symbolic execution is the chosen method of analysis for this project because of its code coverage, supported architectures, and the projected accuracy it could provide for function matching.

2.4 *angr*'s Symbolic Execution Engine

angr's symbolic execution engine version 6.7.6.9 was chosen for this project. This is a binary analysis framework written by researchers at UC Santa Barbara [15]. It provides both static and dynamic analysis techniques. One analysis technique it provides is the extraction of Control Flow Graphs that allow a user to see how execution flows through the binary, including information on functions, basic blocks, states and more.

Injecting a binary into *angr* for our purposes involves executing a function and analyzing the final state of the function. A sample program in Figure 4 can demonstrate how symbolic variables are created during execution of a function.

```
int add_5(int a) {  
    int final_value = 0;  
    final_value = a + 5;  
    return final_value;  
}
```

Figure 4. This is a sample program. The parameter 'a' will be made symbolic at the beginning of the function analysis and then later 'final_value' will also be symbolic because of its dependency on a symbolic variable.

For the function in Figure 4, *angr* will first make the input symbolic because of our specifications. Then `final_value` will concretely be assigned to 0. In the next step `final_value`

will be reliant on the symbolic value 'a' and 5, with the operation of 'add'. This creates a tree that looks like the diagram in Figure 5.

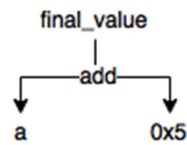


Figure 5. The data dependency tree for 'final_value'.

This a small example of how *angr* would begin to keep track of symbolic analysis during execution. Once conditionals are added to a function this process becomes much more complicated and this is described more later.

Basics

angr needs to be initialized by loading a project. From there the user can define the *state* to begin analysis at. This state can start at a specific entry point, it can be forced to make the environment symbolic and unconstrained, or it can be unspecified and the program will run normally by starting at the entry point of the binary. Then the user must create a path and path group for execution. A *path* is a collection of basic blocks that have been traveled up to the current state and the path's state stores all the information such as the constraints that caused that path to be taken and additional metadata. A *path group* is a collection of paths. *angr* will initialize the path group to solely contain the initial path to start the program analysis. Below is a sample script for starting up the analysis.

```
import angr

project = angr.Project(<Binary file>)
state = project.factory.entry_state()
path = project.factory.path(state)
pathgroup = project.factory.path_group(path)
```

Figure 6. Initializing a project in *angr*.

The script in Figure 6 demonstrates how to set up a project in *angr* using python. When *angr* first gets a new binary to analyze symbolically it will begin by first lifting the binary into an intermediate representation known as VEX. The conversion to intermediate representation transforms instructions into a new non-architecture dependent representation of the code. This allows *angr* to support many architectures including ARM, MIPS, PowerPC, X86 and more. All the analysis techniques from *angr* used for this project rely on this intermediate representation.

After initialization of the project, state, path and path group, there are many analysis techniques that could be applied using *angr*. The Control Flow Graph structure in *angr* is the most important collection of analysis results since almost everything else relies on it.

Control Flow Graphs

angr has a structure that they term a *Control Flow Graph*. An *angr* CFG is a graph that has basic blocks as its nodes and its edges are jumps and returns. The term CFG is not specific to *angr* and, in fact, *angr* adds a lot of other analysis and tracking data into the CFG, where other CFG's would not contain this information. In a CFG, each edge represents a call, branch, jump or return to the basic block it's directed to. The basic blocks are the nodes because each basic block defines a section of code without any branches. Each time there is a branch, such as a conditional statement, a constraint is added to both new paths and the CFG splits to show the possible successors for that return. A constraint is the condition or limitation of each path that spawns from a conditional statement.

Our proposed function matcher relies on the code flow throughout a function. This visual, in Figure 7, allows us to see the interactions between basic blocks. These interactions are important because together, they form a possible path through a function. Each time a basic block splits into two more basic blocks that is creating two potential paths of execution. Each possible path that could be taken is denoted by the decisions made during execution. At the end of each basic block there is a jump, return or some sort of change in code flow. For example, in the `func+0x14` block we can see the `jle 0x400533` instruction. This represents a conditional statement and a constraint that is being added to both subsequent possible paths. These constraints and paths are used in our function matcher.

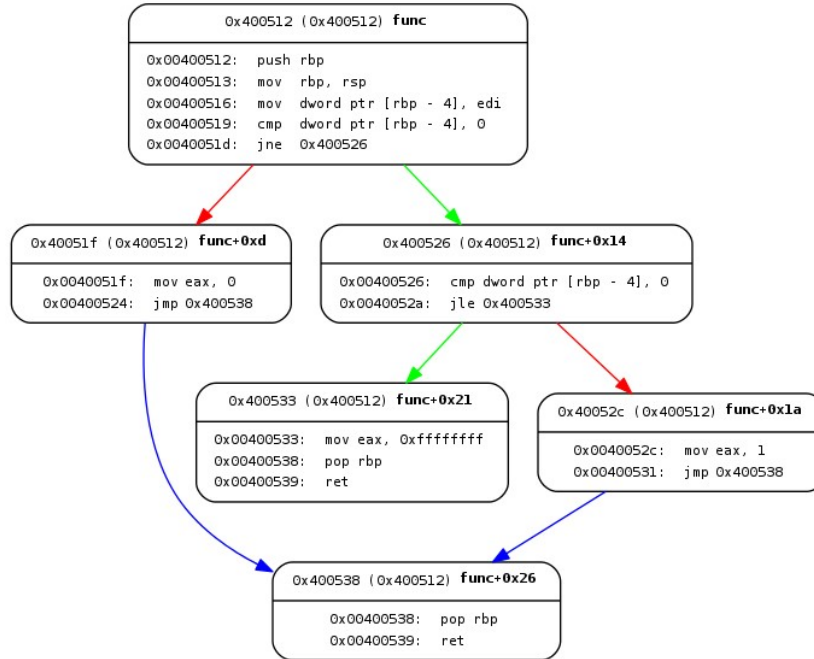


Figure 7. A CFG demonstrating relationships between basic blocks.

angr stores the active paths in a stash of paths separately and the Control Flow Graph is generated through stepping each active path until every path has dead-ended, meaning it steps through execution until there are no more successors. More information on Control Flow Graphs in *angr* can be found in Appendix A.

Constraints

When *angr* is performing symbolic execution each time a path reaches a conditional statement, a jump call, or any branch, this causes the path to be split into two new paths. When this occurs, the solver engine adds a constraint to each path which denotes what conditions would allow that path to continue based on what the condition in the branch causing operation was. For example, if the code contained a statement of `if(counter < 6){}`, this would cause one active branch to branch into two paths. One of these paths would contain the constraint of `counter is less than 6` and the other path would have a constraint of `counter is greater than or equal to 6`. These constraints are stored as abstract syntax trees.

```
<BV32 reg_10_9_32> __SLE__ <BV32 0x0>
```

Figure 8. Constraints in *angr* are stored as abstract syntax trees.

Figure 8 provides an example of a constraint on a path for x86 where the register is signed less than or equal to 0. Here the left leaf is symbolic because it does not have a concrete value however the right leaf is concretely assigned to 0. This figure also demonstrates the BitVector naming notation.

System State

During the dynamic analysis that *angr* provides, a user can look at the program state at any point in the program. This state provides information on the stack, registers and more. Figure 9 demonstrates what printing register values would look like from the terminal with the register name, value and whether it is symbolic at that point in time.

eax	<BV32 0x0>	False
ecx	<BV32 reg_18_13_32>	True
edx	<BV32 reg_20_14_32>	True
ebx	<BV32 reg_28_15_32>	True
esp	<BV32 0xffff0000>	False
sbp	<BV32 Reverse (Reverse (reg_38_11_62))	True
esi	<BV32 reg_40_16_32>	True
edi	<BV32 Reverse (Reverse (reg_48_12_64))	True

Figure 9. This shows a sample program state for the values stored in each register and whether the value is currently symbolic. This is an x86 program.

Summary

Overall *angr*'s symbolic execution engine provides a wide array of analysis techniques. These techniques ensure full code coverage because symbolic execution ventures down every possible path. Using symbolic execution to analyze binaries is a useful technique because it gives a functional analysis of a program rather than a static and string based analysis. Using these different forms of analysis provides a multitude of possibilities for matching techniques. More information on Value Set Analysis, Control Flow Graphs and more can be found in the Appendix.

2.5 Related Work

Symbolic Execution Engines

Symbolic execution has been around for at least a decade and just recently has been gaining interest again. The DARPA Cyber Grand Challenge, a challenge created by the Defense

Advanced Research Projects Agency to develop automatic defense systems that can discover, prove and correct software flaws in real-time [16], brought forth a new frontier in symbolic executions capabilities. This included tools like *angr*, Mayhem [17] and more. Mayhem was the ultimate winner of the competition with a symbolic executor and directed fuzzer [17]. Alongside Mayhem there are plenty more symbolic executors, assisted fuzzers, and more in the field of symbolic execution.

Another one of these symbolic executors is CUTE. This is a tool that is used to test C programs and concurrent Java programs. This uses symbolic execution to generate test cases and improve code coverage. While *angr* is used for reverse engineering, CUTE is generally used as a software testing tool. Their goal is to generate concrete values for inputs that would allow all paths of a function to be taken [18]. To do this they combine concrete execution with symbolic execution. Often there will be techniques added alongside using symbolic execution to avoid the memory and computation issues that symbolic execution presents.

Symbolic execution can be used for a variety of use cases. It is most commonly used for its full code coverage. Tools like CUTE, Cloud9 and more use symbolic execution in conjunction with other tools to provide support for software testing. Some tools use symbolic execution for vulnerability analysis, such as Kudzu. Tools such as SAGE, implement symbolic execution in conjunction with fuzzing to create a smart fuzzer. Symbolic execution provides support for a variety of use cases within the testing and vulnerability analysis fields.

Symbolic Execution and Function Matching

To the best of our knowledge, we are the first to apply symbolic execution to function matching. The most similar matchers in the field, however, are Abstract Syntax Tree matchers [19] [20]. *Abstract Syntax Trees* are the tree of variables and how they relate to each other throughout a program. Most commonly, they are generated for an entire binary, however specifically for *angr* they are only generated for a single constraint. These Abstract Syntax Trees for the full binary have been researched in relation to binary function matching. Our matcher has advantages over this matcher because it provides a more accurate match through matching by value rather than matching using hashing.

