Binary Analysis and Symbolic Execution with angr

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
Electrical and Computer Engineering
On October 13, 2016

By: Eric Cheng

WPI Advisor: Professor Alexander Wyglinski
Sponsor: The MITRE Corporation

MQP – AW1 – M161

Abstract

This project involves the development of the binary analysis tool called angr. A tutorial for the binary analysis tool was created through annotated example scripts and documentation. This was accomplished by experimenting with and exploring the capabilities of angr keeping in mind the desires of a reverse engineer. The testing included learning about angr's compatibility with different computer architectures including x86, ARM, MIPS, and PowerPC as well as angr's support for both Linux binaries, Windows binaries, and binary blobs. The different use cases of angr were also tested and documented, specifically focusing on embedded systems. One of the main focuses of angr was its symbolic execution engine. A script was created to lower the learning curve of angr as well as give analysts more useful information extracted from the symbolic solver engine.

Alongside the tutorial and scripts, a written report and briefing was created and given to summarize the project. The results show that angr is a successful implementation of a binary analysis tool and symbolic execution engine. The drawbacks to angr are its limitations with Windows binary support as well as binaries that may be too large to solve. The script that was created allows for analysts to focus less on learning the intricacies of angr and more on actually performing reverse engineering tasks.

Acknowledgements

I would like to thank the following for all of their help and support on our project:

The MITRE Corporation for sponsoring this project

Peter Lucia – for his mentorship throughout the project and always making sure that I was having fun

Karen Lamb – for her brilliant ideas that always pointed me in the right direction when solving a problem

Peter Brown, Carlos Cheung, Brandon Hillan, Anthony Louie, and Corre Steele – for the abundance of moral support to ensure success for me during my time at MITRE

Members of the WPI Community

Professor Wyglinski – for his guidance and advising throughout the project

Table of Contents

List of Figures

Executive Summary

In Cyber Security, reverse engineering is an important cornerstone that is used for many purposes. Some purposes include analyzing the functionality of malware to develop solutions to fight that malware or finding vulnerabilities in certain software to create patches before the vulnerabilities are exploited. The main goals of reverse engineering a subject is to take its components and their relationships and create a representation at a higher level of abstraction that is more understandable and easy to work with. More specifically, software reverse engineering deals with analyzing binary code to find out more about what is contained in the binary. From a security standpoint, it is important to know exactly what a binary does and how it does it in order to ensure nothing malicious is happening when the binary code is run.

"angr" is a binary analysis framework developed by researchers from the Computer Security Lab at UC Santa Barbara. The tool provides binary analysis information such as finding functions and generating function call graphs and it also includes a symbolic solver engine capable of performing symbolic execution. angr can be a significant part of expanding the field of reverse engineering not only because of its symbolic execution engine, but also because of its strengths as a binary analysis tool. Its ability to find functions and generate function call graphs are important to reverse engineering.

Project Goals

The goal of this project was to create an effective solution to lower the learning curve necessary to use angr. When taking into consideration the interests of MITRE, there were three main objectives decided on for this project to achieve this goal:

1. Explore the symbolic execution capabilities of angr and document its intricacies
2. Explore the ability to use angr as a binary analysis tool
3. Create a platform for angr to make it more easily accessible by reverse engineers

The first objective was accomplished by testing angr as a symbolic execution engine. This meant creating example binaries and example test scripts to use angr to solve symbolic problems. These tests included solving for hardcoded passwords in binaries, extracting flags from a Linux command, solving the Carnegie Mellon University reverse engineering Bomb Lab, and performing "Live" symbolic execution. The second objective extended the testing of angr to use its binary analysis tools. The tools tested were the Control Glow Graph generators to find more information about the functions involved in a binary and BoyScout and GirlScout for binary blobs. BoyScout

was used for determining architecture and endianness of a binary blob and GirlScout was used for finding functions and calls between functions in a binary blob.

The third objective was accomplished by evaluating the important aspects of that an analyst would want in a reverse engineering tool. To implement this, a script was written that will allow analysts to see symbolic information at each step of execution. The script gives information on maximum and minimum values of a register and the constraints of the register.

Symbolic Execution Results

Password authenticator solver. This symbolic execution example dealt with creating a simple binary that had a hardcoded password and using angr to symbolically solve for that password. In the binary, the password was checked using the *strcmp* function. angr was used to create a path and *pathgroup* that could be used to step through the binary. When the execution of the binary branched after the *strcmp* call, angr was used to symbolically solve for the inputs that triggered each path. The source code and script used is shown in Appendix A.

Extracting flags from a Linux command. This example takes the *chmod* command from Linux and attempts to symbolically solve for the flags that can be passed into the command. The example utilizes creating symbolic bitvectors (BVS) in angr to symbolically represent the command line flag. This example also uses the hooking method in angr to take a complicated library command that may cause path explosion and hook it with a simpler implementation. Path explosion needs to be avoided because it creates unsolvable constraints for angr. The script for this example is shown in Appendix B

Solving CMU Bomb Lab Phase 2. This example uses angr to solve Phase 2 of Carnegie Mellon University's Bomb Lab. The bomb lab is well known in the reverse engineering field as a good introduction to learning reverse engineering techniques. This example deals with creating BVS and pushing and popping them onto the stack. It also requires examining the disassembly of the binary beforehand to determine the correct entry point for angr to being execution at in order to avoid path explosion. The script for this example is shown in Appendix C.

"Live" symbolic execution. The idea behind "Live" symbolic execution is to have firmware running either on a physical device or emulated system and then be able to pause the execution of that firmware at a specified point of interest and extract memory and code data. That data is then to be plugged into angr to have symbolic execution performed on it. This example uses the Quick Emulator (QEMU) and GNU Debugger (GDB) to emulate the firmware and extract data for angr. The source code and scripts for this example are shown in Appendix D.

Binary Analysis Results

Control Flow Graph (CFG) Generation. Control Flow Graph generation was done using two different angr methods: CFGFast and CFGAccurate. They perform similar functionality but are used under different circumstances. CFGFast is more useful for well-formed binaries where the header is included and functions boundaries are easily specified. CFGFast will make assumptions on the binary in order to more quickly generate a graph of the binary. CFGAccurate can be used for binaries that are not as well-formed, where functions boundaries are not easily recognized. CFGAccurate will explore through the entire binary to create a graph without making any assumptions. This method is much slower but will be more accurate than CFGFast. The CFGs generated by these two methods can be used for visual analyzation of the graphs as well as for the function manager that is created by generating the CFG. The function manager provides a list of the functions that are in the binary and information on where each function is called. Example scripts on CFGs are shown in Appendix E and Appendix F.

BoyScout and GirlScout. BoyScout and GirlScout are binary analysis tools used for analyzing binary blobs. A binary blob does not have information on its architecture or any information on what functions it might contain. BoyScout can be used for determining architecture and endianness of the binary blob. GirlScout can be used to find the functions of the blob. These tools provide useful information but are limited in their efficiency. The tools only work for smaller binaries. As the binaries grow larger, it takes an increasing amount of time and computational resource to perform these analyses. Example scripts for how to use BoyScout and GirlScout are shown in Appendix G.

angr Improvement Results

In order to make the symbolic information in angr more accessible for analysts and reverse engineers, a script was created to highlight the usefulness of the symbolic data that angr can provide. Important information on the symbolic values include:

1. Program counter
2. Maximum possible value
3. Minimum possible value
4. List of possible values
5. List of symbolic constraints

A simple binary was created and compiled for different architectures to test the functionality of the created script. The source code for the example binary is shown in List i. The idea behind the binary is two have three possible paths that the program can take, each with concrete values that constrain the paths. Thus, when solving the binary in angr, the output from the script can clearly show that the symbolic values and constraints are being returned properly.

```
int main(int x) {
    if(x < 3) {
        return x;
    } else if (x < 5) {
        return x;
    } else return x;
}
```

List i. Source code for binary used to test the angr improvement script.

The improvement script used can be found in Appendix H. An example of the output from stepping through the binary with the script and printing out symbolic register information is shown in List ii. The output starts by printing out the program counter so that the analyst running the script knows which path that program has taken. The second line gives the angr object that is representing the symbolic value. The third and fourth lines specify the maximum and minimum possible values that the symbolic register can take. In this case, the value must be greater than or equal to 3 or less than or equal to 4. The fifth line shows this by printing out the possible values that the symbolic register can be. Finally the last line give a formal symbolic constraints lists that tells all the constraints on the symbolic register.

```
--------------------------------------------------
Program counter: <BV32 0x8d3c>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4
Minimum possible value in reg: 3
Possible values in reg: (3L, 4)
State constraints: [<Bool !(symbolic_0_32 <=s 0x2)>, <Bool symbolic_0_32[31:3]
== 0x0>, <Bool symbolic_0_32[2:0] <= 4>, <Bool (symbolic_0_32 == 0x3) ||
(symbolic_0_32 == 0x4)>]
--------------------------------------------------
```

List ii. Output from improvement script

1. Introduction

1.1 Motivation

Reverse engineering is a process commonly used in the Cyber Security sector for many purposes, including analyzing how malware works or finding vulnerabilities in certain software. The main goals of reverse engineering a subject is to identify its components and their relationships as well as to create a representation of the subject at a higher level of abstraction [1]. Software reverse engineering deals with analyzing binary code to find out the binary's functionality. Binary code is the digital representation of text and data using the binary number system. Computer programs and functions will be compiled into binary code for a computer to interpret. From a security standpoint, it is important to know exactly what a binary does and how it does it to ensure nothing malicious is happening when the binary code is run.

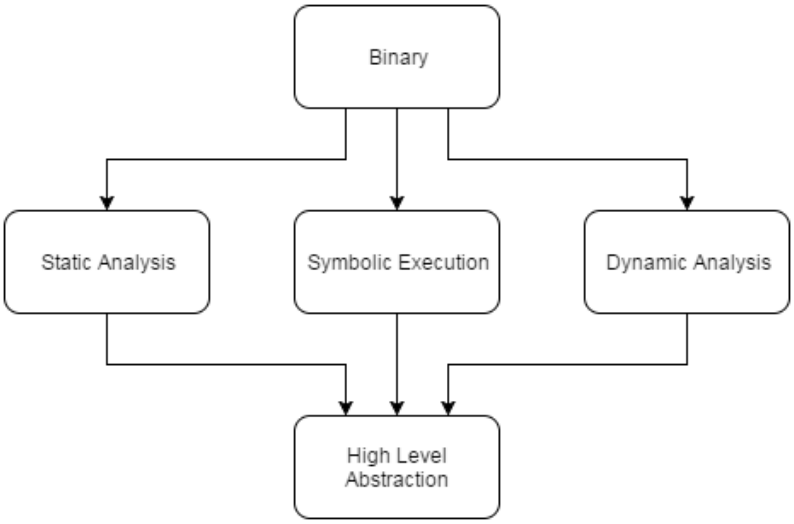


Figure 1. This is a general reverse engineering concept diagram. The second layer shows that there are multiple reverse engineering methods that fall under different categories. Symbolic execution is a method that falls between static analysis and dynamic analysis.

As shown in Figure 1, there are numerous ways to approach reverse engineering and binary analysis as well as different tools to aid in these approaches. This report focuses on symbolic execution; a form of binary analysis to determine what inputs cause each part of a program to execute. Symbolic execution is a useful tool for program testing because it can reveal information on every aspect of the program. It can also be used to find bugs or unwanted behavior as well as ensuring that a program will do what it is supposed to.

"angr" is a binary analysis framework developed by researchers from the Computer Security Lab at UC Santa Barbara [2]. The tool provides binary analysis information such as finding functions and generating function call graphs by using its own symbolic execution engine. The framework is based off of many well-tested tested binary analysis techniques [3].

angr can be utilized as a reverse engineering tool with its symbolic execution engine as one of the main points of interest. Often, there are pieces of firmware with binary code and no source code available that needs to be reverse engineered. In addition, when running a piece of firmware there are certain parts of the binary that require input in order to executed, but these inputs are not available at analysis time (due to missing hardware or required user inputs). This calls for the ability to perform "live" symbolic execution.

The idea behind this is to be able to run firmware either on a device or in an emulator such as Quick Emulator (QEMU) and be able to pause execution when a certain point is reached. Once this point is reached, information on the state of the binary (memory data, code data, register data, etc.) must be able to be dumped out. This extracted state will then be plugged into angr to have symbolic execution performed on it.

angr can be a significant part of expanding the field of reverse engineering not only because of its symbolic execution engine, but also because of its strengths as a binary analysis tool. Its ability to find functions and generate function call graphs are important to reverse engineering. Though successful at what it does, angr is not the only program that has been utilized in symbolic execution.

1.2 Current State-of-the-Art

Symbolic execution is a rapidly growing reverse engineering tool. There are many different tools that have been used for symbolic execution with various results. Figure 2 depicts the general process that goes into symbolic execution.

A tool called CUTE is a concolic unit testing engine for C and Java that tests sequential C programs and concurrent Java programs. Concolic execution uses a mixture of concrete and symbolic execution with the goal of avoiding redundant test cases [4]. There are three general steps to CUTE's concolic testing. The first is to create a logical input map to generate concrete input graphs for pointer values and primitive values. The second is to run the code on the concrete inputs to collect constraints. Finally, CUTE solves the constraint system to obtain a new logical input map. These steps are repeated until a complete input map is created [5].

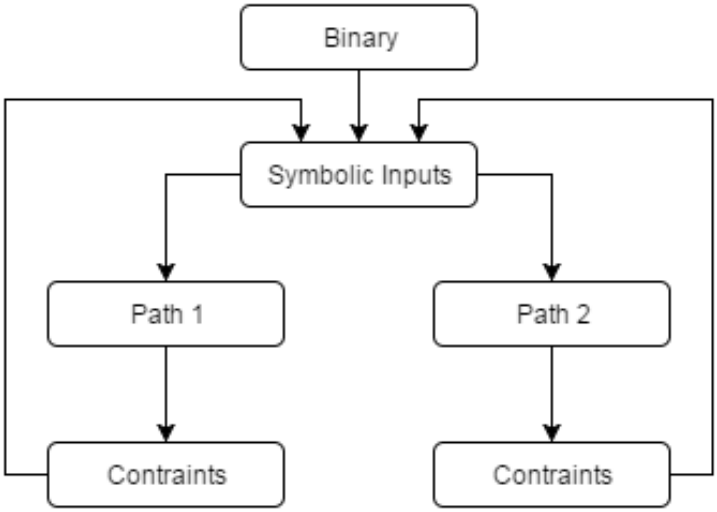


Figure 2. This is a flow chart for general symbolic execution. The execution starts by creating symbolic inputs from the original binary. From the inputs, paths can be deduced and each path will carry certain constraints. The inputs from these constraints will then be executed upon until all paths are found.

The use case behind CUTE is different from the use case behind angr. CUTE is a software testing tool to improve general reliability and safety of software. In this scenario, the person using the tool already has all of the source code and knows the functionality of it. The tester would be using the tool to ensure that there are no unexpected inputs that would cause unexpected results.

Another symbolic execution engine is Selective Symbolic Execution, or S2E. The basis behind this engine is described in its name. S2E will only perform symbolic execution on selective parts of code that are of interest to the analyst. The goal behind S2E's symbolic execution is to avoid the state of path explosion, where a symbolic execution tree grows to an unmanageable size [6]. S2E focuses on the conversions between symbolic memory and concrete memory, as well as preserving correctness between the states. The limitation behind S2E is that it is currently only written to solve Linux binaries.

A more complete binary analysis tool is BitBlaze. BitBlaze consists of three components: Vine for static analysis, TEMU for dynamic analysis, and Rudder for symbolic execution. Vine is used to translate assembly code an intermediate language that will give information on control flow and data flow. TEMU will attempt to execute code to extract OS-level semantics and user-defined data. Finally, Rudder, shown in Figure 3, utilizes the information gained form Vine and TEMU to perform both concrete and symbolic execution at the binary level [7].

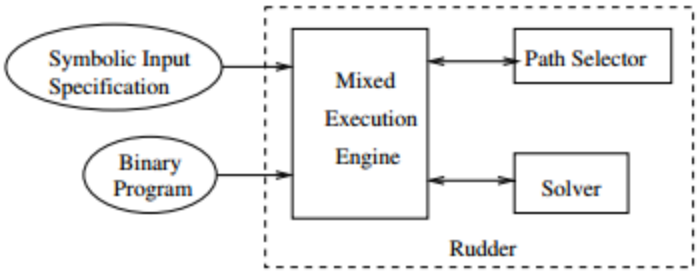


Figure 3. This is the architecture of Rudder, the symbolic execution engine of BitBlaze. It starts by inputting symbolic values into the engine and then determines possible paths that can be traversed based on constraints found [7].

