Validating Software States Using Reverse Execution

Nathaniel Christian Boland
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
Boland, Nathaniel Christian, "Validating Software States Using Reverse Execution" (2022). Browse all Theses and Dissertations. 2545.
https://corescholar.libraries.wright.edu/etd_all/2545

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
VALIDATING SOFTWARE STATES USING REVERSE EXECUTION

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science

by

NATHANIEL CHRISTIAN BOLAND
B.S.C.S, Wright State University, 2020

2022
Wright State University

Junjie Zhang, Ph.D.
Theory Director

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

Committee on Final Examination:

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Phu Phung, Ph.D.

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


A key feature of software analysis is determining whether it is possible for a program to reach a certain state. Various methods have been devised to accomplish this including directed fuzzing and dynamic execution. In this thesis we present a reverse execution engine to validate states, the Complex Emulator. The Complex Emulator seeks to validate a program state by emulating it in reverse to discover if a contradiction exists. When unknown variables are found during execution, the emulator is designed to use constraint solving to compute their values. The Complex Emulator has been tested on small assembly programs and is able to detect contradictions in program states. If developed further the Complex Emulator could be used to validate program states on larger and more elaborate software.
## Contents

1. Introduction ........................................... 1
   1.1 Motivation ......................................... 1
   1.2 Reverse Execution Overview ..................... 2

2. Design .................................................. 3
   2.1 Problem Formulation ................................ 3
   2.2 Overview ........................................... 3

3. Implementation ......................................... 5
   3.1 Emulation Overview .................................. 5
      3.1.1 Assembly Processing .............................. 5
      3.1.2 Compilation and Assembly ..................... 7
      3.1.3 Code Execution ................................ 9
      3.1.4 Imaginary Execution ............................ 10
      3.1.5 Reversing Loops ................................. 10
      3.1.6 Contradictions ................................ 12
   3.2 Structural Overview .................................. 13
      3.2.1 Compilers ...................................... 13
      3.2.2 Assemblers .................................... 16
      3.2.3 Environment ................................... 16
      3.2.4 Registers and Memory .......................... 17
# List of Figures

1. Emulation Overview ................................................. 4  
2. DependencyGraph Example (1) ..................................... 26  
3. DependencyGraph Example (2) ..................................... 27  
4. DependencyGraph Example (3) ..................................... 28
# List of Tables

<table>
<thead>
<tr>
<th>I</th>
<th>Results of Reversing Instructions if Exactly One Input is Symbolic</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>Truth Table for Reconciling Two Symbolic Numbers</td>
<td>21</td>
</tr>
<tr>
<td>III</td>
<td>A Truth Table for a Reverse AND Function</td>
<td>23</td>
</tr>
<tr>
<td>IV</td>
<td>Small Program</td>
<td>24</td>
</tr>
<tr>
<td>V</td>
<td>Reverse Execution Values from Listing 6</td>
<td>31</td>
</tr>
<tr>
<td>VI</td>
<td>Forwards Execution Values from Listing 6</td>
<td>32</td>
</tr>
<tr>
<td>VII</td>
<td>Forwards Execution Values from Listing 7</td>
<td>33</td>
</tr>
<tr>
<td>VIII</td>
<td>Reverse Execution Values from Listing 7</td>
<td>34</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simplified <code>emulate()</code> Pseudo Code</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Simple while-loop</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Pseudo Assembly Sample</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Pseudo Code for <code>ReverseMachineCode</code>’s Subtraction Operation</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Pseudo Code for <code>ReverseMachineCode</code>’s Reverse AND Operation</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Memory Access Example</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>Assembly Loop</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In program analysis it is often desirable to be able to determine whether specific data states are possible for a given program. Numerous methods exist for finding if a state is possible. One method is to perform symbolic execution [1].

In this work we use a form of reverse program execution to find if a given state is possible for a program. Our tool, Complex Emulator, takes as input a program, its reverse control flow graph, and a program state. It then attempts to validate the state by executing the program in reverse to find a contradiction.

1.1. Motivation

Finding if a program state is possible has applications in vulnerability detection. If a program can reach a segment of code with an invalid data state, then a vulnerability may exist. For instance, if the execution of a program can reach a divide operation where the divisor is zero, then a divide by zero vulnerability exists in the program. In this work we utilize reverse execution to discover if a state is reachable.
1.2. Reverse Execution Overview

Reverse execution is a well known technique. Other works have used reverse execution to recover states that were generated using forwards execution [2][3][4][5]. Some works perform reverse execution by reversing code and recovering saved values when an instruction is unreversible [2][3].

The system RETracer uses backward taint analysis to retrieve the cause of a crash [4]. RETracer is designed to be able to perform backward taint analysis even when only the final state is known [4]. RETracer uses concrete reverse execution as well as forwards analysis to achieve this [4].

In this work, we focus on performing reverse execution on states that were not produced by forwards execution in order to detect contradictions. As a result, we cannot use past states to generate the values of unreversible instructions. Therefore, similar to [4], we use constraint solving, and, like [5], we perform forwards execution when necessary to compute unknown values.