Clone detection is the concept of attempting to match binaries to fine sections of code that are performing the same comparisons. This technique is generally applied to student work to detect students cheating on computer science assignments. In a study on clone detection, the authors were attempting to detect near miss clones. In their paper, they reference hashing the ASTs for an entire binary to determine exact tree matches [19]. This is a concrete technique to find a perfect match however they also implement methods to find the near miss matches. This involves comparing every subtree to other subtrees for equality [19]. One way that their method of comparing ASTs is different than this proposed method using *angr*, is that they have a single Abstract Syntax Tree for the entire binary. Their tree is linked, meaning that sometimes the

leaves in a tree can also have leaves of its own. *angr* however, does not store constraints as linked Abstract Syntax Trees. This would lead to more complexity and scaling problems with their constraint solver. If given linked Abstract Syntax Trees in *angr* this would provide the opportunity for graph matching techniques to be applied as well as value comparison however it would introduce recursion issues and other memory issues. The full binary AST for the clone detection research was solely hashed then compared, whereas these matchers put in the time to do a value comparison as well as the string comparison. Regular ASTs provide enough information for the matchers written during this project.

3. System Design

3.1 Overview

Approach

We used constraint comparison to match functions based on the constraints placed on each path within a function. This allows for the functions to be matched based on what they rely on for input. This could include dependencies on user input, parameters to the function, memory reads and more. This allows functions to be matched regardless of the target architecture. Using symbolic analysis, the function is guaranteed to be fully analyzed. Through using this information, we can match functions based on what they rely on and how this affects their code flow.

An overview for how the different pieces interact for this project is shown in Figure 10. This involves at least two binaries being extracted. After extraction, this provides us our architecture-agnostic representation of all functions in the binary. We can then ingest these functions into our pairwise matchers. This class of matchers will then provide matches between functions from the two binaries.

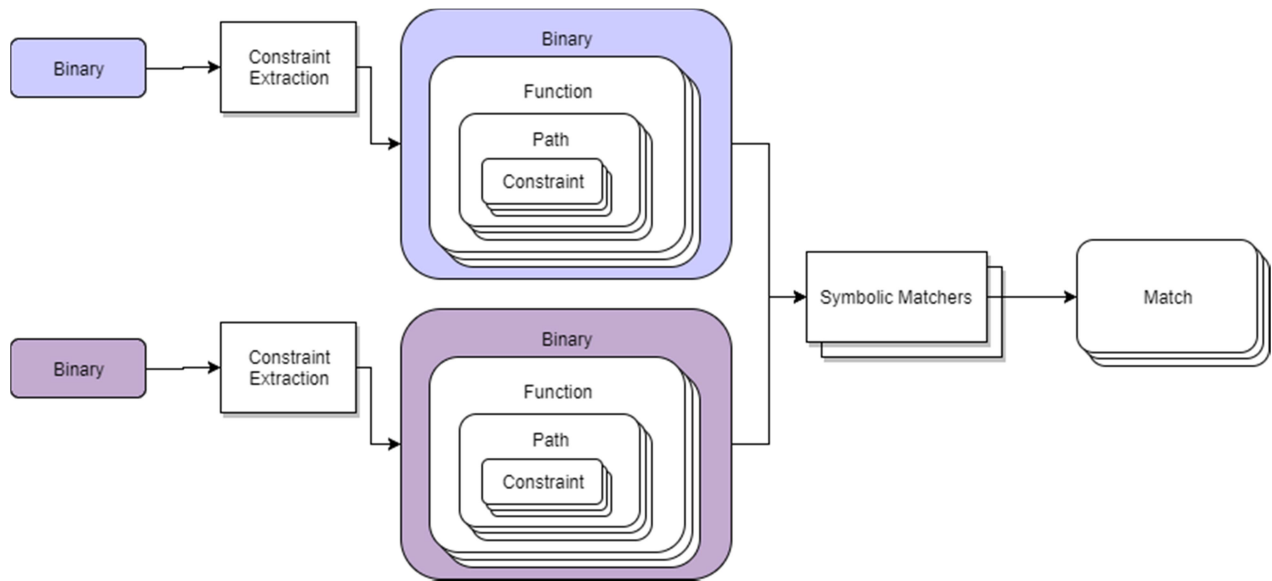


Figure 10. Overall flow chart of our proposed matching and extraction techniques.

We designed three matchers for this project. The first is the symbolic constraint value matcher. This matcher attempts to exactly match constraints with between functions found in different binaries. The second is a symbolic constraint string matcher. This matcher hashes and then uses LCS, described in Appendix C, to match the constraints and only works for the same

architecture binaries. The final matcher is a symbolic constraint fuzzy matcher. This matcher allows for functions to be matched even with some constraint mismatches.

3.2 Constraint Extraction

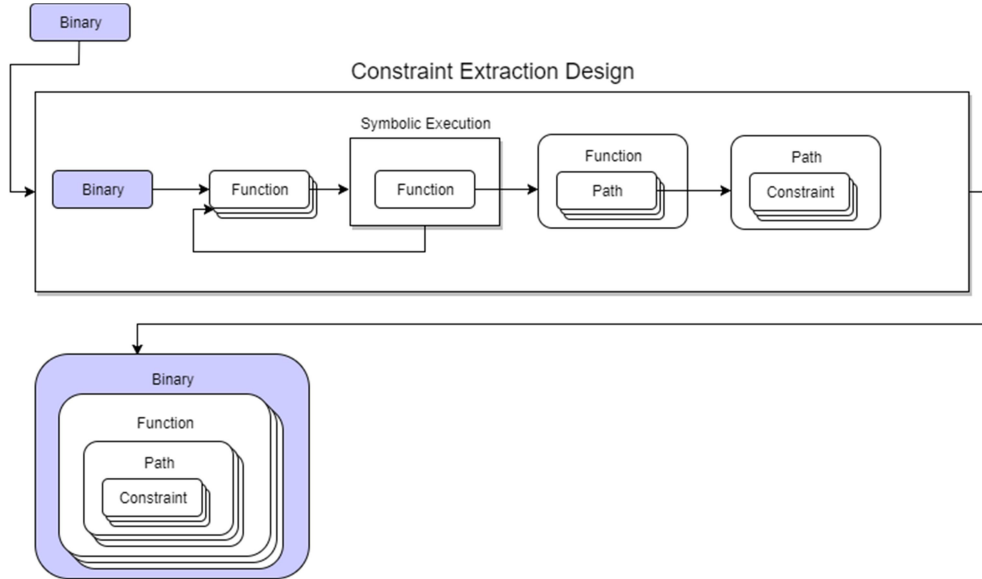


Figure 11. Constraint extraction design for our system.

Constraint extraction is the process of running symbolic execution on every function in a binary for extracting the function’s paths and constraints. Constraint comparison relies on the function to be symbolically executed to determine what the constraints are. The first step taken here is finding the functions. Through looking at the Control Flow Graph we can find the start address of all the functions in the binary. We can then ingest the function into symbolic execution by setting the start state to be blank and to start at the beginning of the function using the entry address. A *blank start state* means that everything is symbolic and unconstrained including all the registers and memory. Doing this allows the function that is being analyzed, to be analyzed standalone, meaning whatever has happened before this function began is not considered.

The *callout sites*, places in a function where the function calls another function or jumps to an outside location, also need to be considered. During symbolic execution of a function the default of *angr* is to call any callout sites with the appropriate parameter values and attempt to execute through the function with those concrete parameters. Sometimes this is a simple process however, sometimes this will cause many more constraints to be added to the path from the called function, to the function in analysis. This large number of constraints is too much of a performance burden therefore during analysis of a function all we overwrite all callout functions using a hook procedure that effectively *no-ops*, removes through adding instructions that do

nothing, the function call. When this hook procedure is implemented to no-op the function calls, this includes tail calls. These cases are handled during execution.

After the function is set up and ready, the extractor will execute through the function with parameters being made symbolic, any memory reads or user input is also made symbolic since these cannot be concretely determined during symbolic run time. Symbolic execution does have some limitations, e.g., loop detection and handling. During execution, the extractor will attempt to determine the path is in a loop by looking at the current address and the past addresses of that path. If the current address is in the list of past addresses, this means that the program is in a loop. To avoid looping without concrete boundaries, the script will check the possible successors of that basic block. If there is only one successor to the basic block, that means the loop is concrete and therefore can be feasibly run during symbolic analysis. However, if there is more than one successor, this is a symbolic loop and the script will dead-end the path that continues through the loop and allow the path that exits the loop to continue execution.

One use case to consider when looking at successors is the tail call scenario. Since function calls are *callee cleanup*, meaning the function that is called will clean up the stack and instruction pointer after execution, and are also overwritten using blank operations, being *no-op'd*, at the beginning of the script, this causes an issue when it comes to looking at the possible successors of a basic block. Since the actual successor of the final basic block of a function is the address of the next function and that address is overwritten, *angr* determines based on the path that the next successor should be the next possible address. This address could be the start address of another function, or depending on the compiler, the address could be in the middle of the current function because of the way the code aligned during compilation. Both scenarios are unsatisfactory since the only actual successor of that basic block during normal execution was the exit function call. The CFG for the project, however, recognizes the actual successors and solely lists the address of the exit function call as the successor to the final basic block. This allows us to compare the proposed successor of the path with the no-op'd function calls to the actual CFG projected successors to determine if this successor is attempting to execute a reasonable address.

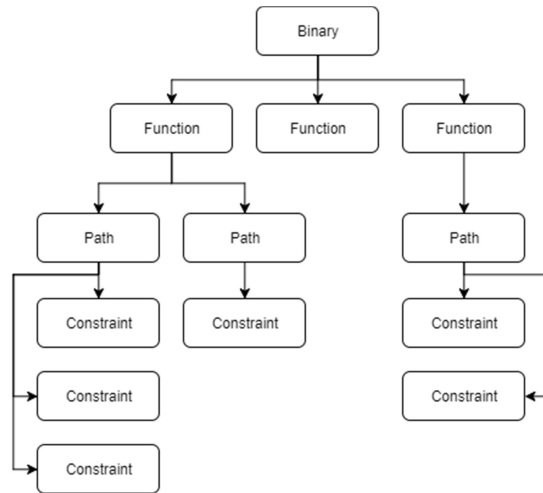


Figure 12. Demonstrates how the extractor stores the results in the graph database. After symbolic execution, each function has different possible paths of execution with their constraints stored on the path.

After symbolic execution, has completed through the function ignoring all function calls and avoiding symbolic loops, a group of paths is the outcome. These paths are considered dead-ended, which means they have executed until there are no more successors in that function for them to execute at. Each of these paths contains the constraints that allowed this path to execute. An example of what this output looks like is in Figure 12. This just shows the view of the structure however there is more data stored inside each of the components that are printed.