A binary analysis tool that is more similar to angr is called Mayhem. Mayhem is a closed source system for finding exploitable bugs in a binary. The key factors behind Mayhem are its uses of symbolic execution and index-based memory modeling. Mayhem has four main principles [8]:

1. The system should be scalable and handle larger binaries
2. The system should be efficient and not redundant
3. The system should preserve all work
4. The system should be able to interpret symbolic memory with user input

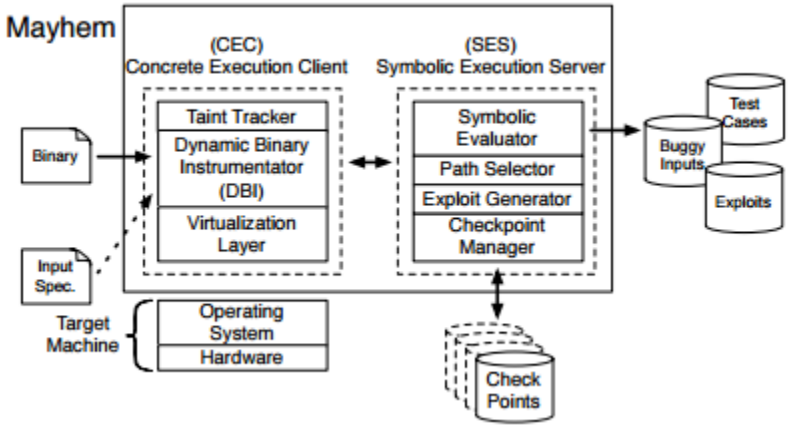


Figure 4. This figure is the architecture overview of Mayhem. Mayhem includes a concrete execution client as well as a symbolic execution server [8].

Through these principles, Mayhem has developed what they call “hybrid symbolic execution”. Hybrid symbolic execution efficiently swaps out symbolic execution engines. This means that when execution is using large amounts of memory, the execution will be paused and dealt with outside of the engine. The resulting state will be restored to the symbolic execution engine where it will now use less memory [8].

1.3 Contributions of MQP

When taking into consideration the interests of MITRE, there were three main objectives decided on for this project:

1. Explore the symbolic execution capabilities of angr and document its intricacies
2. Explore the ability to use angr as a binary analysis tool
3. Create a platform for angr to make it more easily accessible by reverse engineers

The first objective was accomplished by performing numerous tests on angr with different motives in mind. These tests included solving for hardcoded passwords in binaries, extracting flags from a Linux command, solving the Carnegie Mellon University reverse engineering Bomb Lab, and performing "Live" symbolic execution. The second objective looked at the way that angr could be used for binary analysis. This included experimenting with the Control Flow Graph generators in angr as well as its BoyScout and GirlScout modules.

The third objective was accomplished by determining what exactly reverse engineers want when it comes to a binary analysis tool. To implement this, a script was written that will allow reverse engineers to see symbolic information at each step of execution. The script gives information on maximum and minimum values of a register as well as all the constraints of the register.

1.4 Report Organization

This rest of this report covers the information gained from this research and testing to achieve the stated project obejctives. Chapter 2 is a literature review on symbolic execution, computer architectures, and reading disassembly code. Chapter 3 covers how angr can be used for symbolic execution and all of the intricacies that must be considered in the form of a tutorial-styled write-up. Chapter 4 discusses how angr can be used as a binary analysis tool. Finally, Chapter 5 discusses the layout of the implementation script. Chapter 6 discusses the results of the scripts for binaries compiled for different architectures.

2. Symbolic Execution and Computer Architectures

There are a few things that are important to understand before diving into how angr works. The first is symbolic execution and the second is computer architectures.

2.1 Symbolic Execution

Symbolic execution is a means of analyzing a program where instead of supplying normal concrete inputs to the program, symbols representing arbitrary values are supplied instead [9]. Symbolic execution first started out as a way to check sequential programs with fixed integers. However, more recent research has paved the way for programs to perform more complex symbolic execution [10].

While the definition of symbolic execution may seem simple, the implementation is far from it. For a given programming language, there is a set of "execution semantics" that defines the possible symbolic data objects that can be provided as inputs to a program. These are the only symbolic values that can belong in a program. When the evaluation reaches a branch statement, the set of possible symbolic values for the inputs are narrowed in response to the parameters of the branch.

The program information during symbolic execution is all kept in the "state" of the program. The information includes the constraints on each symbolic variable. During a branch statement, there are two possible outcomes. The branch execution can either be "nonforking", where all possible constraints take one path or all possible constraints take another path, or the execution could branch, where some constraints lead down one path and other constraints lead down the other path [9].

In this scenario, execution is forked into two parallel executions. The important part to note is that both paths will have the same state that they shared before the branch, but will then execute independently of each other after the branch.

A simple example of symbolic execution using just integers is shown in List 1.

```
int main(int x, int y, int z) {          (1)
      a = x + 5;                         (2)
      b = y + 5;                         (3)
      c = a + b + z;                     (4)
      return c;                          (5)
}
```

List 1. Symbolic execution code snippet example. The inputs x, y, and z can be represented as symbolic values, therefore making a, b, and c defined as symbolic values as well

The symbolic execution of this program is represented in Table 1 where we explore the symbolic value of each variable after each line of code.

Table 1: Simple symbolic execution represented by a table. The inputs x, y, and z are defined first and a, b, and c are defined based off of the symbolic values that there input variables were assigned as.

After Statement	x	y	z	a	b	c
1	sym_1	sym_2	sym_3	-	-	-
2	sym_1	sym_2	sym_3	$sym_1 + 5$	-	-
3	sym_1	sym_2	sym_3	$sym_1 + 5$	$sym_2 + 5$	-
4	sym_1	sym_2	sym_3	$sym_1 + 5$	$sym_2 + 5$	$sym_1 + sym_2 + sym_3 + 10$
5	sym_1	sym_2	sym_3	$sym_1 + 5$	$sym_2 + 5$	$sym_1 + sym_2 + sym_3 + 10$

From Table 1, it can be gathered that the program will return the sum of any possible symbolic value for each input and the number 10. In this case, there are no concrete constraints on the returned variable. Thus the value could be any number from zero to however large an integer is defined in the architecture.

2.2 Computer Architectures: CISC and RISC

There are two main types of computer architectures that will be covered in this project: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). An example of CISC is the x86 architecture [11]. Examples of RISC are the ARM, the MIPS, and the PowerPC architectures. The differences between CISC and RISC is their behavior in how they execute instructions.

The CISC architecture aims to complete a task in as few assembly instructions as possible. This is done with an emphasis on building rich instruction sets with an emphasis on complex hardware. The hardware in a CISC system must have the ability to interpret several series of operations at a given time [12]. The CISC architecture uses microcode, which is the interpreter between the hardware of a computer and the architectural level of a computer. Characteristics of the CISC instruction set include:

- 2-operand format: instructions have a source and destination
- Multiple addressing modes for memory
- Instructions that take multiple clock cycles to execute

An example of a CISC instruction would be one that performs multiple jobs at once. There could be an instruction called “MULT” that will load two values into registers, multiply them, and then store the result in another register.

There are drawbacks to CISC as well. These include the increased complexity of an instruction set as each new chip hardware generation must maintain the same properties of the previous generation. Since CISC attempts to execute code in as few instructions as possible, the compensation is that the instructions will take more clock cycles to execute [13]. The complexity of CISC can be shown in the flow diagram in Figure 5. CISC starts with the machine instructions, converts them to microcode, then converts to microinstructions, which is then executed.

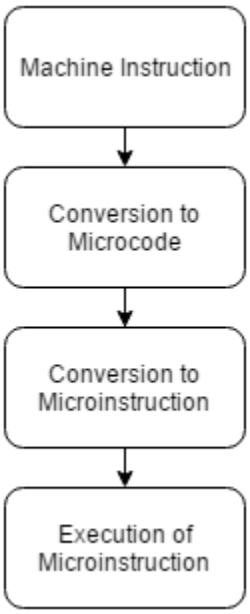


Figure 5. CISC flow diagram. The CISC architecture takes machine instruction and converts it to microcode. The microcode is then converted to microinstructions. The microinstructions are then executed.

The RISC instruction set focuses on using more simple instructions that can each be executed in just one clock cycle. From the CISC example shown above, the RISC equivalent would split up the “MULT” instruction into three different instructions: “LOAD”, “MULT”, and “STORE”. The RISC instruction set is more emphasized on software. Separating commands into multiple instructions allow for pipelining and more efficient use of hardware [12]. Characteristics of the RISC instruction set include:

- more optimized register usage with 32 or more registers
- simplified compiler design by using general purpose registers
- use registers instead of the stack to pass arguments and hold variables
- reduced number of instructions (compared with CISC)

The main drawback to RISC is increased instruction size from having more instructions. Since RISC processors depend on compilers to explicitly emit each instruction, the compilers will determine the performance of the program. The flow diagram of RISC is shown in Figure 6. It is much less complex than CISC.

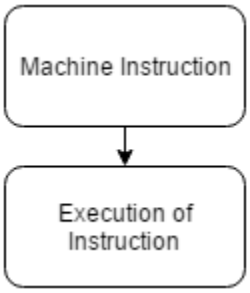


Figure 6. RISC flow diagram. The machine instruction is executed immediately. There are no conversions that need to be made.

2.3 Computer Architecture: x86

This section will give a brief overview on how to interpret x86 assembly code. The x86 architecture has eight general purpose registers, each 32-bits in size which can be seen in List 2 [14]:

```
EAX – Extended Accumulator Register
EBX – Extended Base Register
ECD – Extended Counter Register
EDX – Extended Data Register
ESI – Extended Source Index
EDI – Extended Destination index
EBP – Extended Base Pointer
ESP – Extended Stack Pointer
```

List 2. General purpose registers for x86 architecture

The x86 architecture uses the stack as the part of memory to store local variables and function arguments. The stack has a Last In First Out (LIFO) structure. The ESP and EBP registers are associated with the stack, where the ESP (stack pointer) always points to the top of the stack. The EBP (base pointer) is the reference on the stack that never changes. The ESP register is the pointer to the top of the stack when the function is first called [15].

Arithmetic operations follow the same general syntax. For addition and subtraction, the syntax is `ADD/SUB dest, src`. Where `ADD/SUB` is the operation and `dest` is the destination register and `src` is the source register. Example instructions are shown in List 3 [16].

```
add eax, 7
sub eax, 4
```

List 3. Example add/sub x86 instructions

There are three possible ways to perform multiplication. There is multiplication of just the eax register with a value, multiplication of two values stored into a register, and multiplication of a register with a value stored into that same register. List 4 shows each of these options [16]. The first line in List 4 only has 1 argument. This means that the eax register is taken and multiplied by that value. The second line in List 4 shows that the multiplication of the two end arguments are stored into the first argument register. The last line multiplies the value in the first argument register and the value in the second argument and stores the result in the first argument register.

```
mul 5
mul edx, 3, 6
mul ecx, 8
```

List 4. Example of multiplication in x86. The first line is multiplication of the eax register with a value. The second line is multiplication of two values stored in a register. The third line is the multiplication of a register and a value.

For division the instructions are split up into multiple steps using the eax and ecx register. The eax register stores the dividend while the ecx register stores the divisor. The resulting answer is stored in eax with the remainder in edx. An example is shown in List 5 [16]. The first line in List 5 stores 16 in register eax. The second line stores 5 in ecx. The last instruction divides 16 by 5. It then stores the result, 3, in eax. The remainder, 1, is stored in edx.

```
mov eax, 16
mov ecx, 5
div ecx
```

List 5. Example of division in x86. The values are first stored in the registers using the mov operation. The resulting answer, 3, is stored in eax while the remainder, 1, is stored in edx.

Branching is an important aspect to understand in any assembly language. x86 uses jumps to denote branches. A few jump commands are shown in List 6 [16]. These command include a jump if values are equal to each other, not equal to each other, or greater than or less than zero.

```
JMP   jump
JE    jump if equal to
JLE   jump if less than or equal to
JNZ   jump if not zero
JZ    jump if zero
JBE   jump if below or equal to
JGE   jump if greater than or equal to
```

List 6. A selection of common jump commands used in x86

2.4 Computer Architectures: ARM

ARM has 37 registers each of size 32-bits. The general purpose registers are labeled r0-r12. There is a stack pointer (SP) register, link register (LR), and program counter (PC) register [17]. SP is also known as r13, LR is known as r14, and PC is known as r15.

The syntax for addition, subtraction, multiplication, and division operations are all similar for Intel syntax, with the operation coming first, the destination coming second, and the two values being operated on coming last. AT&T syntax is the other way around, with the destination address coming last. However, these examples will all use the Intel syntax. Examples are shown in List 7 [18]. The first line in List 7 shows the addition of the value in register r1 and 7 stored into the r0 register. The second line subtracts 8 from the value in r3 and stores the result in the r0 register. The third line multiplies 5 and 6 and stores the answer in the r0 register. The final line divides 12 by 4 and stores the result in the r0 register.

```
ADD r0, r1, 7
SUB r0, r3, 8
MUL r0, 5, 6
SDIV r0, 12, 4
```

List 7. Arithmetic examples for ARM

ARM uses branches to transition code. The syntax is similar to the jumping as shown in the x86 architecture. Example ARM branch operations are shown in List 8 [18]. There are branches for if a value is greater than or equal to another value or less than or equal to another value.

```
BGE   Branch if greater than or equal to
BLE   Branch if less than or equal to
BLT   Branch if less than
BNE   Branch if not equal to zero
```

List 8. Branch examples for ARM

2.5 Computer Architectures: MIPS

In MIPS, there are 32 registers of size 32-bits that are for use with integer and logic instructions. They are labeled $0 - $31. Table 2 shows this usage of these 32 registers. There are registers for a dedicated zero value, return values, temporary data, saved registers, etc.

Table 2. MIPS registers are their purposes [19].

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0 - $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions – not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel
$28	$gp	Global Area Pointer
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

Arithmetic syntax for MIPS is similar to that of ARM. Examples for addition, subtraction, multiplication, and division are shown in List 9. The results for division and multiplication are stored in the special register $LO [20].

```
add $t0, $s1, $s2
sub $s4, $s5, $s0
mult $s3, %t4
div $s2, $t0
```

List 9. Arithmetic operations in MIPS. For addition, s1 + s2 is stored in t0. Similarly, for subtraction, s5-s0 is stored in s4. The result of the multiplication and division is stored in the special register $LO.

In MIPS, there are branches to move around code, similar to ARM. List 10 shows some common branch instructions. MIPS is different from the other referenced architectures in that it utilizes the branch delay slot. The branch delay slot means that after a branch operation is reached, the instruction after the branch will be executed before the branch itself is taken. The BEQ instruction will compare two values. If the values are equal to each other, then the branch will be

taken. Similarly, BNE will compare two values and if the values are not equal, the branch will be taken. BGEZ will compare a register to 0. If the value is greater than zero, the branch will be taken. The opposite is true for BLTZ, where the branch will be taken if the value being compared is less than zero.

```
BEQ   Branch if equal to
BNE   Branch if not equal
BGEZ  Branch if greater than or equal to zero
BLTZ  Branch if less than zero
```

List 10. Branch examples for ARM. The first two instructions are comparing two registers to each other. The second two operations compare a register to the value zero.

2.6 Computer Architectures: PowerPC

PowerPC has 32 general purpose 32-bit registers labeled r0 – r31. The r0 register is generally used to hold the old link register. The r1 register is the stack pointer and the r2 register is the table of contents pointer. Registers r3-r10 are generally used to send arguments through a function. The rest of the registers are global registers [21]. Arithmetic operations for PowerPC are shown in List 11. The first line in List 11 shows the addition of the values in r3 and r5 with the result stored in r0. The second line subtracts the value in r10 from the value in r4 and stores the answer in register r6. The third line multiplies the r2 and r8 registers and stores the results in the r3 register. The last line divides the r4 register by the r6 register and stores the result in the r1 register.

```
add   r0, r3, r5
subf  r6, r4, r10
mulhw r3, r2, r8
divw  r1, r4, r6
```

List 11. Arithmetic operations for PowerPC. The general syntax follows that the first item is the destination register and the second and third items are the operands.

Branching in PowerPC is similar to those in ARM and MIPS. There are conditional branches as well as branches to the link registers as shown in List 12. The BLT command compares will branch if the desired value being compared is less than zero. The BGT command bill branch if the desired value is greater than zero. The BEQ will branch if the value is equal to zero. Finally, the BLR command will cause a branch two whatever value is stored in the link register.

```
BLT   Branch if less than
BGT   Branch if greater than
BEQ   Branch if equal to
BLR   Branch to link register
```

List 12. Branch instructions in PowerPC

2.7 Chapter Summary

Before diving into the intricacies of angr and how to use angr, a basic knowledge of symbolic execution and computer architectures must be understood. Symbolic execution is a means of analyzing a program where instead of supplying normal concrete inputs to the program, symbols representing arbitrary values are supplied instead. The goal of symbolic execution is to find all possible paths that a binary can take and what inputs will cause those paths to run, in order to find out more information on the binary.

There are two main types of computer architectures: CISC and RISC. CISC stands for Complex Instruction Set Computing and focuses on completing a task in as few assembly instructions as possible. This is done with an emphasis on building rich instruction sets with an emphasis on complex hardware. RISC stands for Reduced Instruction Set Computer and focuses on using more simple instructions that can each be executed in just one clock cycle, thus being more emphasized on software. An example of CISC is the x86 architecture, while an example of RISC is ARM or MIPS.

3. Symbolic Execution with angr

This section will discuss the many uses of angr. There are multiple examples with references to example scripts in the appendices. The topics covered include:

1. Password authenticator solver
2. Extracting flags from a Linux command
3. Solving CMU Bomb Lab Phase 2
4. “Live” symbolic execution
5. Different architectures and statically versus dynamically linked binaries

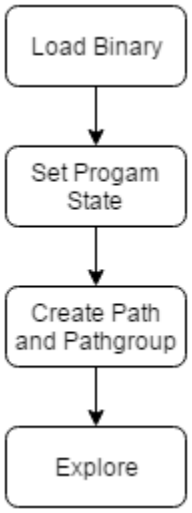


Figure 7. General flow chart for symbolic execution with angr. Begin by loading the binary with parameters, set the program state to begin at, then create the path and pathgroup to explore on

Symbolic execution with angr can be generally summarized by the flow chart shown in Figure 7. The four basic steps are loading the binary, setting the program state, creating a *path* and *pathgroup*, and then exploring through the binary. The first step, loading the binary, creates a representation of the binary for angr to interpret. angr will encompass the binary in its own project class. The class contains information such as the entry point of the binary as well as the architecture of the binary. When loading a binary, there are many load options that are present. The binary can be loaded with custom libraries or some libraries can be purposely excluded. The specific backend of the binary can be specified as well as the custom base address.

The next step, setting the program state, creates the symbolic state tracker in angr. Each path that is created in angr will have its own program state. The state contains all of the data about that path such as register values and values in memory. The state will also hold symbolic values and symbolic constraints. Creating a *path* and *pathgroup* allow angr to have a stash of paths to

step through. A *pathgroup* is a collection of *paths* organized into stashes. When exploring through the *pathgroup*, each path will be pruned into an appropriate stash. The three main stashes are *active, deadended, errored*. An *active* path has further code to be executed and stepped through. A *deadended* path has exited the program normally and therefore will not be stepped through. An *errored* path exited the program with an error where the program was expecting something that it could not access, for example some unloaded memory regions.

3.1 Password authenticator solver

This is an example of angr solving a simple password authenticator. The binary being used is a simple dynamically linked x86 binary with its source code known, which contains a hard-coded password and uses a `strcmp` to check the input password's validity. The source code for the binary used in this example can be found in Appendix A. The source code shows that the hardcoded password is "passw0rd". Interaction with angr is done through Python scripting. The final Python script for this example can be found in Appendix A.

The first step is to load the target binary into angr by using the project loader. The project will be loaded using CLE, a binary loader that loads associated libraries, resolves imports, and provide an abstraction of process memory. This will also give angr information on architecture, entry points, and other properties. There are many parameters that can be passed through when loading a project such as defining a custom architecture or options for loading libraries [22].

Next, the initial program state must be set. Setting the state will return a *SimState* object that allows you to access certain information about the binary at that state, including data in different memory regions as well as register data. The entry state can be given a custom base address as well as argument parameters [23]. Once the state is set, a *path* and *pathgroup* must be created for angr to step through. *Paths* are a sequence of basic blocks that have been executed since the start of execution. A basic block is a straight sequence of code without any branches or jumps. *Pathgroups* are a bunch of paths organized into "stashes" that have different properties [24].

Now, angr can step through all possible *paths* in the *pathgroup* using the explore method. The main stashes that angr separates paths into are *Active, Errored, Deadended,* and *Found/Avoided*. *Active* paths are ones that can still have instructions to be stepped through. *Errored* paths are paths that cause a Python exception in angr which means the binary did not exit properly. A *Deadended* path is a path with no more valid instructions and thus cannot be stepped through anymore. *Found/Avoided* paths are paths that include the address specified by the find/avoid parameter (respectively) in the `explore()` method.

In this scenario, the find address specified above is where the print of "Access granted" is in our compiled binary while the avoid address is at the print of "Access denied". These addresses were found by performing manual binary analysis.

Once the *pathgroup* finished exploring we can access the results by pulling information from the state of the found *path*. In this example, there is a call to `scanf` which means that a call to `dumps(0)` will return the concrete contents for the file descriptor 0, which is `stdin`. The trace of the path can also be accessed to see which basic blocks that path had traversed through.

3.2 Extracting flags from a Linux Command

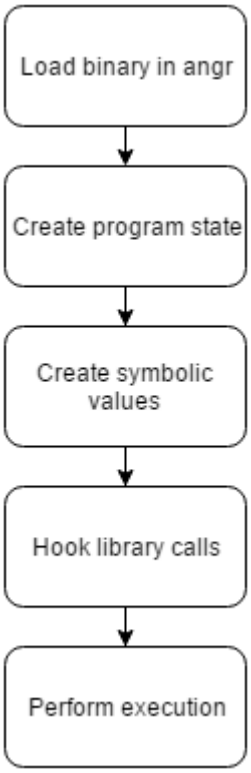


Figure 8. Flow diagram or extracting flags from a Linux command. The diagram follows the same generic structure as shown in Figure 7, with a few extra steps. These steps include creating symbolic values, loading those values into registers, and hooking library calls to avoid having path explosion during execution.

The following is an example of extracting command line flags for a Linux binary, specifically the `chmod` binary. This example deals with loading specific register data and setting registers to have symbolic values. The example script can be found in Appendix B. A flow diagram for the process is shown in Figure 8.

In Linux, `chmod` can be run as `chmod [options] <filename>` where options are the possible flags that we want to symbolically solve for. The command line arguments are represented

in an `argv` list where `argv[0]` would be the command and `argv[1]` would be the option represented symbolically. To represent `argv[1]` symbolically we use angr's solver engine called Claripy [25]. Claripy defines values in bitvectors. A symbolic bitvector (BVS) is what we will use to represent a symbolic value.

When creating the program state, the `argv` list with the symbolic bitvector should be included as an argument, which is done by setting the `args` parameter in the `entry_state` method. The start address must also be determined when creating the program state. Analyzing the disassembly of the `chmod` command shows that the function `getopt_long` is where command line values begin to be processed, so the address of that function is where execution should begin.

The function `getopt_long` parses the command line for arguments from argv and returns them as strings. However, since this is a library function that will cause path explosion if it is stepped through in angr, the function call must be hooked to something simpler with similar functionality. angr has the capability to hook a function at a specific address to call another function instead. In the example `chmod` binary, the `getopt_long` function is located at 0x401ac0, and can replaced with a written implementation called `hook_getopt` [26].

Making a less complex implementation of the `getopt_long` function is simple. The function must return a value to the `eax` return register. The value that the function should return is the same symbolic value that was created earlier and stored in `argv[1]`.

Using the program state, constraints can be added to `argv[1]` so that it can only return printable characters. In ASCII, the printable characters lie between ' ' (Hex 0x20) and '~' (Hex 0x7E).

The *path* and *pathgroup* can be created now, and stepped through. As angr steps through the program, it will check each state to see if it can or cannot be satisfied. In angr, an unsatisfiable or unconstrained path is when there are more than 257 ways to solve a symbolic value in that path. LAZY_SOLVES will automatically assume all states can be satisfied and then later on solve the states when absolutely necessary. Turning LAZY_SOLVES off will make it so that states that cannot be satisfied will be automatically avoided.

angr will step through all possible paths in the *pathgroup* using the `explore()` method. In this case, no parameters need to be passed into the `explore()` method so that it will step to completion. In the *Deadended* paths, paths that path occur when the program has as normal exit and has no more instructions that can be stepped, the constraints on `argv[1]` can be solved to return the possible characters that cause each path to run.

3.3 Solving CMU Bomb Lab Phase 2

The CMU bomb lab is a well-known lab for learning the basics of reverse engineering [27, 28]. This example uses the 32-bit version of the binary. This example will deal with solving Phase

2 of the lab. The idea of the lab is that for each phase, there is a certain input that must be provided in order to defuse the bomb and move onto the next phase. If the provided input is incorrect, the bomb will "explode". The lab can be solved by using a combination of analyzing the disassembly and utilizing angr. Phase 2 deals with storing and loading values to and from registers. The example script can be found in Appendix C

Looking at the disassembly of Phase 2 shows there is a function call to `read_six_numbers`. Analyzing `read_six_numbers` shows that it is pushing six values onto the stack and calling `sscanf` to read those values. The function will then return and begin its algorithm to determine whether the inputs are valid by inspecting the stack.

To solve this phase using angr, execution should be set to begin after the call to `read_six_numbers`. Symbolic values should be loaded onto the stack to simulate the initialization phase and call to `read_six_numbers` that are being skipped.

The binary can be loaded, custom entry state added, and path and *pathgroup* created to be explored on. The next step is to load symbolic values onto the stack. In x86 the base pointer for the stack frame is represented by the `ebp` register. The `read_six_numbers` function pushes a value onto the stack starting at 0x18 and decreasing by 0x4 for the following five values. This means that six symbolic values must be stored. These are represented by Claripy's BVS.

Next, exploration can begin on the *pathgroup*. The avoid parameter is set to ensure the `explode_bomb` function is not reached. Once the exploration finishes, the *pathgroup* will show one *deadended* path and six *avoided* paths. This makes sense because there is a loop in Phase 2 that checks the validity of each of the six inputted numbers which will result in six possible paths that explode the bomb and one possible path with six correct inputs that defuses the bomb.

For the *deadended* state, the solver engine can solve for each integer at the given addresses on the stack. The values will be represented in little endian so we must set the endianness parameter in the load function. The result can be appended to a list and printed out. The six values we end up with are `1, 2, 6, 24, 120, 720`. The pattern behind these values is that each number must be *n*x*, where *n* is the index and *x* is the previous value.

3.4 "Live" symbolic execution

The following is an example of performing "live" symbolic execution. This means taking a binary that is currently running, stopping it at a certain point and extracting out all of the memory data and register data. All that data will then be passed through to angr where symbolic execution will be performed. This example uses QEMU to run the binary and GNU Debugger (GDB) to extract data, thus the prerequisites for running this example are: GDB compiled for the desired architecture and with Python support as well as an installed and functioning QEMU User Emulation. This example is illustrated in Figure 9.

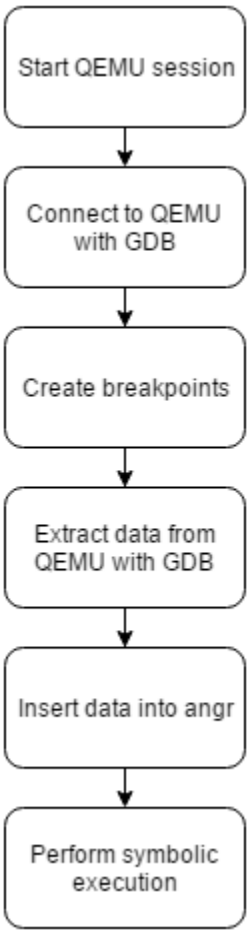


Figure 9. Flow diagram for performing "Live" symbolic execution. This process utilizes QEMU and GDB to emulate the binary and extract data to plug into angr.

For this example, a simple ARM binary with the code shown in Appendix D will be used. The goal is to extract just the foo function as a blob from the compiled binary and run symbolic execution on it to see the possible results. In addition, the binary uses pointers as parameters. To access this data, angr must be used to explore loading data from memory regions.

To begin, a QEMU session must be created and connected to through GDB. The GDB connection to QEMU can be scripted using Python, though this could also all be done on the command line if desired [29].The full GDB Python script can be found in Appendix D.

Once connected to the QEMU session there are a number of possible options. These include stepping through the program, setting breakpoints, and extracting data and memory. The execute command in the Python GDB will take in a string containing the GDB command that would be normally run in the terminal. Information on the registers can be dumped by turning on logging to

a file and issues the info registers command. Data dumps on the binary can also be delivered using built in GDB functions [30].

In this example, the breakpoint should be set at the start of the foo function. Breakpoints are made by creating a breakpoint object at the desired address. The created breakpoint sub-class is derived from the gdb.Breakpoint base class in the Python GDB module. The class includes a stop function that will be called when the breakpoint is triggered. The program being debugged will break if the stop function returns True. If desired, the stop function can be customized to perform certain tasks each time the breakpoint is triggered.

A breakpoint can be created by instantiating a class with a string representation of an address. The address we provided in this example is the location of the start of the foo function. When the breakpoint is hit, the registers can be examined and parsed out to a text file. An example of the register data is shown in List 13:

```
r0             0xf6ffec0c
r1             0xf6ffed74
r2             0xf6ffed7c
r3             0xf6ffec0c
r4             0xf6ffec30
```

List 13. Register output from dumping registers from GDB

Analysis of the binary in a disassembler shows that the parameter being passed through the r0 register is a pointer. The data in memory at the pointer address must be extracted to preserve the state when passing information to angr. The `parse_and_eval` command can be used to access data at a specific register. For this situation, since the value stored in the r0 register is a pointer, the binary memory at the register address can be dumped using the dump binary memory command.

Once the desired data has been retrieved, it can all be loaded into angr for analysis. The Python script for the angr analysis can be seen in its entirety in Appendix D. Since a blob is being loaded, there are some parameters that must be specified for angr such as the architecture as well as the entry point of the blob being loaded. These are configured under `main_opts` in the `load_options` parameter of the project loader.

The registers can be set by passing in the register file that had been dumped using angr's GDB module. The GDB module can also load data into memory at specific locations. Once all the registers and data are set, execution can begin by using the angr `explore()` tool. In this case, angr will only take one path and return the proper value determined by the branch. This is because there was a concrete value passed to the parameter of the foo function. angr takes this concrete value and takes the correct corresponding path through the function.

If the parameter is not set to a concrete value, angr will treat that parameter as a symbolic value. When angr performs execution on the symbolic value, it will take the branch and return two different possible paths each with a different return value.

If the parameter registers can be set by passing in the register file that had been dumped using angr's GDB module. The GDB module also can also load data into memory at specific locations. Once all the registers and data are set, execution can begin by using the angr tool. In this case, angr will only take one path and return the proper value determined by the branch. This is because there was a concrete value passed to the parameter of the foo function. angr takes this concrete value and takes the correct corresponding path through the function.

3.5 Different architectures and statically versus dynamically linked libraries

For the most part, angr tends to have the best support for x86 binaries. However, angr does support other architectures fairly well. There are a few exceptions here and there that must be taken care of when running analysis on binaries of different architectures.

When angr loads a binary, it will load the necessary libraries and their functions to go along with it. The functions that angr finds in the binaries will be re placed by angr's own python implementation of the library functions. This is because the library functions tend to be very large and will cause path explosion if stepped through. The Python implementations in angr have similar functionality to the originals, but implemented directly with symbolic variables in python (so no unbounded exploring is needed).

The library calls used in dynamically linked binaries are handled well by angr. However, the library calls from statically linked binaries have been embedded directly in the binary, making it hard for angr to automatically recognize and substitute with the Python implementations.

To solve this problem, the library calls must be hooked to angr's python implementation of the library calls The easiest way to see if hooking a call is necessary is to print out the trace of a *pathgroup* and see how the function calls are represented. If the `strcmp` function is represented as `ReturnUnconstrained strcmp`, it has not been hooked, and instead represents a valid path to be explored symbolically (with possible path explosion). Properly hooked functions will be represented as a `SimProcedure` object [26].