These paths and constraints are the focus of our class of matchers. As shown in Figure 13 these constraints can appear complex due to the naming scheme. This is because of the register, memory and temporary variable names assigned during the intermediate representation's single static assignment. When interpreting the meaning of this constraint the most important pieces of information come from determining if the leaves are symbolic or concrete and what operation resides between them. These pieces of information can be extracted through formatting the string representation. Something to consider here is that depending on the computer architecture the same constraints could be different, such as in a different order, although the original C source code is the same. The ordering of leaves is dependent on the architecture so, to analyze these, considering the commutative property allows us to determine if swapping the arguments constitutes a match. These considerations are applied during the matching process.

```

<Path with 5 runs (at 0x4000040 : ##angr_externs##)>
[<Bool reg_48_10_64[31:0] >s 0x63>]
<Path with 7 runs (at 0x4000040 : ##angr_externs##)>
[<Bool reg_48_10_64[31:0] <=s 0x63>, <Bool reg_48_10_64[31:0] >s
0x31>]
<Path with 8 runs (at 0x4000040 : ##angr_externs##)>
[<Bool reg_48_10_64[31:0] <=s 0x63>, <Bool reg_48_10_64[31:0]
<=s 0x31>, <Bool reg_48_10_64[31:0] <=s 0x18>]
<Path with 8 runs (at 0x4000040 : ##angr_externs##)>
[<Bool reg_48_10_64[31:0] <=s 0x63>, <Bool reg_48_10_64[31:0]
<=s 0x31>, <Bool reg_48_10_64[31:0] >s 0x18>]

```

Figure 13. A demonstration of 3 different possible paths and the constraints that live on those paths. This is from a binary compiled for x86.

During constraint extraction, some error handling techniques need to be applied. Sometimes, *angr* cannot concretize variables into nice comparisons as shown in Figure 14. This will lead to an incredibly long representation of variables that contain every possible value, or sometimes even just one randomized value. For example, if the program has a segment that states `eax != mem_addr` and both of these variables are symbolic, this can lead to concretization issues. If neither of the symbolic variables can be neatly concretized into a value, then sometimes *angr* will store one of the variables in every possible form with the given constraints that previously existed on that variable. Say earlier in the program `mem_addr` was set to be between `0x400000` and `0x400010` and *angr* couldn't determine the final value, then the constraint could be stored like this: `if <BV32 eax == 0x400000> if <BV32 eax == 400001> if...` and so on. An actual example of this is shown in Figure 14.

```

<BV64 if (mem_fffffffff000000_145_2048[7:0] == 0) then
0xfffffffff000000 else (if (mem_fffffffff000000_145_2048[15:8] ==
0) then 0xfffffffff000001 else (if
(mem_fffffffff000000_145_2048[23:16] == 0) then 0xfffffffff000002
else (if mem_fffffffff000000_145_2048[31:24] == 0) then
0xfffffffff000003 else (if (mem_fffffffff000000_145_2048[39:32] ==
0) then 0xfffffffff000004 else (if
(mem_fffffffff000000_145_2048[47:40] == 0) then 0xfffffffff000005
else (if (mem_fffffffff000000_145_2048[55:48] == 0) then
0xfffffffff000006 else (if (mem_fffffffff000000_145_2048[63:56] ==
0) then 0xfffffffff000007 else (if
(mem_fffffffff000000_145_2048[71:64] == 0) then 0xfffffffff000008
else (if (mem_fffffffff000000_145_2048[79:72] == 0) then
0xfffffffff000009 else (if (mem_fffffffff000000_145_2048[87:80] ==
0) then 0xfffffffff00000a else (if
(mem_fffffffff000000_145_2048[95:88] == 0) then 0xfffffffff00000b
else (if (mem_fffffffff000000_145_2048[103:96] == 0) then
0xfffffffff00000c else (if (mem_fffffffff000000_145_2048[111:104]
== 0) then 0xfffffffff00000d else (if
(mem_fffffffff000000_145_2048[119:112] == 0) ETC...

```

Figure 14. Example of an error that can occur in angr when the constraint solving does not concretize properly.

When this happens, the constraint is unusable for any sort of matching technique and therefore an error is appended to the constraint instead. This error is later handled during matching.

Another possible error in *angr*'s concretization strategies occurs when the framework accidentally stores both leaves with the operation in one leaf. There are a few causes behind this happening. The not operation only takes one argument therefore sometimes the second leaf is unnecessary for the instruction. Also, the important leaf could be complex due to previous operations. An example of what this would look like is leaf1: <BV32 reg_11_30_32 __SLE__ BV32 0x1> operation: NOT leaf2: '. Trying to match this with another constraint and hoping the same error happened during symbolic execution is not reasonable. Therefore, an error is added to this constraint as well which is also handled during matching.

3.3 Matcher Overview

Using constraints to match functions gives full code coverage and allows the user to match based on what the function relies on. To match constraints, they must first be extracted

using the techniques discussed in the above section. After the extraction has completed, this provides all possible paths that could have been taken during execution of the functions in a binary and the constraints that allowed the program to take those paths. These are then compared using a combination of techniques. Matching is done at a functional level rather than a full binary being compared to another binary. It will take in two full binaries and pairwise match the functions from each binary.

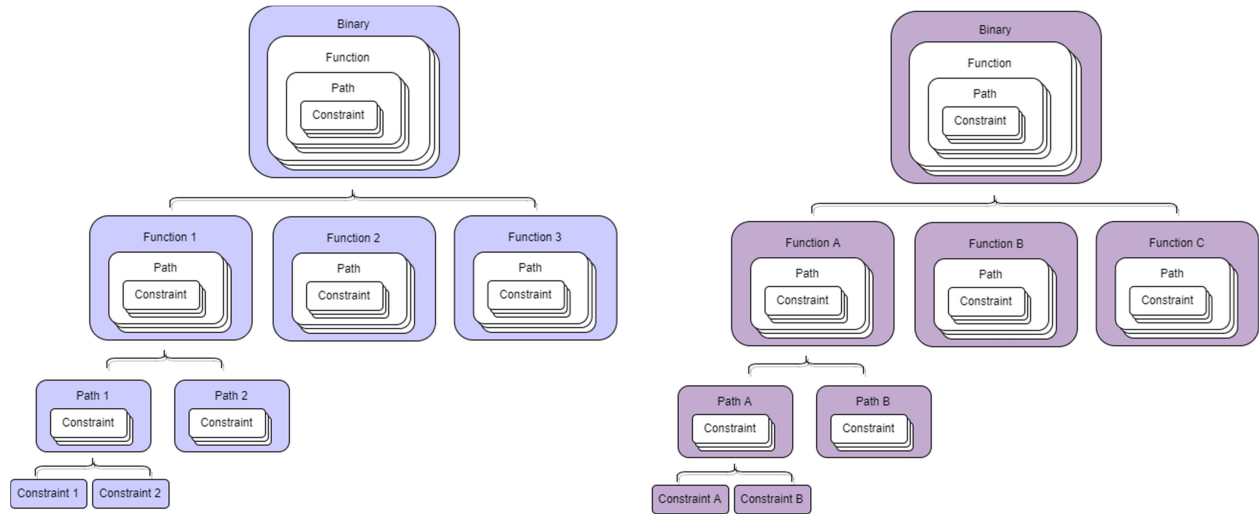


Figure 15. A demonstration of the output from extraction. Depending on the function there can be more or fewer paths and depending on the path there could be more or fewer constraints.

Looking at this sample output from the extractor allows us to demonstrate how the matcher will use this information. First Function 1 and Function A will be matched. Once these two functions have been ingested into the matcher the matcher will first look at Path 1 and Path A. Looking deeper, this involves looking at the constraints from these paths. Constraint 1 will be matched to Constraint A. After this similarity is determined, Constraint 1 will be matched to Constraint B. Every constraint pair must be matched. This would continue until Constraint 1 had been matched to every constraint from Path A. Constraint 2 will follow the same process; first being matched to Constraint A, Constraint B and so on. Once these are done this gives us the constraint scoring matrix for Path 1 and Path A. This will be described more in depth later. Path 1 must then be compared to Path B and so on. This continues until Path 1 has been matched to all the paths from Function A. Path 2 will then be compared to all the functions from Function A. Once these have completed there will be a path matrix which is then scored to provide a final similarity between Function 1 and Function A. Function 1 will then be matched to every possible function from Binary 2. Following this Function 2 will be matched to every possible function from Binary 2 and so on for the rest of the functions.

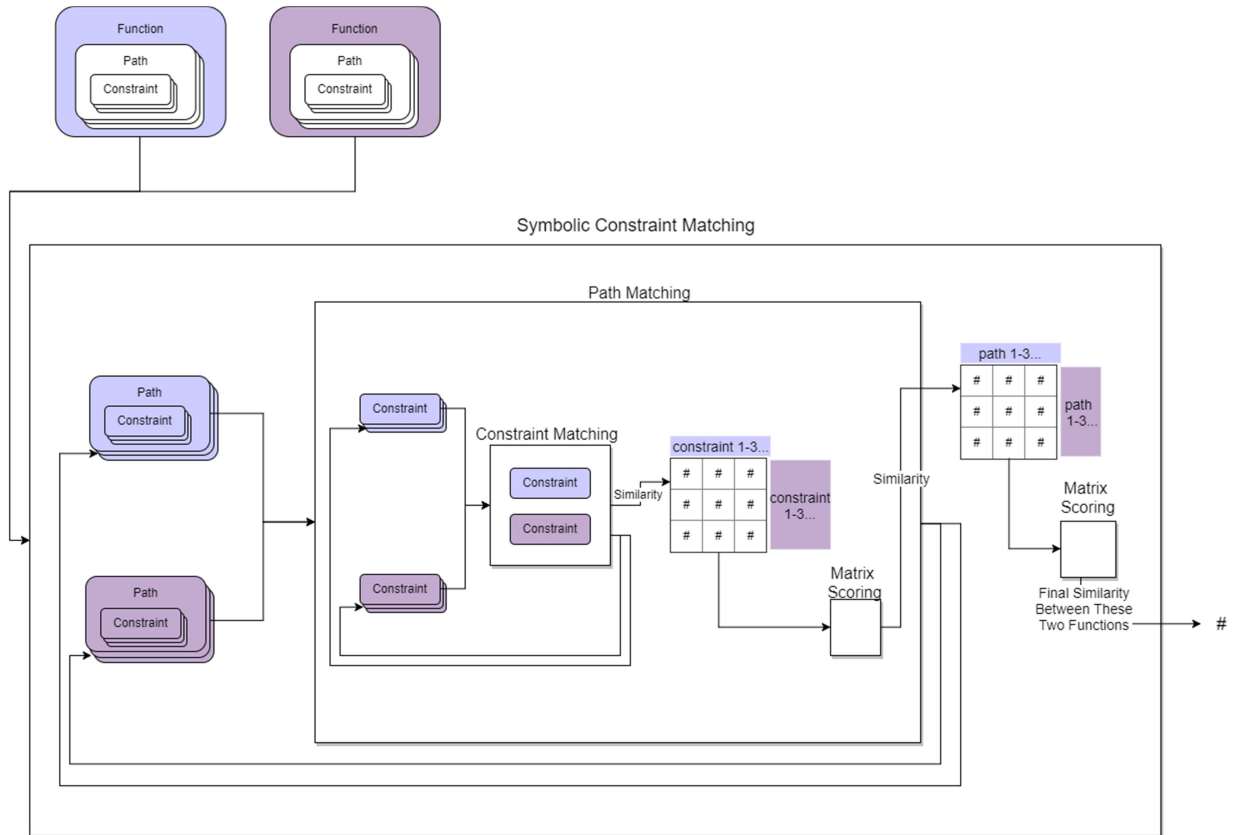


Figure 16. Flow graph for all matchers. The only place where matchers differ is within the Constraint Matching methodology.

Ingesting pairwise functions into the matcher is demonstrated in Figure 16. Through using preprocessing techniques, we will avoid wasting time matching functions that clearly will not provide a match. These techniques are described in Appendix C. The process described above involving looping through paths and constraints is demonstrated here as well. Once the matcher determines that the function, path, and list of constraints can be reasonably matched, each of these constraints needs to be compared; this process is described in Figure 16 as *Constraint Matching*. This is the part where each of the matchers from the class of matchers differ. The actual constraint comparison for each matcher is described in their respective chapter.

3.4 Symbolic Constraint Value Matcher

Since constraints are stored as abstract syntax trees, it is reasonable to match them with 100% similarity in this value matching scenario. Given there were no errors during extraction stored on the constraint, each leaf can be matched and the operation between the two will also be checked to make sure that they are the same. These three conditions being satisfied constitutes a 100% match. The actual matching of constraints is done by first comparing the first leaf to the other first leaf. If these are a match then the second leaves are matched, finally the operation is

matched between them. If the first two were not a match, the first leaf will be matched to the second leaf. If they are a match the second leaf of the first constraint will be matched to the first leaf of the second constraint and the operation will be swapped. A simpler example of this is demonstrated in Figure 17 where x , leaf1, $<$, operation, and 5, leaf2, are all a perfect match to $x < 5$ on the other axis and therefore it is a 100% match. After each constraint has been matched with every other constraint from the other path, there is a final constraint matching matrix like Figure 17.

	X<5	X>=5	Y!=0	A=True
X>=5	0	100	0	0
X<5	100	0	0	0
A=True	0	0	0	100
Y!=0	0	0	100	0

Figure 17. An example of a constraint matrix.

Once there is a matrix for the constraints from the first pass of matching a pair of paths the Hungarian method, described in Appendix C, is implemented to find the best one to one match between constraints. If there are too many constraints however, a brute force method is applied to save time and memory. After one of these methods of matrix scoring is applied, the return is a list of the ideal constraint matches and the similarity between them. These similarity values are then averaged together to create a final similarity between the two paths. The script's loop will then loop through the rest of the possible path pairs from each function and compare their respective constraints. Finally, a matrix will have been created for the paths from function one and the paths from function two. This can then be scored using the same methods applied for matrix scoring for constraints. This gives a final similarity value between the two functions.

One consideration to make when comparing constraints is that the leaves can be stored with indexes in their naming scheme that could cause a missed match. The names of the temporary variables need to be altered to dis-include the indexing. Since single static assignment only uses one variable name for one purpose this could alter the matching of constraints with variables when including their indexes. Through removing the index, the function that was compiled at an earlier offset and therefore has small temporary variable indexes can still be matched with a function that had larger indexes due to being farther down in the source code.

Some architectures order their opcodes differently which needs to be considered when matching constraints since they won't be in the same order for two different architectures, even if they are functionally doing the same comparison. Another thing to keep in mind when matching cross architecture, and cross compiler, is that the leaves can be stored in the opposite order. Rather than having x is less than 1, another architecture might have 1 is greater than x . To avoid overlooking this match, the matcher needs to check for scenarios like this.

reg_48_10_64[31:0] <=s 0x63	0x63 >=s reg_8_13_32
reg_48_10_64[31:0] <=s 0x31	0x31 >=s reg_8_13_32
reg_48_10_64[31:0] >s 0x18	0x18 <s reg_8_13_32

X86

ARM

Figure 18. On the left is x86 and on the right is the same source code compiled for arm. As shown, arm and x86 order their variable and constants differently. This is the final possible path's constraints.

To check for these scenarios, when the matcher doesn't find a match in the original order, one of the constraints will be swapped. Swapping the constraint takes the second leaf and the first leaf and switches them, then reverses the signed operation between them. For example, if the operation was originally less than the opposite would be greater than. The constraints can then attempt to be matched again.

3.5 Symbolic Constraint Fuzzy Value Matcher

This version of the constraint value matcher is used to allow for small deviations within the similarity of a match between constraints. Where the Symbolic Constraint Value Matcher only allows constraints to be matched either perfectly with 100 or not at all with 0, this matcher was written to test situations that could be considered close enough to be a match. For this matcher, the only difference is in constraint comparison. The Symbolic Constraint Value Matcher tests for the left leaf, right leaf, and operation between the two all to be the same and then can assign that match a 100% similarity. In this matcher, the Symbolic Constraint Fuzzy Value Matcher, it allows a 66% match to be created when both leaves are the same and the operation is different and allows for a 33% match to be created when only one leaf is the same. The idea behind allowing these matches to be made was the idea that since these matchers are matching based on the dependencies of the function, it is reasonable to match an example situation where leaf 1 is a file line and leaf two is the concrete string that is expected, even if the operation is not the same. These constraints are testing whether the function is reading from the same memory space and making a jump based on what that memory is. This concept of the functions relying on a certain memory address or register doesn't necessarily require the operation to be the same. If the memory address is the same, then those two functions are calling out to the same place which makes them somewhat similar.

	X<5	X>=5	Y!=0	A=True
X>=5	66	100	0	0
X<5	100	66	0	0
A=True	0	0	0	100
Y!=0	0	0	100	0

Figure 19. An example of the same constraint matrix as in Figure 18, however this one uses fuzzy matching.

Here constraints can still be granted 100% similarity, however if the operation is different between them and the leaves are the same, they are granted 66% similarity in the match. In allowing for partial matches to be made this allows for more variation in the functions even if they are still considered a match. However, functions will still not be considered a match if after matrix scoring of the constraint and path matrices do not produce a final similarity of higher than 70%.

3.6 Symbolic Constraint String Matcher

Since comparing constraints using value matching can be computationally expensive, a simple string comparison using constraints was implemented. This string comparison uses the same preprocessing techniques to avoid comparing paths from different functions that are clearly not a match. For example, one path has one constraint and the other has fifty and these are clearly not the same function and if either of the constraints have errors they will obviously not be matched.

The only difference between value and string comparison with constraints is in the actual constraint comparison. While value comparison can look at both leaves and determine if they need to be swapped and can also allow for removal of register indexes during comparison, string matching does not do this. String matching simply takes the list of constraints from the path, hashes them using a sha256, and compares them to the string of constraints from another path using the Longest Common Subsequence algorithm to determine how similar they are. The LCS minimum confidence is set at 75. The similarity is then injected into the paths matrix and the rest of constraint string matching is done the same way as constraint value matching.

4. Methodology

4.1 Datasets

We collected a wide variety of binaries to test symbolic execution’s functionality and the accuracy of the written matchers. *angr* claims to support a variety of architectures and a large variety of operation types so initially tests were run to ensure *angr* was a reliable symbolic execution engine. We manually compiled binaries from a variety of simple C programs (see Appendices E-I) to test a large variety of situations. These programs consisted of loops, conditionals, function dependency, recursion, use of signed and unsigned integers, memory reads and writes and other simple programs to understand the impact of common control and data structures on symbolic constraints and, consequent, the performance of our proposed matchers. These were easily cross compiled for both x86 and ARM. After they were used to test *angr*, they were also used to test the matchers.

Size (KB)	Filename	Source	Compiled For...
8.4-8.8	Small self-written test binaries	Self-compiled	X86, ARM
26.7	Mysqldadmin	LEDE	X86, ARM, MIPS
37	Whoami	LEDE	X86, ARM, MIPS
129.2	True 8.26, True 8.27	CoreUtils	X86
129.2	False 8.26, False 8.27	CoreUtils	X86
138.9	Echo 8.26, Echo 8.27	CoreUtils	X86
143.6	Yes 8.26, Yes 8.27	CoreUtils	X86
179.3	Tcpbridge	LEDE	X86, ARM, MIPS
219.9	Sha256Sum 8.26, Sha256Sum 8.27	CoreUtils	X86
238	Printf 8.26, Printf 8.27	CoreUtils	X86
431.5	Wget	LEDE	X86, ARM, MIPS
745	Bash	LEDE	X86, ARM, MIPS
839.9	Git	LEDE	X86, ARM, MIPS

Figure 20. Summary of selected test data. A variety of architectures and code needed to be tested so self-written binaries were used to test cases like looping, recursion and more and the large binaries were used to test the usability for production.

In addition to the simple benchmarks described above, we also tested our matchers against a variety of real-world binaries. The idea behind selection of these binaries had a few criteria. There needed to be a wide range of sizes to test the speed and accuracy of the symbolic execution matchers in relation to size and number of basic blocks per function. These binaries also had to be found in the same version compiled for different architectures to check for real

world instances of cross architecture comparison. These binaries were found using the LEDE project [21] which is an open source project developed for router security.

For version proliferating, we chose some binaries from CoreUtils [22] in both versions 8.26 and 8.27. We wanted to easily test our function matchers with two binaries from the same architecture so we could conclude that file size does not impact the matcher's accuracy.

These test binaries were used to both test and evaluate the extractor and class of matchers. The provided confusion matrices use the test binaries to effectively demonstrate where our matchers succeeded and failed. This information can be found in the results and discussion chapter.