3.6 Chapter Summary

Symbolic execution in angr can be handled by four general steps. First, load the binary with desired parameters. The parameters being supplied can be: which libraries to load, which backend to use and what architecture to define the binary as, or a custom base address. Next the state can be set. Parameters such as starting addresses and *argc* or *argv* can be provided. Options

to avoid unsupported *syscalls* or to keep all memory reads symbolic. The *path* and *pathgroup* can then be created from the program state. The *path* and *pathgroup* are what will be used for the last step, to explore through the binary. Exploration can either search through the entire binary, or look for specific addresses. The paths found in the resulting stashes can be used to find the possible concrete values of the binary.

4. Binary Analysis Techniques with angr

This section will discuss the ways angr can be used for binary analysis. After loading a binary into angr and creating a Project object, a number of analyses methods can be called on that project. The ones shown in this section are:

1. CFGAccurate and CFGFast
2. BoyScout and GirlScout

4.1 CFGAccurate and CFGFast

Binary analysis in angr is based off of its control flow graph (CFG) generator. The CFG creates a graph based off of basic blocks. A basic block is a sequence of code with no branches except for the entry and exit. In angr, there are two CFG generators, one called CFGAccurate and one called CFGFast. They perform similar functions with differences shown in their names. CFGAccurate will step through and simulate every single program execution and keep track of all the states. CFGFast will only perform light-weight analysis based off of heuristics and strong assumptions.

In a well formed binary that has well defined functions and function symbols, CFGFast will be able to create the same CFG as CFGAccurate, but in a significantly faster manner. However, when the binary is not as well formed, CFGFast tends to be unable to solve constraints based off of the assumptions it makes and CFGAccurate ends up being the only one able to return a CFG. An example script on how to create graphs in angr and visual representations of them using angr-utils is shown in Appendix E.

Figure 8 shows a graph view representation of the CFG created by CFGAccurate of the password authenticator binary mentioned Section 3.1. The graph is plotted using the Python module angr-utils. The graph is relatively busy even though the binary being used is not that large. Most of the nodes in the beginning are initialization steps. The main function of the binary does not start till near the bottom. By specifying a start point for the CFG, the graph can be started at the main function to make a clearer, more easily interpreted graph.

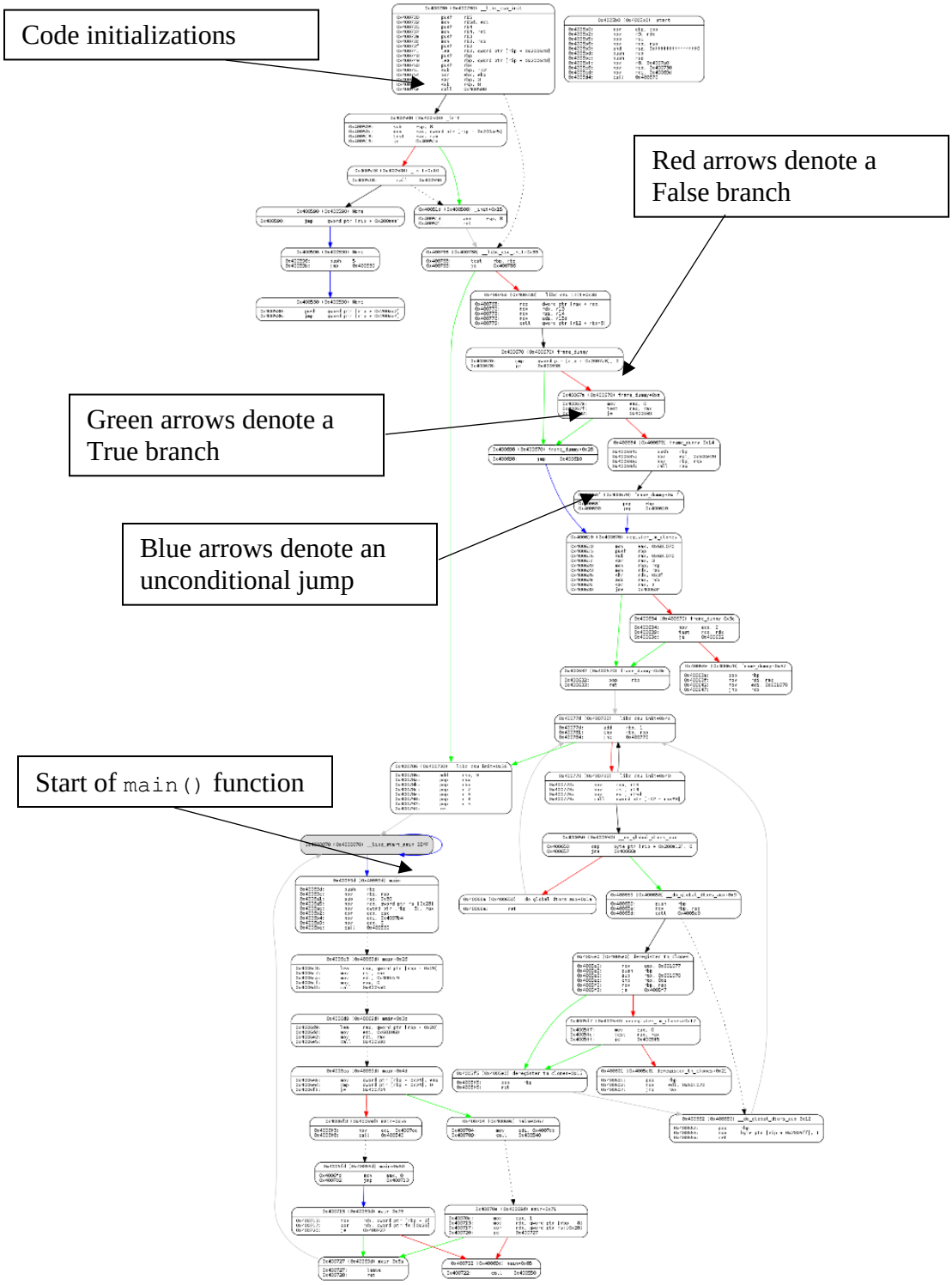


Figure 8. Full CFGAccurate of Password Authenticator Binary. Full view of the binary with all the initialization. The actual main function does not start until the middle of the graph.

The graph shown in Figure 9 is the same password authenticator binary but started at the main function. This graph is more readable and shows us more clearly what is happening in the binary. Each node represents a basic block and includes the beginning address, the name of the function is a part of, and the addresses and opcodes of each instruction in the basic block.

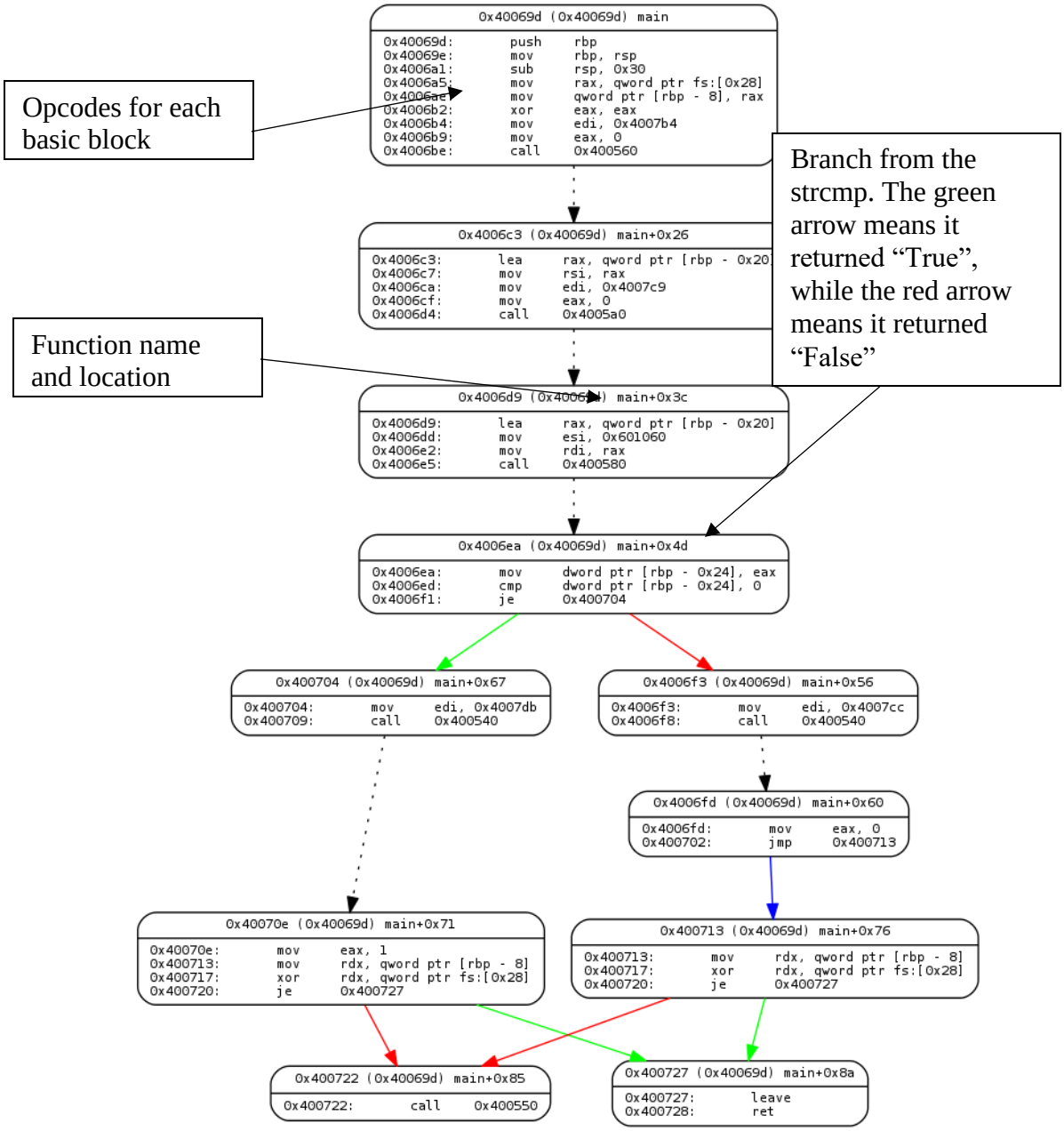


Figure 9. CFGAccurate on main() of Password Authenticator Binary

Referencing the source code of the password authenticator binary in Appendix A shows that there is a point where the program branches when comparing the hardcoded password to the

user inputted password. That point can be seen in Figure 9 at address 0x4006ea. The green arrow branching to the left denotes a “True” path while the red arrow branching to the right denotes a “False” path.

Generating CFG’s can also give analysts other important information about a binary, most notably its functions. A CFG has a knowledgebase that stores information about data references, function, etc. Using this knowledgebase, we can access and print out a list of functions. Part of an example output from doing so on the password authenticator binary is shown in List 14:

```
[<Function _init (0x400508)>,
 <Function sub_400530 (0x400530)>,
 <Function plt.puts (0x400540)>,
 <Function plt.__stack_chk_fail (0x400550)>,
 <Function plt.printf (0x400560)>,
 <Function plt.__libc_start_main (0x400570)>,
 <Function plt.strcmp (0x400580)>,
 <Function plt.__gmon_start__ (0x400590)>,
 <Function plt.__isoc99_scanf (0x4005a0)>,
 <Function frame_dummy (0x400670)>,
 <Function main (0x40069d)>,
...]
```

List 14. Output from printing out functions generated by the CFG. The items in the list include the function names as well as their start addresses.

angr will list the name of the function and the start addresses of each function. The knowledgebase can also be queried for a specific function to obtain more detailed information about it as shown in List 15. The script used to obtain this information is located in Appendix F.

```
Function main [0x40069d]
Syscall: False
SP difference: 0
Has return: True
Returning: True
Arguments: reg: [], stack: []
Blocks: [0x40069d, 0x400704, 0x4006fd, 0x4006d9, 0x4006c3, 0x400713, 0x400727,
0x40070e, 0x4006f3, 0x400722, 0x4006ea]
Calling convention: None
```

List 15. Output from printing out specific function data. The information includes whether the function is a syscall, the arguments, the basic blocks the function is apart of, etc.

4.2 BoyScout and GirlScout

For binary blobs that are not as well formed, angr has analyses tools that will attempt to determine architecture, endianness, and find functions. An example script on how to use BoyScout and GirlScout can be found in Appendix G.

BoyScout is the tool that will determine architecture and endianness. BoyScout does this by performing a pattern matching on all the architectures that angr supports with the binary and then voting on the architecture with the most hits. An example output on running BoyScout on a binary blob is shown in List 16.

```
angr.analyses.boyscout | AMD64 Iend_LE hits 12 times
angr.analyses.boyscout | X86 Iend_LE hits 8 times
angr.analyses.boyscout | ARMEL Iend_LE hits 0 times
angr.analyses.boyscout | ARMEL Iend_BE hits 0 times
angr.analyses.boyscout | AARCH64 Iend_LE hits 0 times
angr.analyses.boyscout | AARCH64 Iend_BE hits 0 times
angr.analyses.boyscout | PPC32 Iend_LE hits 0 times
angr.analyses.boyscout | PPC32 Iend_BE hits 0 times
angr.analyses.boyscout | PPC64 Iend_LE hits 0 times
angr.analyses.boyscout | PPC64 Iend_BE hits 0 times
angr.analyses.boyscout | MIPS32 Iend_LE hits 0 times
angr.analyses.boyscout | MIPS32 Iend_BE hits 0 times
angr.analyses.boyscout | MIPS64 Iend_LE hits 0 times
angr.analyses.boyscout | MIPS64 Iend_BE hits 0 times
angr.analyses.boyscout | The architecture should be AMD64 with Iend_LE
```

List 16. Output from running the BoyScout analyses to determine architecture and endianness. The analyses will vote on the possible architecture and endianness combinations and choose the one with the most hits.

GirlScout can be called as an analysis method on binary blobs. From the result, function addresses can be obtained, as well as a call map for the binary. The call map will give a function call graph with functions as nodes and calls between functions as edges. These results can all be printed out into and read from lists as shown in List 17.

The success of these two tools varies between different binaries. There are times when the outputted results do not seem realistic. For example, there are times when a small binary may return to have thousands of functions. These tools are also not optimized well and can only realistically be used for smaller binary blobs. Larger binary blobs take much too long to explore through and will most likely error out due to a lack of RAM.

```
Nodes:
['0x1000c',
 '0x10034',
 '0x10098',
 '0x100cc',
...]
Edges:
[('0x1000c', '0x49850'),
('0x1012c', '0x4a784'),
 ('0x10154', '0x100cc'),
 ...]
```

List 17. Output from printing the functions and function call edges from running the GirlScout analyses. The nodes are the starting addresses of each function and edges are calls between the functions.

4.3 Chapter Summary

angr can be a very useful binary analysis tool. It provides the ability to generate control flow graphs that show all the possible paths that can be taken in a binary. From the control flow graph of basic blocks, a function call graph can be created that will give a more general overview of the binary. These graphs can be used in a bigger scale for comparing binaries together to try to gain more information on what is happening in each binary.

The analysis on binary blobs is very useful in the case that an analyst just has a binary extracted from firmware with no information on architecture or endianness. It allows the analyst to automatically find out this information without having to manually analyze the binary.

5. Proposed angr Improvements

While angr is a great tool for reverse engineers, there is a steep learning curve before one can use it effectively. In an attempt to lower this learning curve, a script has been created to simplify the process. The script takes into account what a reverse engineer would typically look for when analyzing a binary. It mainly deals with symbolic variables and the possible values that each symbolic variable could possible take on.