4.2 Ground Truth and Minimum Similarity

In binary matching, there are many opinions on what a true match can be defined as between two binaries' functions. Reverse engineers using function matching for different purposes will have different concepts of what is a true match. Many people will consider a 100% string comparison match using the opcodes of the function to be a perfect match. We define our true match to be two functions we manually analyze and determine are a match. We used BinDiff to assist our manual analysis.

Similarity is defined as how similar two functions are. As the developers of the symbolic execution constraint value matcher we define similarity based on how many of the constraints were matched. For a developer using LCS they may say that a LCS match is constituted by having 80% of the function matched consequently. Similarity is configurable for all matchers in this project and we can define the similarity that we want to use for any matcher.

Ground Truth

For our proposed matchers, a ground truth had to be determined to compare the accuracy of symbolic execution function matching to a set truth. We used manual analysis to match functions from different binaries. This involved using BinDiff to match functions that we know BinDiff can match, such as same architecture comparisons or thunk functions. In these cases, BinDiff giving 90% similarity on two functions was our choice of ground truth because it is industry standard and simple to run on binaries. Since BinDiff couldn't reliably be used to match cross-architecture, we manually determine if functions were supposed to match in the situations like this.

Minimum Similarity

Alongside this ground truth, a minimum similarity in the constraint matchers also must be determined. For symbolic execution through various manual tests, it is determined that a

minimal match of 70% similarity must be reached to consider the two functions a match. This is accounting for small variations between the two functions and still allowing them to be considered a match. Our confidence in the match is always dependent on how similar the functions are. For any function with over 70% similarity we are 100% confident that this is a good enough symbolic similarity to be considered a match.

4.3 Environment

We tested our project inside a virtual machine running Ubuntu 16.04. This machine had 32 GB of RAM and has a few other processes running at the same time as the symbolic execution processes. These outside processes include services like a running database. During extraction, the symbolic execution script will store the data into the database and during matching, the matcher will query the database for this information. These processes are constantly running in the background and use around 10GB of RAM while the symbolic execution function matcher and extractors are running. This affects the speed of the extractors and matchers.

5. Results and Discussion

5.1 Results

After running the test data with the extractor and matchers, we have realized that the thumbprint matcher can only do well for certain functions. To describe these ideal functions, we must first classify all function types. Figure 21 shows a few of our larger test binaries and the breakdown of each function type.

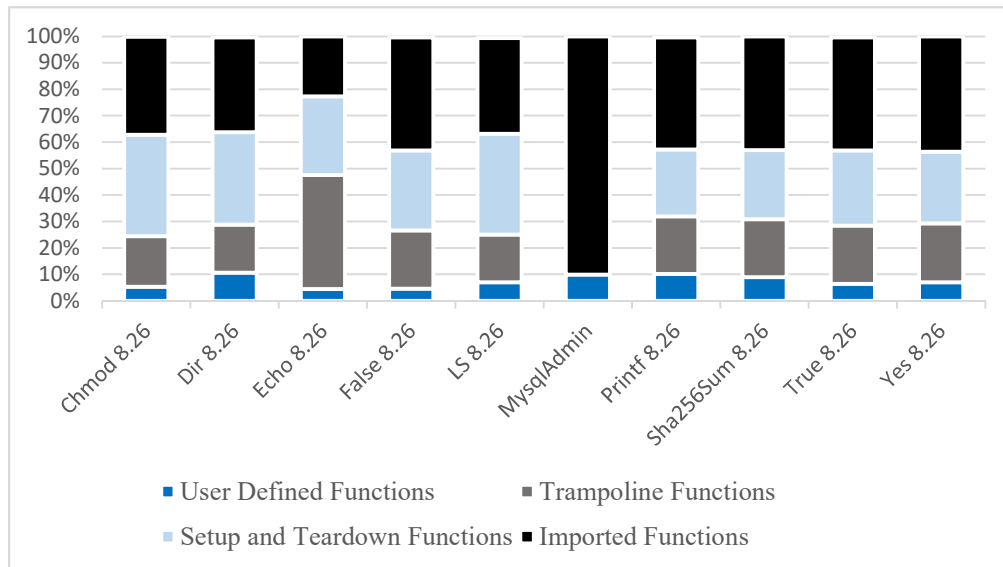


Figure 21. This figure shows the composition of a binary and the categorization of the functions by percentage of the binary.

User defined functions. These functions consist of the functions that the programmer directly wrote. These are the functions that most reverse engineers are most interested in matching between binaries.

Trampoline functions. These functions are solely used to call other functions in the program [23].

Setup or teardown functions. These functions consist of all the functions that set up a program, such as setting the entry point in `_start`.

Imported functions. The final type of function is called by trampoline functions during execution and is linked to an outside package or source.

These four types of functions only categorize what the function is doing, however, it is also important to consider if the specific function, regardless of category, will provide any

symbolic analysis results. Since the function matcher relies on path constraints, this matcher can only be used on functions that contain conditionals, read from files, or anything else that can cause something to be symbolic. Within the four types of functions, the user defined functions most often have constraints and dependencies which makes them the perfect candidate for function matching with symbolic execution. The other three function types can have constraints however matching these functions can usually just as easily be done with an LCS opcode matcher. This symbolic execution function matcher will provide the most interesting results on functions that are written by a user where there is a change in code flow due to run time data. The frequency for each function type having symbolic constraints is shown in Figure 22. This was generated using our test data which is described in the methodology chapter.

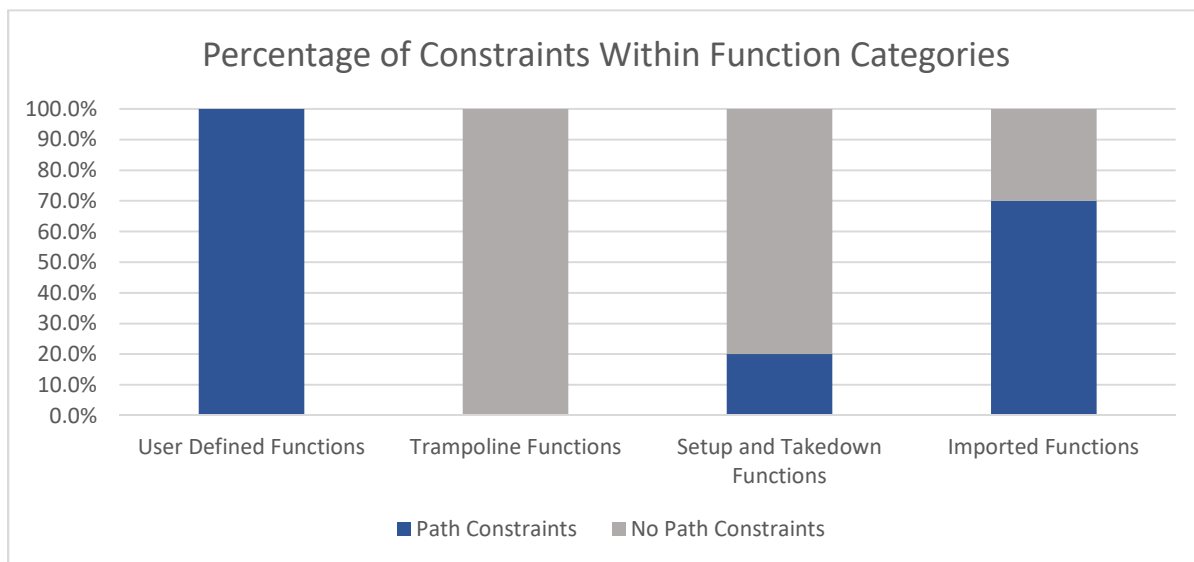


Figure 22. The percentage of functions of each type with constraints. Generated using our test data.

In general, these functions with constraints are either user defined functions or imported functions. That makes these functions usable for symbolic matching. Trying to match a function that does not have constraints like a trampoline function will not provide any results.

Something to consider is that *angr* creates its own version of some imported functions. The reasoning behind this is so when symbolic execution is running on a function that calls something, for example `strcmp`: the *angr* implementation will be a rewritten version of the function using the concrete parameters specified by the caller function. This avoids the potential path explosion problem of using the original imported function. Usually, the *angr* implementations of these functions will match assuming they were used in the same exact way in both binaries, such as having the same parameters upon call time from the same caller function. However, these functions are not expected to match because they are so specific to the exact parameters of the caller function. On average, *angr* will create their own version of functions for

approximately one third of the functions in the binary. These functions are generally repeats of imported functions which is one of the reasons that Figure 22 shows such a high percentage of constraints for imported functions.

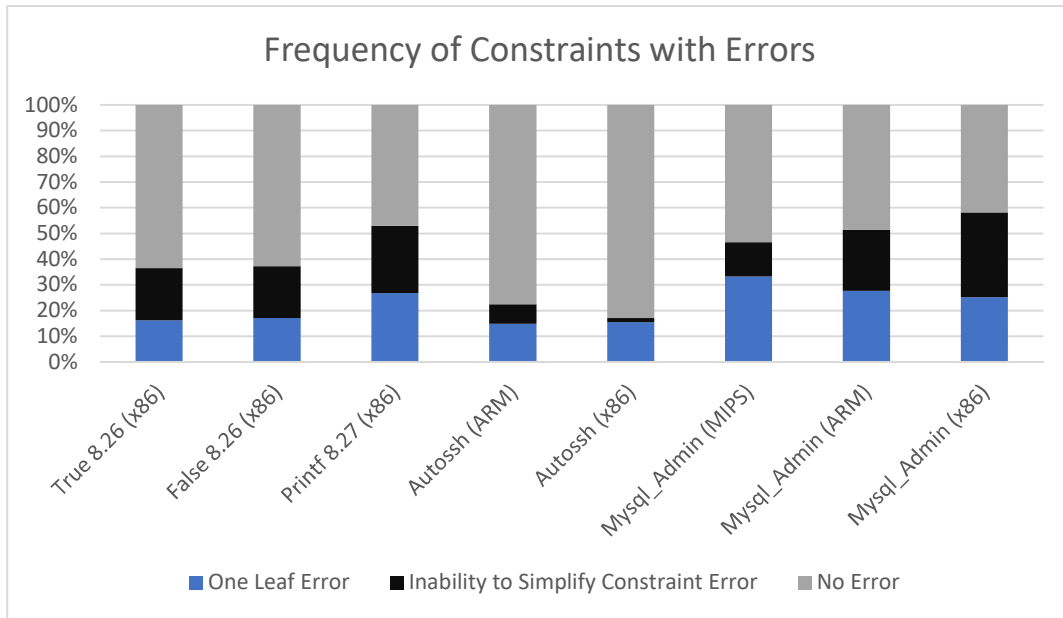


Figure 23. This shows the number of times *angr* threw errors on constraints during extraction for some of our test binaries.

Figure 23 is demonstrating how often *angr* could not properly concretize constraints during extraction. These errors are most common for the lesser used architectures like MIPS, which leads to issues during cross architecture comparison.

5.2 Evaluation

Finally, with all the considerations in place and understanding that the symbolic constraint matcher will work best for user defined functions with constraints and it also works for any function with constraints; the confusion matrix is shown in Figure 24. These numbers represent the comparisons between the pairwise functions. In Appendix D, there is a table describing in detail the binaries that were pairwise matched.

	Predicted Match	Predicted No Match
Match	85	0
No Match	27	1384

Figure 24. The confusion matrix for the pairwise function comparisons for functions with constraints only.

These comparisons involve cross architecture, same architecture and different versions of the same program comparisons. For this confusion matrix, only the ideal functions, functions with a change in code flow due to run time data, were considered. Any functions that could not be matched with our matcher, functions without path constraints, were not considered. The ground truth used here was manual analysis.

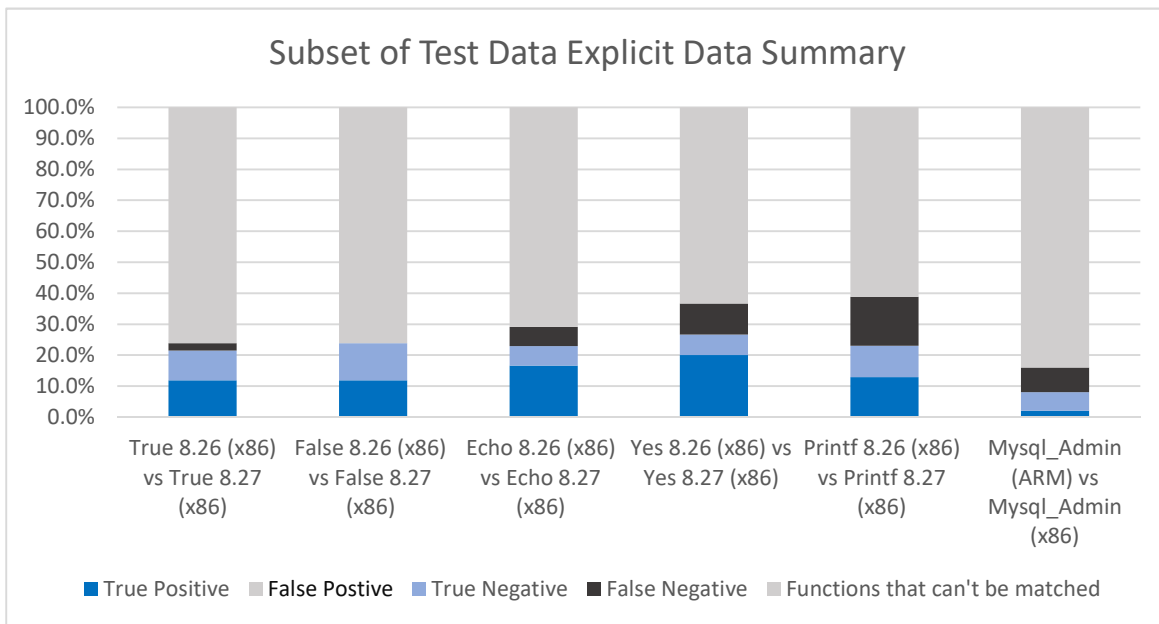


Figure 25. A subset of some of the test data to show how the distribution of functions that were matched with our class of matchers in comparison to how many functions there were in total.

Almost all of the causes of false negatives, missed matches, are due to *anqr*'s errors during the concretization of constraints in our extraction phase. When errors are thrown on the concretization of a constraint then that renders the constraint unusable for matching. This error is unavoidable however there is no great way to determine what to do with this error. In this project, anytime a constraint with an error is found it is automatically assumed to be a 0% match with the other constraint and the matcher moves on. One case where this could happen is during a file read. If *anqr* cannot neatly concretize the file line, then this will affect the constraint

matrix for this path even if there was a perfect match in the other function, had *angr* properly concretized the constraints. For a select subset of our test data these errors occur almost 50% of the time, and Figure 25 shows more in-depth analysis of how often it happened. This can cause some of the false negatives with these matchers.

A lot of times *angr* will implement their own version of a function such as `strncmp` to simplify symbolic execution. These extra functions are tacked on top of the existing functions and are another one of the causes of false negatives. Sometimes, a binary will have multiple *angr* implementations of `strncmp` based on how many times it is used in the other functions of a binary. Although matching these could seem like it could be a match or could be important, the *angr* implemented function does not matter regarding binary analysis because it is not a function that exists in the actual binary. These *angr* implemented functions can also account for false negative situations as well if they do not match the *angr* implemented version from the other binary. The cost of these false negatives is the wasted time attempting to match them and retrieving no data. Some ways to tell if one of these functions is an *angr* defined function is to see if it has a repeated name as one of the imported functions. For example, if *angr* replicates `strncmp` then there will be two `strncmp` functions in the binary after it is ingested into *angr*. We could manually look at those functions and ignore them during extraction and matching. These *angr* functions do not actually have any significance other than their use during the symbolic execution analysis.

It is also important to consider how these symbolic constraint matchers work on the larger scale. Although in an ideal world the human can see and know to use this matcher on the ideal function type, it is important to consider how comparable they are to other matchers in the reverse engineering fields. The industry standard is BinDiff and this tool was used as an aide in determining ground truth when considering how the written symbolic matchers compare to other tools. If we were to only use BinDiff to compare these binaries, BinDiff would be incapable of matching most cross architecture functions. BinDiff can match imported functions from two different architecture binaries, but cannot match the user defined functions. We used manual analysis to match these functions that BinDiff cannot handle.

	Predicted Match	Predicted No Match
Match	85	0
No Match	9284	21

Figure 26. The confusion matrix for all possible function comparisons.

All function types, regardless of whether they have constraints or not are considered in the second confusion matrix, in Figure 26. This confusion matrix shows how the symbolic matcher would perform only in relation to BinDiff’s capabilities and our manual analysis.

The false negatives here are the functions that cannot be matched with our matcher but are actually a match. Figure 27 shows the large percentage of functions that cannot be matched with our matcher.

In all, our matcher could perform comparisons that BinDiff could not. These cases of cross architecture matching of user defined functions are important. Figure 28 has a more detailed breakdown of how well our matcher did in comparison to BinDiff for these cases.

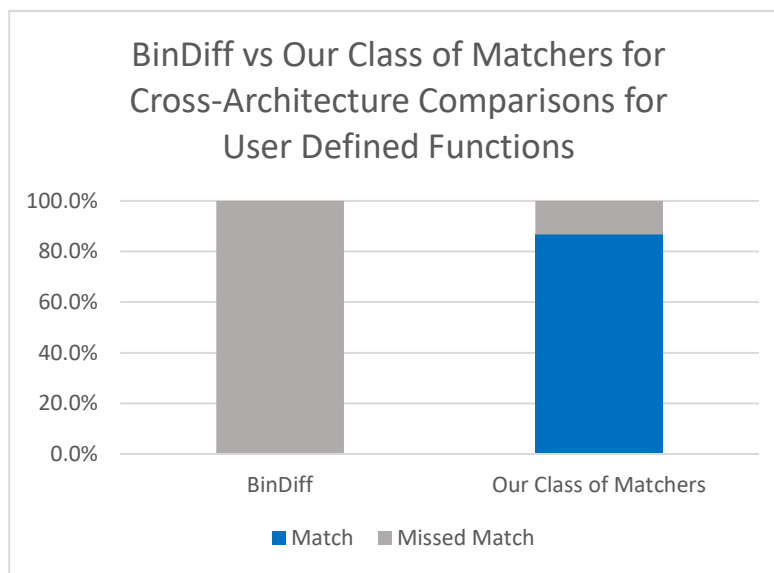


Figure 27. BinDiff in comparison to our class of matchers for cross architecture comparisons on functions with a change in code flow due to run time data: the ideal functions.

As we can see in Figure 27, BinDiff didn’t match any of the ideal functions when it came to cross architecture comparisons. BinDiff could match some of the imported functions during these comparisons but completely missed every user defined function that should have been matched during comparison. Our matcher performed well on these functions, only missing a small subset of them.

5.3 Comparing Symbolic Matchers

Comparison	Value Matches	Fuzzy Matches	Missed Matches
True 8.26 vs True 8.27	fwrite, main, version_etc, verstion_etc_va	fwrite, main, version_etc, verstion_etc_va, set_program_name	version_etc_arn
Echo 8.26 vs Echo 8.27	set_program_name, usage, version_etc, version_etc_va	fwrite , set_program_name, usage, version_etc, version_etc_va	version_etc_arn, main, strcmp, usage

Figure 28. Summary of some of the places where the fuzzy matcher performed better than the value matcher.

Between the fuzzy and value matcher the fuzzy matcher matched slightly more functions. Figure 28 demonstrates a few of the binaries where the fuzzy matcher matched more functions than the value matcher. When analyzing the functions that were matched with the fuzzy matcher but not matched with the value matcher; it is clear the fuzzy matcher is generating more accurate results. The fuzzy matcher was designed to give similarity values to constraint pairs that were not perfect matches, and this technique provides more room for flexibility. The fuzzy matcher matched 9% more functions than the value matcher. All the functions that were matched with the fuzzy matcher but not the value matcher were within the large test binaries. This allows us to make the conclusion that allowing a similarity to be granted even if there is no perfect match allows for better results.

The symbolic string matcher did not produce any significant results however it was an easy to implement algorithm that allowed us to test the capabilities of improving performance. Through string matching rather than value matching, the process would have been much faster however it was too inaccurate to provide meaningful results.

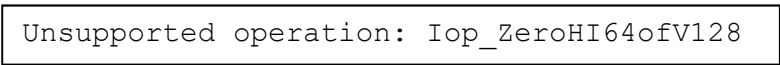
5.4 *angr*'s Symbolic Execution Engine

angr's symbolic execution engine overall does a mediocre job at supporting a wide variety of situations without a reverse engineer manually writing a specialized script for each situation. This could involve manual analysis to see what a binary is doing and writing a script that complements the specific traits of that binary. Although the engine is well built for specific use cases, such as capture the flags and other manual projects, writing a generic script to run symbolic execution on an abundance of binaries does not perform incredibly well. For small and simple test programs such as nested if statements or switch statements contained in a single function, *angr* performs as expected. However, once small programs use more complex C programming concepts, *angr* does not handle them well.