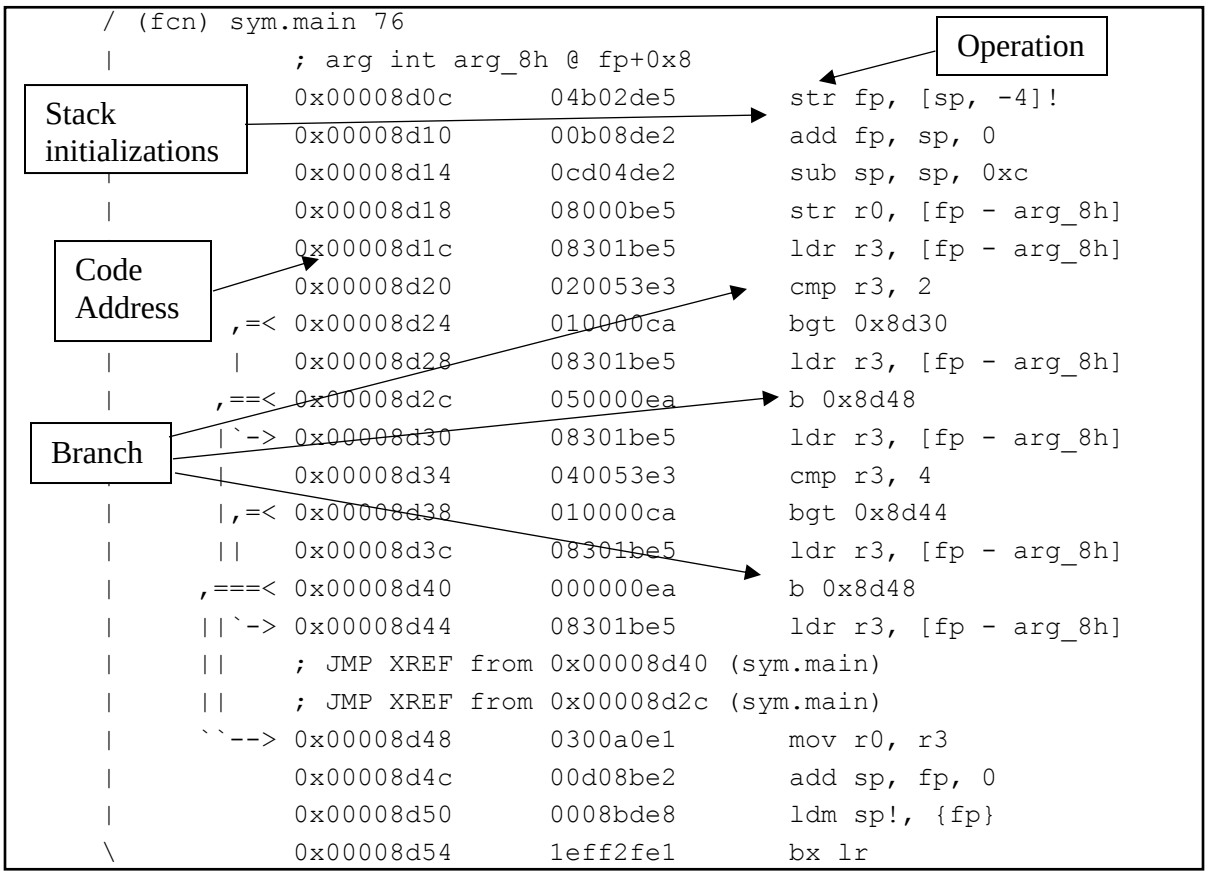
The script utilizes different portions of angr, but mainly interacts with the symbolic solver engine, Claripy. Claripy provides information on the symbolic variables at each stage of program execution. The information on each variable includes: maximum and minimum values, constraints, and a list of possible values. To showcase the functionality of the script, a simple test binary was used and compiled for four different architectures: ARM, x86, PowerPC, and MIPS.

```
int main(int x) {
    if(x < 3) {
        return x;
    } else if (x < 5) {
        return x;
    } else return x;
}
```

List 18. Source code for testing angr script. There are 3 possible paths that the program can take: if the input value is less than three, greater than three and less than five, or greater than five.

The source code for the simple test binary is shown in List 18. The binary takes an input and based off of that inputs, branches off into three possible paths. The paths are if *x* is less than three, if *x* is less than five, or if *x* is greater than five. To solve this example symbolically, the input *x* will be represented by a symbolic bitvector. The different architectures will use different registers to represent parameters of a function. The differences between the disassembly of the architectures is expanded on in the rest of this section.

5.1 ARM Disassembly




```
/ (fcn) sym.main 76
|           ; arg int arg_8h @ fp+0x8
|           0x00008d0c      04b02de5       str fp, [sp, -4]!
|           0x00008d10      00b08de2       add fp, sp, 0
|           0x00008d14      0cd04de2       sub sp, sp, 0xc
|           0x00008d18      08000be5       str r0, [fp - arg_8h]
|           0x00008d1c      08301be5       ldr r3, [fp - arg_8h]
|           0x00008d20      020053e3       cmp r3, 2
|        ,=< 0x00008d24      010000ca       bgt 0x8d30
|        |   0x00008d28      08301be5       ldr r3, [fp - arg_8h]
|      ,==< 0x00008d2c      050000ea       b 0x8d48
|      |`-> 0x00008d30      08301be5       ldr r3, [fp - arg_8h]
|      |    0x00008d34      040053e3       cmp r3, 4
|      |,=< 0x00008d38      010000ca       bgt 0x8d44
|      ||   0x00008d3c      08301be5       ldr r3, [fp - arg_8h]
|     ,===< 0x00008d40      000000ea       b 0x8d48
|     ||`-> 0x00008d44      08301be5       ldr r3, [fp - arg_8h]
|     ||    ; JMP XREF from 0x00008d40 (sym.main)
|     ||    ; JMP XREF from 0x00008d2c (sym.main)
|     ``--> 0x00008d48      0300a0e1       mov r0, r3
|           0x00008d4c      00d08be2       add sp, fp, 0
|           0x00008d50      0008bde8       ldm sp!, {fp}
\           0x00008d54      1eff2fe1       bx lr
```

List 19. Source code from List 18 compiled for the ARM architecture.

List 19 shows the source code from List 18 compiled for the ARM architecture. The ARM architecture uses the *r0* register to store the parameter of the main function. To initialize the start of the main function, the stack pointer and frame pointer addresses must be established. The stack is used to store temporary data, such as variables, of a function. The frame pointer will keep track of these variables in between functions. The frame pointer points to the where the stack previously was and the stack pointer has the current address of the stack.

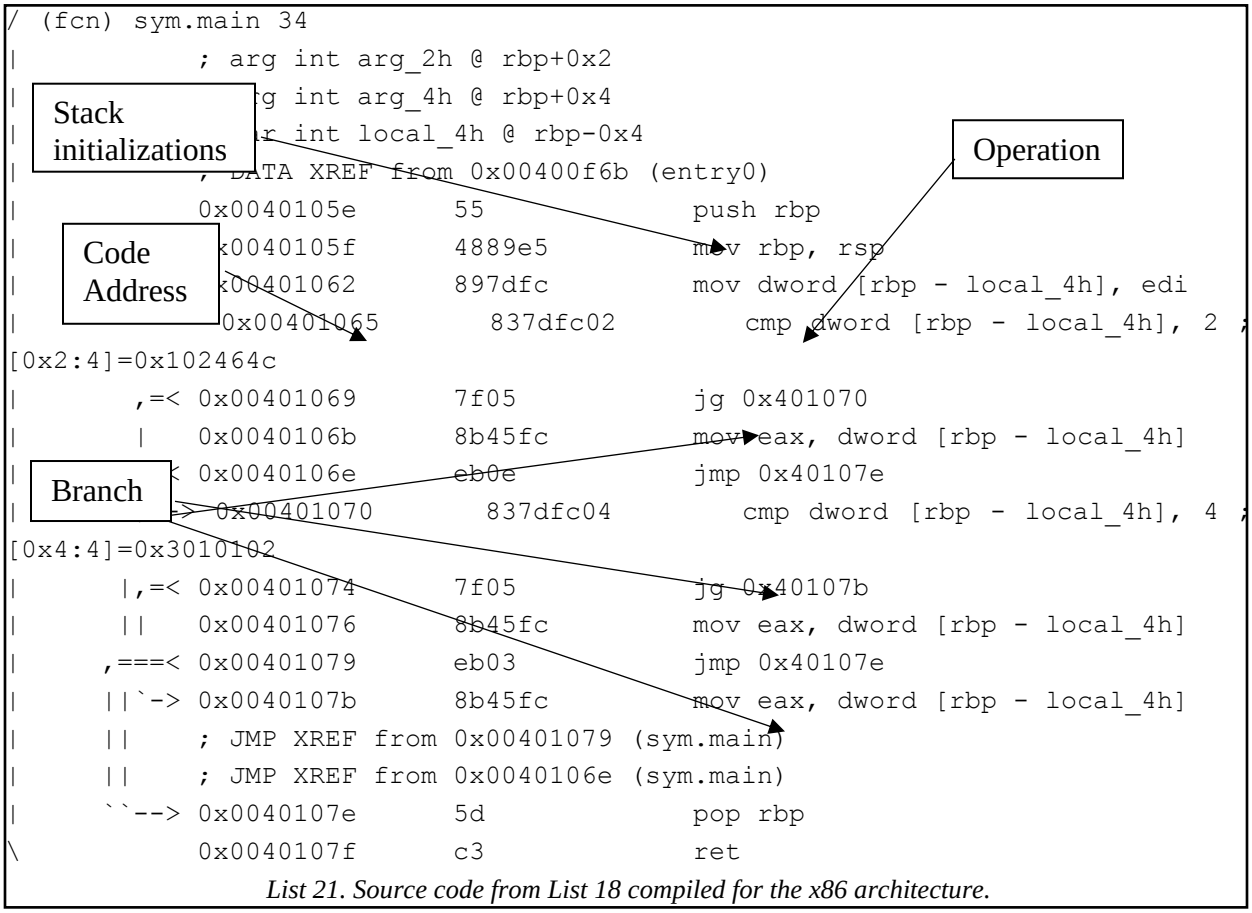
The disassembly starts with storing the value in the old stack pointer to register *r0* which is the symbolic argument. That value is then loaded into register *r3*. The *r3* register is now a symbolic value that represents the *x* variable and will be used for comparisons with concrete values using the `cmp` command. This is shown in List 20, where *r3* is compared to *2*. If the value is greater than two, it will branch to the address `0x8d30`.

```
cmp r3, 2
bgt 0x8d30
```

List 20. ARM comparison and branch command with a register and concrete value.

5.2 x86 Disassembly

While x86 disassembly follows the same general structure, there are a few noticeable difference that it has from ARM. List 21 shows the disassembly of the List 18 source code compiled for the x86 architecture.




```
/ (fcn) sym.main 34
|           ; arg int arg_2h @ rbp+0x2
|           ; arg int arg_4h @ rbp+0x4
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from 0x00400f6b (entry0)
|           0x0040105e      55              push rbp
|           0x0040105f      4889e5          mov rbp, rsp
|           0x00401062      897dfc          mov dword [rbp - local_4h], edi
|           0x00401065      837dfc02        cmp dword [rbp - local_4h], 2 ;
[0x2:4]=0x102464c
|       ,=< 0x00401069      7f05            jg 0x401070
|       |   0x0040106b      8b45fc          mov eax, dword [rbp - local_4h]
|       |   0x0040106e      eb0e            jmp 0x40107e
|       |-> 0x00401070      837dfc04        cmp dword [rbp - local_4h], 4 ;
[0x4:4]=0x3010102
|      ,==< 0x00401074      7f05            jg 0x40107b
|      ||   0x00401076      8b45fc          mov eax, dword [rbp - local_4h]
|     ,===< 0x00401079      eb03            jmp 0x40107e
|     ||`-> 0x0040107b      8b45fc          mov eax, dword [rbp - local_4h]
|     ||    ; JMP XREF from 0x00401079 (sym.main)
|     ||    ; JMP XREF from 0x0040106e (sym.main)
|     ``--> 0x0040107e      5d              pop rbp
\           0x0040107f      c3              ret
```

List 21. Source code from List 18 compiled for the x86 architecture.

In x86, the initialization of the main function starts with setting the stack base pointer (*rbp*) and the stack pointer (*rsp*) registers. The parameter passed through the main function is stored in the *edi* register. The value in the *edi* register is stored onto the stack using the *mov* command. That value on the stack is then compared to a concrete value as shown in List 22.

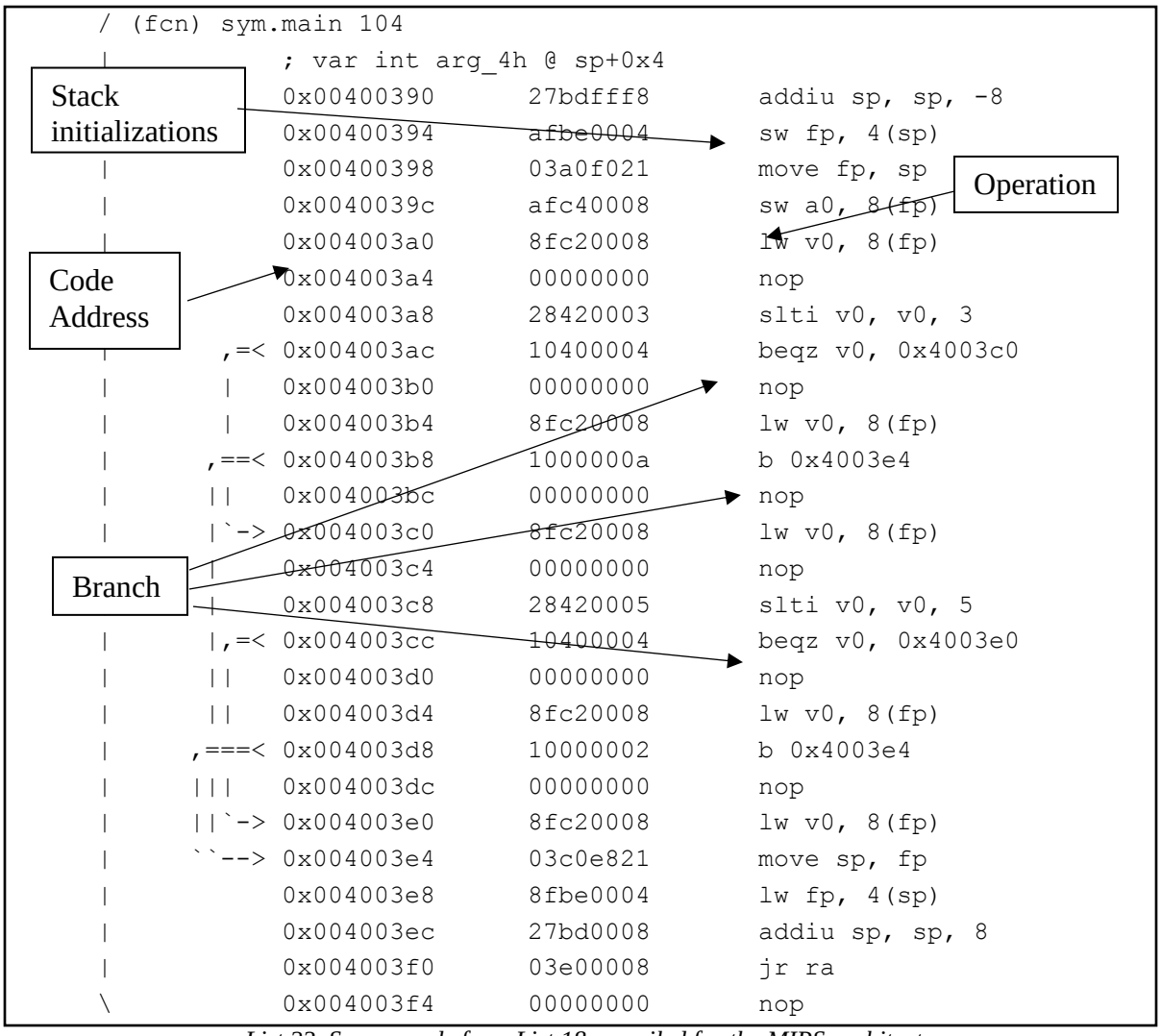
```
cmp dword [rbp - local_4h], 2
jg 0x401070
```

List 22. x86 comparison and branch command with the value on stack and concrete value.

The difference here between ARM and x86 is that ARM will load the value from the stack onto a register before performing a comparison, but x86 will simply compare the concrete value

to a value on the stack. Next is the MIPS architecture which again will follow the same general assembly flow with a few differences in the intricacies of how values are stored and compared.