Errors in *angr*, during concretization of constraints, occur frequently. There are two types of errors that are handled in these extractors and matchers, the constraint being completely contained in one leaf and the inability to neatly simplify the constraint value. Both errors cause the symbolic matcher to perform poorly because they render the constraint unusable for matching. The completely one leaf error occurs 23.8% of the time and the inability to neatly simplify constraints error occurs 26.4% of the time. Combined, these cause 50.2% of constraints to be unusable during matching which accounts for a significant amount of inaccuracy.

As *angr* improves the extractor can easily be updated to include these improvements. The symbolic constraint concretization methods being improved would allow for 50% more of the constraints found during analysis to be usable. We estimate that this would allow our matcher to be at least 20% more accurate. Almost all the false negatives are due to constraint errors.

More places where *angr's* improvement would greatly affect our matchers are in the support for operations such as square root, modular arithmetic and more, especially across the different architectures. These errors are most often seen when running MIPS binaries through *angr*. Other places that would allow for improvement include loop handling and programs that modify directories. The *angr* error that is thrown when these unsupported operations are found looks like Figure 29.



```
Unsupported operation: Iop_ZeroHI64ofV128
```

Figure 29. Unsupported operation error in angr.

Some of the situations that are not fully supported for automated symbolic execution could be supported using scripts created perfectly for each program, however, that was not possible for this project. The idea was to create a script that can work on any situation, but the limitations of *angr* did not allow for this to be done perfectly. Other situations that are unusable with this project include: modifications to the working directory or operating system level methods.

Another situation that cannot be solved in general with symbolic execution is looping. Currently this issue is a research topic that has not been solved yet. Improving *angr's* static analysis is out of the scope of this project. Therefore, this project implements methods, described in system design extraction section, to get around the looping problems of symbolic execution in a fashion that doesn't render all the information unusable. Since *angr* is under active development, these issues could be fixed soon.

6. Conclusions

6.1 Matchers

Our main contribution was our attempt to solve the problem of comparing the similarity of functions using architecture agnostic symbolic execution data. We had good results in beating BinDiff when it came to user defined functions being compared cross architecture, which is shown in Figure 27. In these cases, BinDiff could only give a slight similarity score, generally under 20%, whereas we could match with over 70% similarity.

Both the fuzzy and the value symbolic constraint function matchers performed well for the given test sets, shown in Figure 28. For functions with constraints, they matched upwards of 87% of functions that were expected to be matched. If all functions are considered, including those without constraints, the results do not appear great, but these additional functions are outside the target function type for this technique.

6.2 Recommendations for Future Study and Improvement Measurement

Several areas of possible improvement have been discovered during this research. One such improvement that can be made is allowing for linked Abstract Syntax Trees to be used for matching. In conjunction with the Clone Detection research done with Abstract Syntax Trees [19], this will allow for graph matching and other techniques to be applied to the matchers. Graph matching with linked ASTs would help provide a more accurate match because we could then see the ordering of constraints that are added to the path rather than just having a list of constraints. This would also allow for detection of situations where a sub-tree of one tree is equal to the entirety of another tree, for example if one function was split up to be three functions in a different binary. This was out of the scope of this project because of the path explosion issues that would be encountered when linking the ASTs.

Another major improvement that can be made is within the preprocessing techniques, described in detail in Appendix C. Currently, our matcher only uses a length matching technique and an error checking technique. However, if there were a way to know what functions should be extracted and matched using symbolic matchers before running the script, upwards of twenty minutes of time could be saved for each of the large binaries. Knowing that the function matcher works especially well for user defined functions, using a preprocessing technique to identify this subset would be rewarding, though this could be difficult. With the addition of machine learning in areas like this, the symbolic matcher's speed could be improved significantly.

For other improvements, as *angr* improves there is room for improvement in the extractor and matcher. Improvements in Z3, the constraint solving engine, will allow for the errors in

constraints to be diminished and render them usable for matching which will significantly impact the matchers usability.

Future work could be done to implement some other symbolic function matchers. Some proposed ideas include data dependency matching, environment and state matching, and aided fuzzing or tainting, found in Appendices A and B. These matchers have potential; however, they may require architecture specific information and thus may lead to architecture specific matchers.

6.3 Lessons Learned

This project taught us many lessons. We started with nothing and now have a prototype symbolic function matcher. At the beginning of the project knowing nothing about symbolic execution or binary analysis, this project was daunting. However, through asking for help from experts in the field, we could find the right resources to learn more on our own. Had we never asked for help, we would have spent a significant amount of our time looking for the proper resources to learn reverse engineering.

Something else we learned is that there is a lot to learn from research papers. In the past reading research papers was a difficult task because often it is hard to fully grasp the concept that the author(s) are trying to get across. This comes from not having enough background knowledge on the subject before diving in. However, after the large amounts of time that we put into research on computer architectures, compilers, symbolic execution, reverse engineering, function matching techniques and more, research papers in these areas became easily understandable. Understanding other researchers worked helped us move ours forward with ideas gained from other's past experiences.

Overall, this project taught us many computer science topics, but more importantly, how to better set and meet expectations from advisors and mentors while working on a project.

References

- [1] zynamics, "Zynamics BinDiff," zynamics.com, [Online]. Available: <https://www.zynamics.com/bindiff.html>.
- [2] X. Xu, L. Chang, Q. Feng, H. Yin, L. Song and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," ACM CCS, Dallas, 2017.
- [3] N. Lageman, E. D. Kilmer, R. J. Walls and P. D. McDaniel, "BinDNN: Resilient Function Matching Using Deep Learning".
- [4] E. Brown, "MIPS Takes on ARM in the Internet of Things," Linux.com, 2014.
- [5] "Washington.edu," 6 October 2004. [Online]. Available: <https://courses.cs.washington.edu/courses/cse378/05au/lec378au04-01.pdf>. [Accessed 30 August 2017].
- [6] S. G. Shiva, "Advanced Computer Architectures," Taylor & Francis Group, Boca Raton, FL, 2006.
- [7] A. V. Aho, R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
- [8] "How to Build a GCC Cross-Compiler," Phreshing on Programming, 19 November 2014. [Online]. Available: <http://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/>. [Accessed 30 August 2017].
- [9] steeve, "Dockcross," GitHub.
- [10] "crosstool-NG," GitHub.
- [11] J. Ming, X. Dongpeng, Y. Jiang and D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," USENIX Security Symposium, Arlington, TX, Pennsylvania., 2017.
- [12] B. Engard, "The Power of Reverse Engineering," Software Guild, 21 November 2016. [Online]. Available: <https://www.thesoftwareguild.com/blog/what-is-reverse-engineering/>. [Accessed 28 August 2017].
- [13] K. N. Otto and K. L. Wood, "Product Design: Techniques in Reverse Engineering and New Product Development," Pearson Education Asia Limited and Tsinghua University Press, Asia, 2003.
- [14] "OWASP," 7 August 2017. [Online]. Available: https://www.owasp.org/index.php/Static_Code_Analysis. [Accessed 28 August 2017].
- [15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "(State of) The Art of War: Offensive Techniques in Binary Analysis," Santa

Barbara, 2016.

- [16] "Darpa Cyber Grand Challenge," Paris Hotel and Conference Center, Las Vegas, NV, 2016.
- [17] ForAllSecure, "Unleashing the Mayhem CRS," 2016.
- [18] K. Sen, D. Marinov and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," Illinois.
- [19] I. D. Baxter, A. Yahin, L. Moura, S. Marcelo and L. Bier, "Clone Detection Using Abstract Syntax Trees," ICSM, Bethesda, Maryland, 1998.
- [20] R. Saarsoo, "Matching Patterns in Syntax Trees," GitHub, 2 April 2016. [Online]. Available: <http://nene.github.io/2016/04/02/matches-ast>.
- [21] "LEDE Project: Linux Embedded Development Environment," 2017.
- [22] "GNU Operating System," Free Software Foundation, 25 April 2016. [Online]. Available: <http://www.gnu.org/software/coreutils/coreutils.html>. [Accessed 9 October 2017].
- [23] P. Godefroid, M. Y. Levin and D. Molnar, "acmqueue: SAGE: Whitebox Fuzzing for Security Testing," Microsoft.
- [24] x-ray, "Assignment Problem and Hungarian Algorithm," TopCoder, [Online]. Available: <https://www.topcoder.com/community/data-science/data-science-tutorials/assignment-problem-and-hungarian-algorithm/>.
- [25] T. Fahringer and B. Scholz, Advanced Symbolic Analysis for Compilers, Berlin Heidelberg: Springer-Verlag, 2003.
- [26] D. P. Bertsekas, "Auction Algorithms," Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA.
- [27] E. Eilam and E. Chikofsky, "Reversing: Secrets of Reverse Engineering," Wiley Publishing, Indianapolis, IN, 2005.
- [28] A. Kochkov, aoighost, A. Hartzheim, D. Tomaschik, DZ_ryyk, G. Rechistov, hdznrrd, J. Crowell, J. J. Dredd, j. K. Grandemanage, muzlightbeer, P. C. sghctoma, SkUaTeR, TDKPS and Thanat0s, "R2 "Book"," Gitbook, [Online]. Available: <https://radare.gitbooks.io/radare2book/>. [Accessed 10 May 2017].
- [29] "Clang vs Other Open Source Compilers," Clang, May 2007. [Online]. Available: <https://clang.llvm.org/>. [Accessed August 2017].
- [30] B. Hat, "Static Binary Analysis," 2015.
- [31] N. Nethercote, "Dynamic binary analysis and instrumentation," University of Cambridge Computer Laboratory, United Kingdom, 2004.

- [32] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software," Carnegie Mellon University, 2005.
- [33] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song, "A Symbolic Execution Framework for JavaScript," IEEE, Berkeley/Oakland, CA, 2010.
- [34] S. Bucur, V. Ureche, C. Zamfir and G. Candea, "Parallel Symbolic Execution for Automated Real-World Software Testing," Ecole Polytechnique Federale de Lausanne, Switzerland.
- [35] D. Atchley, "Datchley," Musings of a caffeinated coder, [Online]. Available: <http://www.datchley.name/recursion-tail-calls-and-trampolines/>.
- [36] "Malware," AV-Test, 21 September 2017. [Online]. Available: <https://www.av-test.org/en/statistics/malware/>. [Accessed 22 September 2017].
- [37] P. Klee, "KLEE LLVM Execution Engine," 2017.
- [38] "clang: a C language family frontend for LLVM," clang.llvm, [Online]. Available: <http://clang.llvm.org>.
- [39] "ObjDump," GNU.org, 3 May 2002. [Online]. Available: ftp://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_chapter/binutils_4.html.
- [40] "GDB: The GNU Project Debugger," Free Software Foundation, 26 September 2017. [Online]. Available: <https://www.gnu.org/software/gdb/>.

Appendix A

Control Flow Graphs in *angr* store an abundance of information alongside the regular CFG. They can store and show more information about the basic blocks such as the intermediate representation, the opcodes and information on the basic block such as size, successors and more. A picture of this is in Figure 30.

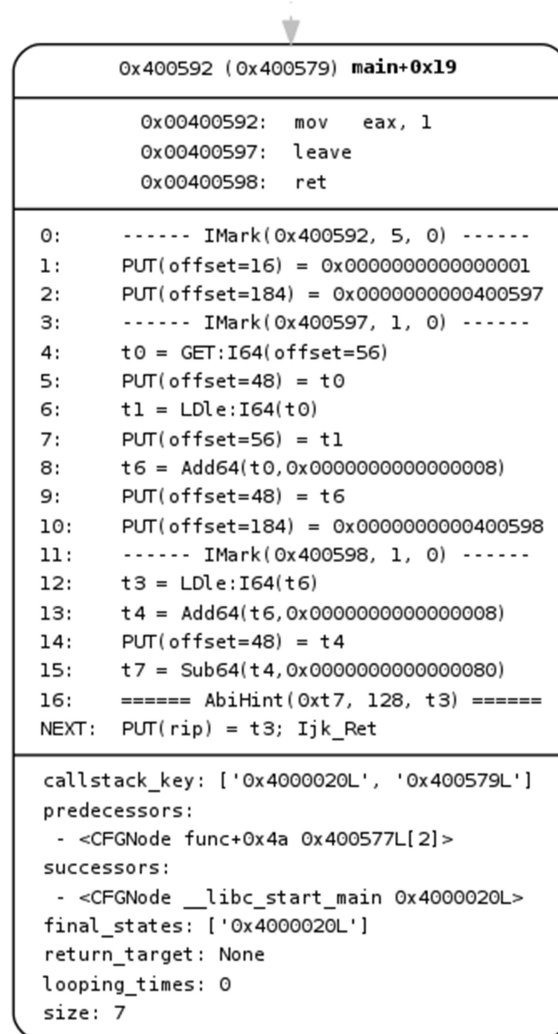


Figure 30. Here the Control Flow Graph shows all debugging information. From top to bottom it provides the basic block's address and name of the container function and offset, then it moves to the opcodes for the architecture, then the intermediate representation statement block, then the information on the return target and more.

Appendix B

angr also allows for many value set analysis techniques. One of which is analysis of a data flow graph. This form of analysis relies on expression trees [15] which is a tree of expressions that track operations on variables. Claripy, *angr*'s backend, tracks operations on expressions. Z3, a symbolic constraint solver, can then go in and solve for symbolic values or simplify concrete expressions [15]. These trees can be used to get different information about the program.

A data dependency graph is the attributes of the nodes in the parse tree that is depicted by a directed graph [7]. This graph is a record of the variables from the program and how they depend on each other. For *angr* this is stored as a DDG. The DDG tracks the variable and every location that caused the variable to change. For this analysis *angr* does not use the intermediate representation. Therefore, data dependency graphs are only available for the fully supported architectures. Looking at this information can tell you about the variables that are being used in the program and how often they are referenced and modified.

Appendix C

These algorithms and techniques are used in the matchers and extractors. Background on these techniques is critical to understanding what the function matchers are doing. LCS and Hungarian were found in pre-existing packages however the preprocessing technique and brute force matrix scoring methods was written specifically for this project.

Longest Common Subsequence

Longest Common Subsequence is used in constraint string matching. This method takes a string and finds the longest common subsequence using that string in comparison to another string. This technique provides a strong confidence between two strings because if the strings have a very long LCS then that denotes that they are practically the same string.

Hungarian Algorithm

	A	B	C
D	0	100	0
E	100	0	0
F	33	0	66

Figure 31. This is a sample matrix that would be used for matrix scoring using the Hungarian algorithm.

The Hungarian algorithm [24] provides a way of matrix scoring. The algorithm attempts to find a one to one match for each row and column, meaning each row and column are paired up based on the value shared between them. The algorithm is typically used to find the smallest sum of numbers. In this use case, it was modified to find the largest sum of numbers, because the best match for each constraint wants to be chosen. The Hungarian algorithm will take in a cost matrix and then will generate a graph with the match similarity as the weighted edges. Then the algorithm will determine the smallest possible sum of numbers. This final step is the most computationally expensive. Attempting to find the most ideal matches to provide the sum of numbers is a difficult and time-consuming process. An example matrix is shown in Figure 31. For this example, the Hungarian algorithm is used to find DB, EA, and FC as the matches with the values of 100, 100, and 66. The matrix scoring will use these values during the matching process. In these cases, where Hungarian would take too long, a brute force solution is applied in replacement.

Brute Force Algorithm

Using a brute force method of matrix scoring will simply take the highest value for each row and render that column unusable for any other row. This doesn't account for potential better matches in the future. This method is much faster however less accurate than Hungarian. It is only utilized when there are too many constraints or paths for Hungarian to be reasonably used without taking a significant amount of time and computation.

Preprocessing

Preprocessing is a technique used to avoid wasting computation time and power. This technique involves comparing lengths of lists before comparing them such as the list of constraints from a path or the list of paths from a function. This helps in avoiding the matchers trying to compare lists that are clearly not a match. Another way this technique saves time and memory is through noticing errors in *anqr*'s constraint solving methodology. Sometimes it cannot properly solve for a constraint into a simple comparison and will return unsatisfactory results. These are discussed more in depth in Symbolic Constraint Value Matcher's extraction section.

Appendix D

Test Data

Binary	Architecture	Size (Kilo Bytes)	Extraction Time (Seconds) **	Number of Functions *	Number of Ideal Functions *	Matched to...
Conditional_if_3deep	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Conditional_switch_2deep, Appendix E	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Conditional_if_3deep_3long, Appendix F	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Conditional_if_withloop, Appendix G	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Input_check_string, Appendix H	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Unsigned_integers, Appendix I	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
Dependencies, Appendix J	x86,ARM	8.6	<10	15	1	Itself cross architecture, every other binary we wrote
True 8.26	x86	129.2	62	42	10	True 8.27
True 8.27	x86	129.2	65	42	9	True 8.26
False 8.26	x86	129.2	54	42	9	False 8.27
False 8.27	x86	129.2	51	42	9	False 8.26
Yes 8.26	x86	143.6	126	60	22	Yes 8.27
Yes 8.27	X86	143.6	124	60	21	Yes 8.26

Echo 8.26	X86	138.9	95	48	14	Echo 8.27
Echo 8.27	X86	138.9	120	48	13	Echo 8.26
Sha256Sum 8.26	X86	219.9	1146	133	52	Sha256Sum 8.27
Sha256Sum 8.27	X86	219.8	1341	134	54	Sha256Sum 8.26
Printf 8.26	X86	238	2207	139	60	Printf 8.27
Printf 8.27	X86	238	2115	140	59	Printf 8.26
Autossh	MIPS	19.9	ERROR			
Autossh	ARM	21.6	1178	157	46	Autossh x86
Autossh	x86	21.6	1609	159	46	Autossh ARM
Mysql_Admin	MIPS	26.7	28	43	7	
Mysql_Admin	ARM	27.7	863	63	11	Mysql_Admin x86
Mysql_Admin	x86	27.7	256	40	8	Mysql_Admin ARM
TcpBridge	x86, ARM, MIPS	179.3	ERROR			
Bash	x86, ARM, MIPS	745.0	ERROR			
Git	x86, ARM, MIPS	1500	Ran out of memory on test machine			

*The number of ideal functions and regular functions includes *angr* repeated functions.

** If there was a halting error during extraction due to *angr*'s incapacibilities it is noted here.

Some of the binaries we wrote ourselves are not included in this table.

Appendix E

```
# This was used to test switch cases.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void func(int i){
    switch(i){
        case 0:
            switch(i+5){
                case 5:
                    printf("I was 0");
                default:
                    printf("I was not 0");
            }
        default:
            printf("I was not 0");
    }
}

int main(void) {
    int i = 1;
    func(i);

    return 1;
}
```


Appendix F

```
# This was used to test simple conditionals.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int func(int i){
    if (i<100){
        if (i<50){
            if (i<25){
                printf("less than 25");
            } else {
                printf("between 25 and 50");
            }
        } else {
            printf("between 50 and 100");
        }
    } else {
        printf("greater than 100");
    }
}

int main(void) {
    int i = 1;
    func(i);

    return 1;
}
```

Appendix G

This was used to test looping within the function we are symbolically executing.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
void func(int i){
    int in = 0;
    for(in = 0; in < 5; in++){
        i = i + in;
    }

    if(i==0){
        printf("0");
    } else if (i>5){
        printf(">5");
    }
}

int main(void) {
    int i = 1;
    func(i);

    return 1;
}
```

Appendix H

```
# This was used to test input from a command line.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static char password[] = "passw0rd";

int main(void) {
    register int j = 1000;
    register int i=0;
    char buf[20];
    printf("Enter the password: ");
    scanf("%s", buf);

    int match = strcmp(buf, password);
    if (match) {
        printf("Access denied.\n");
        return 0;
    }
    else {
        printf("Access granted!\n");
        return 1;
    }
    return 0;
}
```

Appendix I

```
# This was used to test unsigned integers.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
    signed int a = -5;
    unsigned int ua = a;
    unsigned int b = 2147483648;
    func(a, ua, b);

    return 1;
}

int func(a, ua, b){

    printf("a=%d ua=%u b=%u\n", a, ua, b);

    if(a > b){
        printf("True\n");
    } else{
        printf("False\n");
    }
}
```

Appendix J

```
# This was used to test using outside functions like strcmp.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int check_cond_string(char* str){
    int match = strcmp(str, "hello");
    if (match) {
        printf("hello!");
        return 1;
    } else {
        printf("goodbye!");
        return 0;
    }
}

int check_cond(int i){
    if (i==0){
        if(check_cond_string("hello")){
            return 1;
        }
    }
    return 0;
}

int main(void) {
    int i = 0;

    check_cond(i);
    return 1;
}
```