5.3 MIPS Disassembly




```
/ (fcn) sym.main 104
|           ; var int arg_4h @ sp+0x4
|           0x00400390      27bdfff8       addiu sp, sp, -8
|           0x00400394      afbe0004       sw fp, 4(sp)
|           0x00400398      03a0f021       move fp, sp
|           0x0040039c      afc40008       sw a0, 8(fp)
|           0x004003a0      8fc20008       lw v0, 8(fp)
|           0x004003a4      00000000       nop
|           0x004003a8      28420003       slti v0, v0, 3
|         ,=< 0x004003ac      10400004       beqz v0, 0x4003c0
|         |   0x004003b0      00000000       nop
|         |   0x004003b4      8fc20008       lw v0, 8(fp)
|       ,==< 0x004003b8      1000000a       b 0x4003e4
|       ||   0x004003bc      00000000       nop
|       ||`-> 0x004003c0      8fc20008       lw v0, 8(fp)
|       ||    0x004003c4      00000000       nop
|       ||    0x004003c8      28420005       slti v0, v0, 5
|       ||,=< 0x004003cc      10400004       beqz v0, 0x4003e0
|       ||    0x004003d0      00000000       nop
|       ||    0x004003d4      8fc20008       lw v0, 8(fp)
|     ,===< 0x004003d8      10000002       b 0x4003e4
|     |||   0x004003dc      00000000       nop
|     ||`-> 0x004003e0      8fc20008       lw v0, 8(fp)
|     ``--> 0x004003e4      03c0e821       move sp, fp
|           0x004003e8      8fbe0004       lw fp, 4(sp)
|           0x004003ec      27bd0008       addiu sp, sp, 8
|           0x004003f0      03e00008       jr ra
\           0x004003f4      00000000       nop
```

List 23. Source code from List 18 compiled for the MIPS architecture.

List 22 shows the source code from List 18 compiled for the MIPS architecture. The disassembly at main starts with preparing the stack pointer and frame pointer registers, similar to ARM and x86. The value that is stored in *a0*, the register used for passing arguments, is stored onto the stack through the frame pointer. The *a0* register is can also be labeled as *r4*. The value stored onto the stack is then loaded onto the *v0* register, also known as the *r2* register. This value is the used to make comparisons to the concrete values as shown in List 24.

```
slti v0, v0, 3
beqz v0, 0x4003c0
```

List 24. MIPS comparison and branch command with the value in v0 and concrete value.

The logic used for comparisons is MIPS is also different from ARM or x86. The *slti* command stands for "set on less than immediate" with the syntax `slti $t, $s, imm`. This means that if `$s` is less than `imm`, `$t` will be set to one and if not, `$t` will be set to zero. The value stored into `$t` is then used to compare to see if it is equal to zero, and if so, the branch will be taken. This is different from ARM or x86 as those simply compare the register or value on the stack with a concrete value and based on that decide whether to take a branch. There is no additional extra uses of change the value of the register and then checking if it is equal to zero or not.

5.4 Python Script

The script for making binary analysis simpler is shown in Appendix H. The script contains initializations for loading the binaries as well as methods to step through the binary and analyze symbolic register information. It also contains helper methods that perform tasks such as returning all the active paths in a *pathgroup*, or returning all the possible registers for a certain architecture.

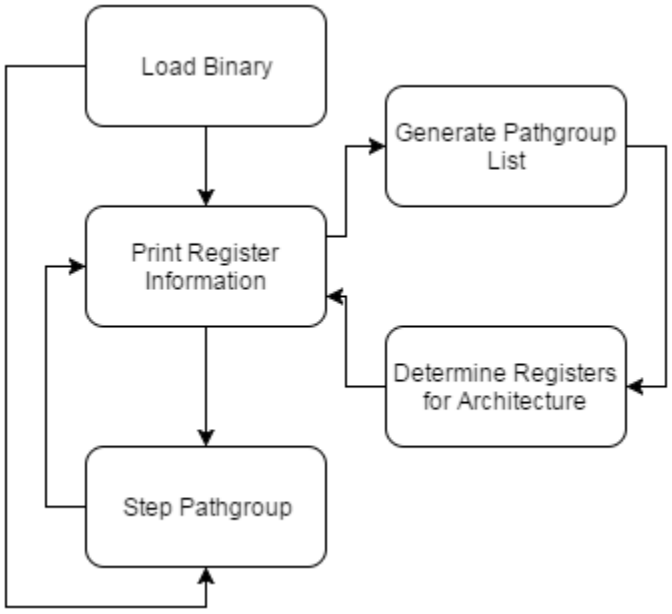


Figure 10. Flow chart diagram for the angr improvement script. The script starts by loading the binary and creating the pathgroup. Then a continuous loop of either stepping the pathgroup or printing register information occurs until there are no more active pathgroups to be stepped.

Figure 10 shows the general flow chart for the script. First the binary is loaded into angr where the path and *pathgroups* are created. From there, the *pathgroups* can either be stepped through or register information can be printed out. When printing out register information, there are two intermediate steps: generating a list of paths in the *pathgroup*, and determining the correct registers to use for the specific architecture. The cycle of stepping a *pathgroup* and printing out register information continues until there are no more active paths in the *pathgroup.*

To run the script, the user must provide the file name of the binary they want to perform execution on in the following format: `python script.py binary`. The script uses the Python *sys* module to read values from the command line. From the project object that is created, the *main* object can be extracted using the *get_symbol* method. The *main* object is used to set the *entry_state* to start at the main function.

Depending on the architecture, different registers will be set to have symbolic values. In this scenario, the register that will be passing the first parameter of a function is going to be set to be symbolic. The detected architecture is retrieved by calling the *arch* parameter of the project state. If the architecture is detected to be ARM, the *r0* register will be set to be symbolic. If the architecture is x86, the *eax* register will be symbolic. Finally, if the architecture is MIPS, the *r4* register will be set symbolic.

The *step* function in the script is straightforward. It first calls the *step()* method on the *pathgroup*. Once all the paths are updated, the *pgList* function is called to return all the current active paths in the *pathgroup* in a list. Using the list, the program counter for each path is printed out to give the user an idea of where they are in the program they are stepping. The *pgList* function iterates through the active paths in the *pathgroup* and adds the state of each path to a list and returns that list.

To obtain information about the registers, the *printRegInfo(reg)* function can be called. The function first calls the *pgList* function to obtain the list of possible active paths. Next, for each path in the *pathgroup* list, the path state will be used to determine symbolic information on the desired register. The register that the user wishes to examine is provided as a parameter when calling the *printRegInfo(reg)* function as *reg*.

Since different architectures have different naming conventions for their registers, the *stateRegs(state)* function is called to determine which architectures register names are needed. The *stateRegs* function consists of *if*-statements using the *arch* field of the program state to determine which architecture to use. Depending on the architecture, the function will return a dictionary that maps the register names to the corresponding state register. For example, a key for the ARM dictionary would be *r0* and the corresponding value in the dictionary would be “state.regs.r0”, where *state* is the program state of the path that is passed in as a parameter to the function.

Once the correct register names are returned from the *stateRegs* function, information on the registers can be obtained. The first value returned by the function is the program counter. The

program counter will tell the user where in the program the current path is at. The next value returned the type of object that is represented in the register by angr. The third and fourth values returned are the minimum and maximum values that the register could take on. The fifth value is a list of possible values the register could be. The number of values in the list can be adjusted to whatever the user wishes. Finally, the constraints on the register for all the possible values it can be are printed out. The outputs for this function will be expanded upon further in the results section.

5.5 Chapter Summary

There is a steep learning curve when it comes to learning how to use angr. This chapter dealt with creating a simple script that a reverse engineer can utilize to perform symbolic execution with angr without having to worry about many of angr's intricacies. The goal of the script is not only to allow the reverse engineer to use angr more easily, but also to provide the reverse engineer with more useful information that he would not find otherwise. The script includes information about the symbolic variables that are used in the binary.

The two main functions of the script are the *step()* function and the *printRegInfo()* function. The *step* function will step through the binary and print out the program counter to let the reverse engineer know where they are in execution. The *printRegInfo* function will print information about the symbolic variables that includes: minimum and maximum possible values, a list of possible values, and the symbolic constraints on the variable.

6. Experiment Results

The script created and shown in Appendix H can be run in an interactive shell in Python. This can be done by adding a '-i' flag with the Python command as follows: `python -i script.py binary`.. When this is a done, for the script in Appendix H, an interactive shell well be produced with the initialization of the binary already completed. This means that the entry state, path and *pathgroup* have all already been created.

6.1 Output Results

When the *step()* function is called, the *pathgroup* is stepped. The program counter for each active path in the *pathgroup* is printed out. The program counter register is represented by a 32-bit bitvector. The results are shown in List 25. The *pathgroup* is stepped by basic block, so after the *step()* function is called once, there are two active paths and thus two program counters printed out. After the second time the *pathgroup* is stepped, there are three active paths.

```
>>> step()
<BV32 0x8d30>
<BV32 0x8d28>
>>> step()
<BV32 0x8d44>
<BV32 0x8d3c>
<BV32 0x8d48>
```

List 25. Output from calling the step() function. The program counter register is displayed for each active path in the pathgroup

After each *step()* function is called the *printRegInfo(reg)* function can be called. The output from calling the *printRegInfo* function after one *step* function call for the ARM architecture is shown in List 26. There are two different sections in the output that represent the two different active paths that are in the *pathgroup*. Looking at the program counters, one path is at `0x8d30` while the other is at `0x8d28`. Looking back at the disassembly shown in List 19, this is the branch after comparing the symbolic value to the concrete value of 3. In the source code of List 18, this is after the line `if(x < 3)`.

```
>>> printRegInfo('r3')
---------------------------------------------------
Program counter: <BV32 0x8d30>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 2147483647
Minimum possible value in reg: 3
Possible values in reg: (1056964611L, 2013265920L, 251658243L,
2130706432L, 1073741824L)
State constraints: [<Bool !(symbolic_0_32 <=s 0x2)>]
---------------------------------------------------
Program counter: <BV32 0x8d28>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294967293L, 0, 4294967041L, 4294950913L,
3758096385L)
State constraints: [<Bool symbolic_0_32 <=s 0x2>]
```

List 26. Output of printRegInfo after one step() call. The binary used is the ARM compiled binary.

The path with program counter `0x8d30` means that the symbolic value *x* is not less than 3. The maximum and minimum possible values returned are `2147483647` and `3` respectively. This is logical because no value lower than 3 should be able to take this path. The value `2147483647` is $2^{31} - 1$, which is the positive limit of a signed integer. For the possible values list `(1056964611L, 2013265920L, 251658243L, 2130706432L, 1073741824L)` is shown. They are random values chosen by the solver engine to be shown by the *eval* method. In this script, the *eval* method is told to return up to five possible values, but that number can be increased to any number that is desired. The symbolic constraints for this path are `[<Bool !(symbolic_0_32 <=s 0x2)>]`. This means that the symbolic value in the register must be *not* (note the exclamation point) less than or equal to the value 2.

The other path with program counter `0x8d28` shows similar but opposite results. The maximum and minimum possible values are `4294967295` and `0` respectively. This may seem off because the value is supposed to be less than three, but `4294967295` is actually $2^{32} - 1$, which in binary is represented by `0xffffffff`. For a signed integer, that value translate to -1, which fits the constraints for this path. It doesn't exactly give the desired result, but that is why the constraint list is also printed out. The constraints list return `[<Bool symbolic_0_32 <=s 0x2>]` which means that the symbolic value should be a value less than or equal to 2.

When the *pathgroup* is stepped again, there are three results paths. The information for the registers in these paths can be printed out as shown in List 27. This time, the program counters for

the three paths are `0x8d44`, `0x8d3c`, and `0x8d48`. The first path, `0x8d44`, is now the path taken after the comparison of `else if (x < 5)` where the symbolic variable is not less than 5. The second path, `0x8d3c`, is the path after `else if (x < 5)` where the symbolic variable is less than 5 and not less than 3. The final path, at `0x8d48`, is after the branch `if(x < 3)`, where the symbolic variable is less than 3.

```
>>> printRegInfo('r3')
-----------------------------------------------------
Program counter: <BV32 0x8d44>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 2147483647
Minimum possible value in reg: 5
Possible values in reg: (131071L, 2147483589L, 2147483520L, 251658243L,
2139095040L)
State constraints: [<Bool !(symbolic_0_32 <=s 0x4)>]
-----------------------------------------------------
Program counter: <BV32 0x8d3c>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4
Minimum possible value in reg: 3
Possible values in reg: (3L, 4)
State constraints: [<Bool !(symbolic_0_32 <=s 0x2)>, <Bool symbolic_0_32[31:3]
== 0x0>, <Bool symbolic_0_32[2:0] <= 4>, <Bool (symbolic_0_32 == 0x3) ||
(symbolic_0_32 == 0x4)>]
-----------------------------------------------------
Program counter: <BV32 0x8d48>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294967293L, 0, 4294967041L, 4294950913L, 3758096385L)
State constraints: [<Bool symbolic_0_32 <=s 0x2>]
```

List 27. Output of printRegInfo after two step() calls. The binary used is the ARM compiled binary.

For the first path at `0x8d44`, the maximum and minimum values are `2147483647` and `5` respectively. This is logical as the value should only be greater than 4. This is also seen in the constraints list: `[<Bool !(symbolic_0_32 <=s 0x4)>]`. The second path at `0x8d3c`, has a couple more constraints. The maximum and minimum values are `4` and `3`. This is because the constraints for the variable are that it should be greater than 2 but less than 5. The only values that fit this description are 3 and 4. This is reflected when calling the *eval()* method on this path, returning `(3L, 4)` which contains the only two possible values.

The state constraints list is: `[<Bool !(symbolic_0_32 <=s 0x2)>, <Bool symbolic_0_32[31:3] == 0x0>, <Bool symbolic_0_32[2:0] <= 4>, <Bool (symbolic_0_32 == 0x3) || (symbolic_0_32 == 0x4)>]`. The list contains four different constraints to define the symbolic variable. The first constraint in the list states that the symbolic variable must not be less than or equal to 2. The second constraint states that bit 31 through bit 3 of the symbolic variable must have the value of 0. The third constraints states that the symbolic variable must be less than or equal to 4. The final constraint states that the only possible values that the symbolic variable can be is 3 or 4.

Finally, for the last path at `0x8d48`, the maximum and minimum values are `4294967295` and `0`. Again the maximum value is the decimal value of the unsigned integer -1. The constraints on the state is `[<Bool symbolic_0_32 <=s 0x2>]`, which tells that the value should be less than or equal to 2.

```
int main(unsigned int x) {
    if(x < 3) {
        return x;
    } else if (x < 5) {
        return x;
    } else return x;
}
```

List 28. The List 18 source code modified to have an unsigned integer instead of a signed integer.

The source code from List 18 was altered to test how angr handles an unsigned integer. The altered source is shown in List 28. The input variable *x* is now an *unsigned int*. The source code was compiled in ARM loaded with the script. The *step* function was called on it twice. The result is three active *pathgroups* and the information on registers can be printed out. The results are shown in List 29.

Since the symbolic variable is now restricted to 0 through 2^{32}, the first path has a maximum and minimum of `4294967295` and `5`. The maximum is notably bigger than the original `2147483647` because there does not need to be a bit designated to tell whether the number is positive or negative. The constraints for the second path are the same as before the source code was altered. However, the constraints for the last path are now different.

The maximum and minimum possible values are now shown as `2` and `0`. This is logical because angr no longer has to take into consideration negative numbers. The state constraint list has also changed. It now shows: `[<Bool symbolic_0_32[31:2] == 0x0>, <Bool symbolic_0_32[1:0] <= 2>, <Bool Or((symbolic_0_32 == 0x2), (symbolic_0_32 == 0x0), (symbolic_0_32 == 0x1))>]`. The first constraint states that bit 31 through bit 2 must

have the value of 0. The second constraints states that the possible values for the register must be 2, 0, or 1.

```
      Program counter: <BV32 0x8d44>
      Value in reg: <BV32 symbolic_0_32>
      Maximum possible value in reg: 4294967295
      Minimum possible value in reg: 5
      Possible values in reg: (16252943L, 4294443018L, 2147483648L, 1572879L,
4294967242L)
      State constraints: [<Bool 0x5 <= symbolic_0_32>]
      Program counter: <BV32 0x8d3c>
      Value in reg: <BV32 symbolic_0_32>
      Maximum possible value in reg: 4
      Minimum possible value in reg: 3
      Possible values in reg: (3L, 4)
      State constraints: [<Bool symbolic_0_32[31:3] == 0x0>, <Bool
symbolic_0_32[2:0] <= 4>, <Bool 0x3 <= symbolic_0_32>, <Bool (symbolic_0_32 ==
0x3) || (symbolic_0_32 == 0x4)>]
      ------------------------------------------------
      Program counter: <BV32 0x8d48>
      Value in reg: <BV32 symbolic_0_32>
      Maximum possible value in reg: 2
      Minimum possible value in reg: 0
      Possible values in reg: (2, 0, 1L)
      State constraints: [<Bool symbolic_0_32[31:2] == 0x0>, <Bool
symbolic_0_32[1:0] <= 2>, <Bool Or((symbolic_0_32 == 0x2), (symbolic_0_32 ==
0x0), (symbolic_0_32 == 0x1))>]
```

List 29. Results from the binary compiled in ARM from the altered List 28 source code.

The original source code in List 18 was compiled for the x86 and MIPS architecture. The results for those binaries tested with the script are shown in Appendix I. They have the same results as the ARM architecture, thus making this script architecture agnostic. Having an architecture agnostic script is useful to reverse engineers because they do not want to have to deal certain edge cases and problems that may arise with different architectures.

6.2 Improvement Measurement

This script improves the usability of angr for an analyst who has no prior experience with using the tool. Without the tool, the analyst would have to create the project themselves and figure out how to load the binary and perform all of the initializations that are necessary for angr to run. The analyst would have to look through angr source code to find out how symbolic values are

represented and how to access that symbolic information. Additionally, the analyst would have to create a new script for each binary that is a different architecture to accommodate for the differences. The script also takes into account problems that may arise that the analyst may not have been aware of, such as starting program execution of the binary at the main function. This is an important factor in avoiding path explosion.

All of these factors make the usability improvement script a must have for analysts wishing to use angr. It greatly decreases the amount of time that would have to be spent figuring out the intricacies and edge cases of angr. It reduces frustration that may be caused by not being able to figure out why some problems in angr are occurring. In addition to making angr more accessible, the script also provides useful information about symbolic values that were not previously available. The script gives symbolic information such as maximum or minimum possible values for a register as well as symbolic constraints for a specific register. This information is updated in real-time as a binary is stepped through and provides the analyst significant insight into what is happening in the binary.

6.3 Chapter Summary

The script shown in Appendix H offers improvements to the usability of angr. It also provides useful information to analysts using the script about symbolic values when performing symbolic execution. When the *step* function is called, the paths in the *pathgroup* created by the script will be stepped through by basic block. The program counter will be printed out so that the analyst knows at what point the program is at. The number of addresses printed out also denotes the number of branches the program has taken. The *printRegInfo* will provide information about a symbolic register, including minimum and maximum possible values of that register, possible values for that register, and a list of symbolic constraints on that register. An example list of constraints is `[<Bool symbolic_0_32[31:2] == 0x0>, <Bool symbolic_0_32[1:0] <= 2>, <Bool Or((symbolic_0_32 == 0x2), (symbolic_0_32 == 0x0), (symbolic_0_32 == 0x1))>]`. This list shows that the symbolic value must be less than or equal to 2.

The improvements that this script makes centers around creating less work for the analyst when it comes to setting up scripts to run angr. The tool allows the analyst to not have to deal with edge cases and debug why an angr script he wrote it not working. The analyst does not have to worry about learning the tool and thus can focus his efforts on reverse engineering and learning more about the binary.

7. Conclusions

This project tackled the complex task of analyzing a binary analysis tool, keeping in mind the desires of someone performing reverse engineering of a binary. Software reverse engineering deals with analyzing a binary to learn more about the binary's functionality. This information can reveal vulnerabilities or find any malicious parts of the binary. A binary analysis and reverse engineering tool should be able to give helpful information to an analyst in a simple manner. angr, a binary analysis tool developed out of UC Santa Barbara, was the focus of this project. The three main aspects of angr that were explored was:

1. Symbolic execution
2. Binary analysis
3. Extracting information for reverse engineers

7.1 Symbolic Execution

One of the interesting aspects of angr is the symbolic execution engine that it is built on. This project explored the possible use cases of the symbolic execution as well as its limitations. The symbolic execution use cases include:

1. Using angr as password authenticator solver
2. Using angr to extract flags from a Linux command
3. Using angr to solve the CMU Bomb Lab Phase 2
4. Using angr to perform "Live" symbolic execution

All of these methods were shown to be compatible with multiple different architectures. For each method, an example script and binary is provided to demonstrate the ability of angr to solve each problem. The password authenticator solver gave a simple introduction to using angr. It explored creating an entry state, a path, and a *pathgroup* to perform execution on and also explained the different stashes that can be part of a *pathgroup*. The example used the *strcmp* command to recognize the presence of two possible paths, one desired and one undesired path, and uses the symbolic solver to return the hard coded password that will trigger the desired path.

Extracting flags from a Linux command explored the idea of creating symbolic values and storing them in registers. This example also touched on how to hook functions. Hooking functions is necessary in some cases where a normal library call may cause path explosion. Library calls tend to be large and complex, so in order to avoid path explosion, they must be hooked with a simple function that has similar functionality. In this scenario, the *getopt_long* was the library call

that needed to be hooked. The function was called to parse the command line arguments and return them, so to replicate functionality, the hooked function returned a symbolic value in the x86 return register.

Another lesson from the *chmod* example was how to load command line arguments into the program state to change how execution is run. To run the *chmod* Linux command, an *argv* list had to be passed into the state creation to ensure that the binary ran properly. The *argv* list ended up having the *chmod* command as the first item and the symbolic variable (represented as a symbolic bitvector) as the second item.

The third example used angr to solve the second phase of the well-known Carnegie Mellon University Reverse Engineering Bomb Lab. This example again dealt with creating symbolic bitvectors but instead of loading them into registers, this time the symbolic values were pushed and popped from the stack. This problem required initial binary analysis using a disassembler to analyze the assembly code of the binary. Custom entry points for angr had to be experimented with to avoid path explosion but still have enough data to maintain the correct program state for the binary to run.

Finally, angr was used to perform "live" symbolic execution. This example showed the possibility of taking a running binary, pausing execution at a certain point of interest, extracting all the code and memory data, and then plugging the extracted data into angr for symbolic execution. To accomplish this, QEMU was used to emulate the binary and GDB was used to connect to the QEMU session to create breakpoints and extract data. The example binary included a *main* function that initialized a pointer variable to point to a concrete value. That value was then passed into a function called *foo* where the value was compared to be less than or greater than another concrete value.

To test out the "live" symbolic execution, the binary was run in QEMU right up until the *foo* function was called inside *main*. At that point, GDB was used to extract code data and register data. This was accomplished through Python scripting. The Python script dumped out all of the memory to text files and if necessary would parse them into a format that was readable to angr. angr contains a GDB module that allows the ingest of GDB data. That module was used to load the extracted state from QEMU into an angr project. The interesting part of this example was that if the memory data pointed to by the initialized pointer was not loaded into angr, angr would treat that value as symbolic. When it would reach the branch after the comparison to the concrete value, angr would return two possible paths that could be taken. However, if the memory data pointed to by the initialized pointer was loaded into angr, angr would return only the correct possible path that could be taken.

7.2 Binary analysis

Another interesting use case for angr is its prowess as a binary analysis tool. angr can be used to generate control flow graphs to visualize all possible paths of a binary, as well find more information about the functions through angr's function manager. For more unformed binary blobs, angr can be used to identify architecture, endianness, and find functions in the blob. The binary analysis tools in angr that were researched on were:

1. CFGAccurate and CFGFast
2. BoyScout and GirlScout

angr's control flow graph (CFG) is the foundation of the tools binary analysis and is a graph based off of basic blocks. The two CFG generators are called CFGAccurate and CFGFast. CFGAccurate will step through and simulate every single program execution and keep track of all the states. CFGAccurate is more useful for not well formed binaries because it does not make any assumptions to solve the graph in a faster manner. CFGFast is better for a well-formed binary that has well defined functions and function symbols. This technique will attempt to solve constraints quicker by making assumptions, which for well-formed binaries will generally be correct.

BoyScout and GirlScout in angr are used to analyze binary blobs. These binaries tend to not have section headers or function names which makes it hard to analyze the binary. BoyScout is an analysis tool that will attempt to find the architecture and endianness of the binary blob. GirlScout makes its own attempt at creating a CFG for the binary blob. It will look for certain jump or branch instructions in order to define where function boundaries are. GrilScout aims to find all the functions and edges between the function calls.

7.3 Extracting information for reverse engineers

In order to make the symbolic information in angr more accessible for analysts and reverse engineers, a script was created to highlight the usefulness of the symbolic data that angr can provide. Important information on the symbolic values include:

1. Program counter
2. Maximum possible value
3. Minimum possible value
4. List of possible values
5. List of symbolic constraints

The information listed can be found by using the script that is shown in Appendix H. The script contains functions that will step through the program and print out the symbolic register information. There are also helper functions that will return a list of active paths in a *pathgroup* as well as return a register list depending on the architecture of the binary being analyzed. Also depending on the architecture, the script will determine which registers must be set symbolic in the initialization of the binary state.

The program counter is important for the reverse engineer so that they know at what point of the binary they are currently stepping through. The maximum and minimum possible values give information on what the possible ranges for the symbolic variable may be. These values can be a bit misleading when it comes to signed integers. For example, the script may detect that the maximum value for a symbolic register is `4294967295`. This value is actually `-1` when translated to signed binary. Though misleading, the values are supplemented by the other information that the script provides.

The list of possible values and list of symbolic constraints give more reasonable representations of the symbolic variable. The list of possible values will return a list of whatever specified size of any possible value that the variable can take on. The list of constraints returns all the constraints on all of the symbolic values in the binary. An example of a symbolic constraint list is: `[<Bool !(symbolic_0_32 <=s 0x2)>, <Bool symbolic_0_32[31:3] == 0x0>, <Bool symbolic_0_32[2:0] <= 4>, <Bool (symbolic_0_32 == 0x3) || (symbolic_0_32 == 0x4)>]`. This list states that the symbolic variable must not be less than or equal to 2 and must be less than or equal to 4. Thus the only possible values for the symbolic variable are 3 or 4.

The information provided by this script is architecture agnostic. This is an important factor because the script should be as simple as possible for the reverse engineer. The reverse engineer will be able to take away from this script important symbolic information that will give more details about how the binary functions. Knowing the constraints of certain variables in different paths will give a better idea of what inputs will cause what parts of the program to execute.

7.4 Recommendations for Future Study

There were limitations to this project because of time constraints as well as resources. Further testing of angr can be done to expand its use cases. One of the major next steps for angr seems to be expanding its compatibility for Windows PE files. There is very rudimentary support for PE files. In general, only will formed PE binaries will have success in angr. One of the main focuses for adding more PE compatibility is having a more robust PE parser. Currently, angr uses the Pefiles Python module to parse PE files. For the most part, the module works well, but there are things that the module does not do. For example, the module is not able to find function names and symbols or identify the symbol table in a Pefile. In some cases, PE binaries to not include

symbol tables for which the Pefile module would not be able to find one. But for the cases where there is a symbol table include, the PE parser should be able to identify it.

Another aspect of angr that can be researched on is improving the avoidance of path explosion. The idea behind this improvement is to have a "smart" way of determining if a path that is being run is an important path to keep exploring. There are parameters that can be passed to the exploration functions that allow for testing of a path to see if it should be explored further. Further testing can be done on this idea to find its functionality. An example of this would be if there is a program that is trying to reach a certain path and the analyst knows that the value of a register must be between 0 and 50 for that path to be reached, a function should be written to prune out all paths where the register is not in the range of 0 to 50. In a similar manner, a performance boost would be a significant improvement to angr. Right now, larger binaries take a significant amount of time to load into angr and to generate control flow graphs. There are research papers about how to improve performance for symbolic execution. The idea of parallelization could be a solution to giving angr a performance boost.

angr is a successful binary analysis tool with an impressive symbolic execution engine. The application is open source and is actively being developed on. Addition of compatibility for more architectures and operating systems will greatly improve the usefulness of angr. Performance improvements to angr will also allow for the tool to be more widely used. Already a great tool for reverse engineers, these improvements will bring angr to the next level of binary analysis.

8. References

[1] M. G. Rekoff, "On Reverse Engineering," *IEEE Transactions On Systems, Man, and Cybernetics,* vol. 15, no. 2, pp. 244-252, 1985.

[2] Shellphish, "angr," [Online]. Available: angr.io. [Accessed 23 May 2016].

[3] Y. Shoshitaishvili, R. Wang, C. Salls and N. Stephens, "The Art of War: Offensive Techniques in Binary Analysis," Santa Barbara, 2016.

[4] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, Berlin, Springer Berlin Heidelberg, 2006, pp. 419-423.

[5] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM SIGSOFT Software Engineering Notes*, ACM, 2005, pp. 263-272.

[6] V. Chipounov, V. Georgescu, C. Zamfir and G. Candea, "Selective symbolic execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability*, Estoril, 2009.

[7] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager and M. G. Kang, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Information Systems Security: 4th International Conference*, Hyderabad, 2008, pp. 1-25.

[8] S. K. Cha, T. Avgerinos, A. Rebert and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380-394.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM,* vol. 56, no. 2, pp. 82-90, 2013.

[10] J. C. King, "Symbolic execution and program testing," *Communications of the ACM,* vol. 19, no. 7, pp. 385-394, 1976.

[11] J. Morris, "Computer Architectures: The Anatomy of Modern Processors," 1999. [Online]. Available: https://www.cs.auckland.ac.nz/~jmor159/363/html/CISC.html. [Accessed September 2016].

[12] C. Chen, G. Novick and K. Shimano, "RISC Architecture," [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/. [Accessed September 2016].

[13] T. Agarwal, "Understanding about RISC and CISC Architectures," 2014. [Online]. [Accessed September 2016].

[14] Sensepost, "A Crash Course in x86 Assembly for Reverse Engineers," Sensepost.

[15] A. K. Khonig, "All About EBP," 3 April 2012. [Online]. Available: https://practicalmalwareanalysis.com/2012/04/03/all-about-ebp/. [Accessed 15 September 2016].

[16] Microsoft, "x86 Architecture," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff561502(v=vs.85).aspx. [Accessed September 2016].

[17] M. McDermott, 2008. [Online]. Available: http://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf. [Accessed September 2016].

[18] ARM, "ARM and Thumb-2 Instruction Set," [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf. [Accessed September 2016].

[19] J. W. Bacon, "MIPS Registers," 2010. [Online]. Available: http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html. [Accessed September 2016].

[20] J. F. Frenzel, "MIPS Instruction Reference," 10 September 1998. [Online]. Available: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html. [Accessed September 2016].

[21] M. Burrell, "PowerPC," [Online]. Available: http://www.csd.uwo.ca/~mburrel/stuff/ppc-asm.html. [Accessed September 2016].

[22] angr, "Loading a Binary - CLE and angr Projects," [Online]. Available: https://docs.angr.io/docs/loading.html. [Accessed May 2016].

[23] angr, "Machine State - memory, registers, and so on," [Online]. Available: Loading a Binary - CLE and angr Projects. [Accessed May 2016].

[24] angr, "Bulk Execution and Exploration - Path Groups," [Online]. Available: https://docs.angr.io/docs/pathgroups.html. [Accessed May 2016].

[25] angr, "Solver Engine," [Online]. Available: https://docs.angr.io/docs/claripy.html. [Accessed May 2016].

[26] angr, "Top-level interfaces," [Online]. Available: https://docs.angr.io/docs/toplevel.html. [Accessed May 2016].

[27] R. E. Bryant and D. R. O'Hallaron, "Computer Systems: A Programmer's Perspective," Carnergie Mellon University, [Online]. Available: http://csapp.cs.cmu.edu/3e/labs.html. [Accessed September 2016].

[28] X. Kovah, "Introductory Intel x86: Architecture, Assembly, Applications, & Alliteration," [Online]. Available: http://opensecuritytraining.info/IntroX86.html. [Accessed September 2016].

[29] SourceWare, "Python API," [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html. [Accessed June 2016].

[30] "Copy Between Memory and a File," SourceWare, [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/Dump_002fRestore-Files.html. [Accessed June 2016].

[31] H. A. Muller, S. R. Tilley and K. Wong, "Understanding software systems using reverse engineering technology perspectives from the Rigi project," *CASCON '93 Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering*, vol. 1, pp. 217-226, 1993.

Appendix A

C source code:

```
#include <stdio.h>

static char password[] = "passw0rd";

int main(void) {
    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;
}
```

Python script:

```
import angr

proj = angr.Project('x86dym')
state = proj.factory.entry_state()
path = proj.factory.path(state)
pathgroup = proj.factory.path_group(path)

pathgroup = pathgroup.explore(find=0x00400704, avoid=0x004006f3)
print pathgroup
        for step in pathgroup.found[0].trace:
                print step
        print pathgroup.found[0].state.posix.dumps(0)
```

Appendix B

```
import angr

# chmod -v perm -rw-rw-r--

def hook_getopt(state):
        state.regs.eax = SYMB

def solve():
        global SYMB

        proj = angr.Project('chmod',
load_options={'main_opts':{'custom_arch':'x86'}})

        argv = ['chmod', angr.claripy.BVS('arg1', 8)]
        state = proj.factory.entry_state(args=argv, addr=0x401ac0)
        state.add_constraints(argv[1] >= ' ')
        state.add_constraints(argv[1] <= '~')
        SYMB = argv[1]

        proj.hook(0x401ac0, func=hook_getopt, length=5)

        path = proj.factory.path(state)
        path.state.options.discard("LAZY_SOLVES")
        pathgroup = proj.factory.path_group(path)
        pathgroup.explore()

        print pathgroup
        print "-------------------------------------"
        print "Errored result"
        # if(len(pathgroup.errored)):
                # print_trace(pathgroup, 'errored', 0)
        err = []
        for i in range(0, len(pathgroup.errored)):
                err.append(pathgroup.errored[i].state.se.any_str(argv[1]))
        print err
        print "-------------------------------------"
        print "Deadended result"
        dead = []
        for i in range(0, len(pathgroup.deadended)):
                # print_trace(pathgroup, 'deadended', i)
                dead.append(pathgroup.deadended[i].state.se.any_str(argv[1]))
        print dead
        print "-------------------------------------"
        print "Found result"
        if(len(pathgroup.found)):
                print_trace(pathgroup, 'found', 0)
                found = pathgroup.found[0].state.se.any_str(argv[1])
                print "The flag: %s" % found
        print "-------------------------------------"

def print_trace(pathgroup, state, i):
        if state == 'errored':
                for step in pathgroup.errored[i].trace:
                        print step
```

```
        elif state == 'deadended':
                for step in pathgroup.deadended[i].trace:
                        print step
        elif state == 'active':
                for step in pathgroup.active[i].trace:
                        print step
        elif state == 'found':
                for step in pathgroup.found[i].trace:
                        print step

if __name__ == '__main__':
        solve()
```

Appendix C

```
import angr

angr.path_group.l.setLevel('DEBUG')

proj = angr.Project('bomb')
state = proj.factory.entry_state(addr=0x08048b60)
path = proj.factory.path(state)
pathgroup = proj.factory.path_group(path)

ebp = state.regs.ebp
for i in range(6):
        state.memory.store(ebp-(0x18-(i*4)), state.se.BVS('push', 8))

pathgroup.explore(avoid=0x080494fc)
print pathgroup

anslist = []
deadend= pathgroup.deadended[0].state
for i in range(6):
        anslist.append(deadend.se.any_int(deadend.memory.load(ebp-(0x18-
(i*4)), 2, endness = 'Iend_LE')))

print anslist
```

Appendix D

C source code:

```
int foo(int * a){
    if (*a < 5) {
        return 0;
    }
    else {
        return 1;
    }
}

int main(void) {
    int * a;
    int aa = 3;
    a = &aa;
    int b;
    b = foo(a);
}
```

Qemu-gdb script:

```
#!/usr/bin/python
import gdb

class Breakpoint(gdb.Breakpoint):
        def stop(self):
                return True

def dumpAllRegs(reg_file):
        gdb.execute('set logging file %s' % reg_file)
        gdb.execute('set logging on')
        gdb.execute('info registers')
        gdb.execute('set logging off')

        #Parse first two columns
        with open(reg_file, 'r') as file:
                data = file.readlines()

        for i in range(0, len(data)):
                data[i] = data[i].split('\t', 1)[0] + '\n'

        with open(reg_file, 'w') as file:
                file.writelines(data)

def dumpRegData(dump_file, reg):
        val = gdb.parse_and_eval(reg)
        dumpMemory(dump_file, val, val + 0x4)

def dumpMemory(dump_file, start_addr, end_addr):
        gdb.execute('dump binary memory %s %d %d' % (dump_file, start_addr,
end_addr))

def connectQEMU(port):
```

```
        gdb.execute('target remote localhost:%d' % port)

if __name__ == "__main__":
        connectQEMU(1234)
        dumpMemory('~/Desktop/fun/leaf/foofunc', 0x68e1c, 0x68e58)
        Breakpoint('*0x68e1c')
        gdb.execute('c') #continue
        #TODO: Note (if true) that the previous line blocks until the
breakpoint is
        dumpAllRegs('regouts')
        dumpRegData('~/Desktop/fun/leaf/dataDump', '$r0')
        gdb.execute('c') #continue
        gdb.execute('q') #quit
```

angr solve script:

```
import angr

def test_gdb():
    proj = angr.Project('foofunc',
load_options={"main_opts":{'custom_arch':'ARM', 'custom_base_addr':0x68e1c}})

    state = proj.factory.entry_state()
    path = proj.factory.path(state)
    pathgroup = proj.factory.path_group(path)

    state.gdb.set_regs('regouts')
    # state.gdb.set_data(0xf6ffec0c, 'dataDump')

    pathgroup.explore()
    if len(pathgroup.active):
        print pathgroup.active
        print pathgroup.active[0].state.regs.r0
        print pathgroup.active[1].state.regs.r0
    if len(pathgroup.deadended):
        print pathgroup.deadended
    if len(pathgroup.errored):
        print pathgroup.errored
        print pathgroup.errored[0].state.regs.r0

if __name__ == "__main__":
    test_gdb()
```

Appendix E

```
import angr, pprint
from angrutils import *

angr.knowledge.function_manager.l.setLevel('DEBUG')

proj = angr.Project('x86dym')

cfg = proj.analyses.CFGAccurate()
cfgF = proj.analyses.CFGFast()

functions = [function for function in cfg.kb.functions.values()]
ffunctions = [function for function in cfgF.kb.functions.values()]

pprint.pprint(functions)
print len(functions)
print "-------------------------------------------------"
pprint.pprint(ffunctions)
print len(ffunctions)
print "-------------------------------------------------"
print cfg.kb.functions.function(name="main")
print "-------------------------------------------------"
print cfg.kb.functions.function(addr=0x400560)

plot_cfg(cfgF, "paper_cfg", asminst=True, remove_imports=True,
remove_path_terminator=True)
```

Appendix F

```
import angr

angr.path_group.l.setLevel('DEBUG')
angr.project.l.setLevel('DEBUG')
angr.cle.loader.l.setLevel('DEBUG')
angr.knowledge.function_manager.l.setLevel('DEBUG')

proj = angr.Project('x86dym',
load_options={'main_opts':{'custom_arch':'x86'}})

cfg = proj.analyses.CFGAccurate(enable_symbolic_back_traversal = True)
print "It has %d nodes and %d edges" % (len(cfg.graph.nodes()),
len(cfg.graph.edges()))
target_func = cfg.kb.functions.function(name="main")
print "------------------------------------------------"
print target_func
print "------------------------------------------------"
```

Appendix G

```
import angr
import pprint

angr.analyses.boyscout.l.setLevel('DEBUG')
angr.analyses.girlscout.l.setLevel('DEBUG')

proj = angr.Project('blob')

boy = proj.analyses.BoyScout()
print boy.arch, boy.endianness

girl = proj.analyses.GirlScout()
pprint.pprint(girl.functions)

func = girl.functions
newfunc = []
print len(func)
for f in func:
    newfunc.append(hex(f).rstrip("L"))
newfunc.sort()
pprint.pprint(newfunc)
print "--------------------------------------------------------------------------
-----"
nodes = girl.call_map.nodes()
newnodes = []
print len(nodes)
for node in nodes:
    newnodes.append(hex(node).rstrip("L"))
newnodes.sort()
pprint.pprint(newnodes)
print "--------------------------------------------------------------------------
-----"
edges = girl.call_map.edges()
newedges = []
print len(edges)
for edge in edges:
    edge = (hex(edge[0]).rstrip("L"), hex(edge[1]).rstrip("L"))
    newedges.append(edge)
newedges.sort()
pprint.pprint(newedges)
```

Appendix H

```
import angr, claripy, sys

proj = angr.Project(sys.argv[1]) # bvtestarm, bvtestx86, bvtestmips

main = proj.loader.main_bin.get_symbol("main")
state = proj.factory.entry_state(addr=main.addr) # start at address of main
path = proj.factory.path(state)
pg = proj.factory.path_group(path)

if state.arch.name == 'ARMEL':
    state.regs.r0 = claripy.BVS("symbolic", 32)
if state.arch.name == 'AMD64':
    state.regs.eax = claripy.BVS("symbolic", 32)
if state.arch.name == 'MIPS32':
    state.regs.r4 = claripy.BVS("symbolic", 32)

def step():
    pg.step()
    group = pgList()
    for state in group:
        print state.regs.pc

def pgList():
    pgList = [] # get list of active paths
    for i in range(len(pg.active)):
        pgList.append(pg.active[i].state)
    return pgList

def printRegInfo(reg):
    group = pgList()
    for state in group: # print out information for each active path
        dictionary = stateRegs(state)
        print"-----------------------------------------------------"
        print "Program counter:", dictionary['pc']
        print "Value in reg:", dictionary[reg]
        print "Maximum possible value in reg:", state.se.max(dictionary[reg])
        print "Minimum possible value in reg:", state.se.min(dictionary[reg])
        print "Possible values in reg:", state.se.eval(dictionary[reg], 5)
        print "Solver engine for reg:", state.se.any_int(dictionary[reg])
        print "State constraints:", state.se.constraints

def stateRegs(state):
    if state.arch.name == 'ARMEL':
        arm_dict = {'r0':state.regs.r0, 'r1':state.regs.r1,
'r2':state.regs.r2, 'r3':state.regs.r3, 'r4':state.regs.r4,
'r5':state.regs.r5, 'r6':state.regs.r6, 'r7':state.regs.r7,
'r8':state.regs.r8, 'r9':state.regs.r9, 'r10':state.regs.r10,
'r11':state.regs.r11, 'r12':state.regs.r12, 'sp':state.regs.sp,
'lr':state.regs.lr, 'pc':state.regs.pc}
        return arm_dict
    if state.arch.name == 'AMD64':
        x86_dict = {'rax':state.regs.rax, 'rcx':state.regs.rcx,
'rdx':state.regs.rdx, 'rbx':state.regs.rbx, 'rsp':state.regs.rsp,
'rbp':state.regs.rbp, 'rsi':state.regs.rsi, 'rdi':state.regs.rdi,
```

```
'r8':state.regs.r8, 'r9':state.regs.r9, 'r10':state.regs.r10,
'r11':state.regs.r11, 'r12':state.regs.r12, 'r13':state.regs.r13,
'r14':state.regs.r14, 'r15':state.regs.r15, 'rip':state.regs.rip,
'pc':state.regs.pc}
        return x86_dict
    if state.arch.name == 'MIPS32':
        mips32_dict = {'r0':state.regs.r0, 'r1':state.regs.r1,
'r2':state.regs.r2, 'r3':state.regs.r3, 'r4':state.regs.r4,
'r5':state.regs.r5, 'r6':state.regs.r6, 'r7':state.regs.r7,
'r8':state.regs.r8, 'r9':state.regs.r9, 'r10':state.regs.r10,
'r11':state.regs.r11, 'r12':state.regs.r12, 'r13':state.regs.r13,
'r14':state.regs.r14, 'r15':state.regs.r15, 'r16':state.regs.r16,
'r17':state.regs.r17, 'r18':state.regs.r18, 'r19':state.regs.r19,
'r20':state.regs.r20, 'r21':state.regs.r21, 'r22':state.regs.r22,
'r23':state.regs.r23, 'r24':state.regs.r24, 'r25':state.regs.r25,
'r26':state.regs.r26, 'r27':state.regs.r27, 'r28':state.regs.r28,
'sp':state.regs.sp, 'bp':state.regs.bp, 'lr':state.regs.lr,
'pc':state.regs.pc, 'hi':state.regs.hi, 'lo':state.regs.lo}
        return mips32_dict
    if state.arch.name == 'PPC32':
        ppc32_dict = {'r0':state.regs.r0, 'r1':state.regs.r1,
'r2':state.regs.r2, 'r3':state.regs.r3, 'r4':state.regs.r4,
'r5':state.regs.r5, 'r6':state.regs.r6, 'r7':state.regs.r7,
'r8':state.regs.r8, 'r9':state.regs.r9, 'r10':state.regs.r10,
'r11':state.regs.r11, 'r12':state.regs.r12, 'r13':state.regs.r13,
'r14':state.regs.r14, 'r15':state.regs.r15, 'r16':state.regs.r16,
'r17':state.regs.r17, 'r18':state.regs.r18, 'r19':state.regs.r19,
'r20':state.regs.r20, 'r21':state.regs.r21, 'r22':state.regs.r22,
'r23':state.regs.r23, 'r24':state.regs.r24, 'r25':state.regs.r25,
'r26':state.regs.r26, 'r27':state.regs.r27, 'r28':state.regs.r28,
'r29':state.regs.r29, 'r30':state.regs.r30, 'r31':state.regs.r31,
'pc':state.regs.pc, 'sp':state.regs.sp}
        return ppc32_dict
```

Appendix I

```
ARM Results:
-----------------------------------------------------
Program counter: <BV32 0x8d44>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 2147483647
Minimum possible value in reg: 5
Possible values in reg: (131047L, 2147483398L, 2113929222L, 2130706438L,
2146435078L)
State constraints [<Bool !(symbolic_0_32 <=s 0x4)>]
-----------------------------------------------------
Program counter: <BV32 0x8d3c>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4
Minimum possible value in reg: 3
Possible values in reg: (3L, 4)
State constraints [<Bool symbolic_0_32[31:3] == 0x0>, <Bool symbolic_0_32[2:0]
<= 4>, <Bool 0x3 <= symbolic_0_32>, <Bool (symbolic_0_32 == 0x3) ||
(symbolic_0_32 == 0x4)>]
-----------------------------------------------------
Program counter: <BV32 0x8d48>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294967293L, 0, 4294967041L, 4294950913L, 3758096385L)
State constraints [<Bool symbolic_0_32 <=s 0x2>]

x86 Result
-----------------------------------------------------
Program counter: <BV64 0x40107b>
Value in reg: <BV64 0x0#32 .. reg_48_11_64[31:0]>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (2L, 2147483651L, 4294967281L, 4294705153L,
4294959105L)
State constraints [<Bool reg_48_11_64[31:0] <=s 0x2>]
-----------------------------------------------------
Program counter: <BV64 0x401076>
Value in reg: <BV64 Reverse(Reverse(symbolic_9_32) .. 0x0#32)>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294966272L, 4294443008L, 33553409L, 1047553L, 0L)
State constraints [<Bool !(reg_48_11_64[31:0] <=s 0x2)>, <Bool
reg_48_11_64[31:0] <=s 0x4>, <Bool True>]
```

```
-----------------------------------------------------
Program counter: <BV64 0x40107b>
Value in reg: <BV64 Reverse(Reverse(symbolic_9_32) .. 0x0#32)>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294966272L, 4294443008L, 33553409L, 1047553L, 0L)
State constraints [<Bool !(reg_48_11_64[31:0] <=s 0x4)>, <Bool True>]

MIPS Results:
-----------------------------------------------------
Program counter: <BV32 0x4003e0>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 2147483647
Minimum possible value in reg: 5
Possible values in reg: (2147482624L, 2013265920L, 2143289344L, 1073741824L,
134217732L)
State constraints [<Bool 0x5 <=s symbolic_0_32>]
-----------------------------------------------------
Program counter: <BV32 0x4003d4>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4
Minimum possible value in reg: 3
Possible values in reg: (3, 4)
State constraints [<Bool symbolic_0_32[31:3] == 0x0>, <Bool symbolic_0_32[2:0]
<= 4>, <Bool 0x3 <= symbolic_0_32>, <Bool (symbolic_0_32 == 0x3) ||
(symbolic_0_32 == 0x4)>]
-----------------------------------------------------
Program counter: <BV32 0x4003f0>
Value in reg: <BV32 symbolic_0_32>
Maximum possible value in reg: 4294967295
Minimum possible value in reg: 0
Possible values in reg: (4294934528L, 4278190080L, 4294967280L, 1L,
4160749568L)
State constraints [<Bool !(0x3 <=s symbolic_0_32)>]
```