Unlike in the works [2][3][4][5], our focus for this work is not to perform reverse execution on states generated by forwards execution, but rather perform reverse execution on hypothetical states that were not produced by forwards execution to see if they are possible. Therefore, if a fully unreversible operation is come across, we only are interested in the unknown value if another concrete value is dependent on it (and therefore might cause a contradiction). If a variable does not effect any concrete values at any point during reverse execution, then we can simply set it to be symbolic and ignore it; it does not contribute anything to the contradiction detection.
Chapter 2

Design

2.1 Problem Formulation

A state is valid for a program if that state can be reached using forwards execution starting at a valid entry state (such as the beginning of a fully initialized program). Finding if a state is valid for a program using ordinary execution would in most cases be infeasible. In order to find if a state is valid we have built the Complex Emulator.

2.2 Overview

The Complex Emulator starts at a user specified state. It attempts to derive what the previous state would have been if the state had been generated using forwards execution. It does this by reverse execution. Like [2] the Complex Emulator performs reverse execution on assembly instructions.

Some instructions will have inputs and outputs that are unknown and unable to be derived at that point during reverse execution. When an instruction is come across that cannot be executed due to missing inputs, that instruction is compiled as an imaginary assembly instruction. Imaginary assembly cannot be executed until more information is obtained; as a result it is saved for later. If at any point one of the imaginary assembly
instructions can be executed (due to newly discovered information from executing other instructions), that instruction will be executed. An overview of the emulation process can be seen in Figure 1.

Contradictions are discovered when one variable must equal two contradictory values at the same time. If this ever occurs, then the path of execution taken by the reverse emulator was impossible.

![Fig. 1. Emulation Overview.](image-url)
Chapter 3

Implementation

The Complex Emulator was written in Python and has about 4850 lines of code. It is designed to emulate a subset of the ARM32 instruction set. It has the capability to emulate assembly programs both forwards and in reverse.

3.1 Emulation Overview

In the following section, the overall emulation design is discussed.

3.1.1 Assembly Processing

The most major component of the emulator is a class called ComplexEmulator. The ComplexEmulator class can emulate assembly code both forwards and in reverse. For a ComplexEmulator object to perform execution on an assembly program, it must first be initialized with the assembly code to be emulated, a direction for emulation, initial values for registers and memory, and, in the case of reverse execution, a reverse control flow graph. When a ComplexEmulator object is constructed, it will construct a ComplexEnvironment object. This ComplexEnvironment object will be used as the main environment for complex emulation.
Once a ComplexEmulator object has been constructed, the emulate() method can be run. The emulate() method executes all lines of code until either a contradiction is reached or all runnable instructions have been executed along a path of execution. Pseudo code of a simplified version of emulate() can be found in Listing 1. The emulate() method takes the next instruction to be emulated and compiles it into a real assembly line and, in some cases, an imaginary assembly line. If an imaginary assembly line was generated, then it is saved for later. The real assembly is then assembled into real machine code. This machine code is executed in the context of ComplexEmulator’s ComplexEnvironment object. After the real part of the execution has been handled emulate() checks if any imaginary code can be run. If it can, then it will be assembled and


```
method emulate(emulator, environment)
{
    while (address_to_execute and not environment.contradiction)
    {
        input_assembly_line = get_next_assembly_instruction(address_to_execute);
        complex_assembly_line = ComplexCompiler.compile(input_assembly_line, environment);
        emulator.save_imaginary_assembly(complex_assembly_line.imaginary);
        real_machine_code = RealAssembler.assemble(complex_assembly_line.real);
        environment.real = real_machine_code(environment);
        if (environment.imaginary.is_ready())
        {
            environment.imaginary.update_program_counter();
            while (not environment.imaginary.is_done())
            {
                imaginary_assembly = emulator.get_next_imaginary_instruction();
                imaginary_machine_code = ImaginaryAssembler.assemble(imaginary_assembly);
                environment.imaginary = imaginary_machine_code(environment);
            }
        }
        address_to_execute = environment.real.get_program_counter();
    }
    return;
}
```
executed with the ComplexEnvironment object. This is repeated until either a contradiction has been found or all code has been executed.

3.1.2 Compilation and Assembly.

The raw assembly program is taken as input for the ComplexEmulator class. Each instruction is processed by a ComplexCompiler object and then it is assembled into ‘machine code’ and executed. ComplexCompiler objects have a class method that will take as input assembly, an environment, and a direction of execution. Based on what direction of execution is passed into the compile() method, a ComplexCompiler object will process the assembly in different ways. If the forward direction is specified, then the assembly will be compiled to forwards assembly by the ForwardsCompiler class. If the reverse direction is specified, then the assembly will be compiled by the ReverseCompiler class into reverse assembly and possibly imaginary assembly. The ForwardsCompiler simply transfers the data from the assembly to a ForwardsAssembly object. The ReverseCompiler must compute the reverse version of that instruction.

The ReverseCompiler attempts to compile an assembly instruction into a ReverseAssembly object: the reverse version of the assembly instruction. To compile a ReverseAssembly object, several objects are required: the ReverseAssembly object to be compiled, a primary_environment, and an imaginary_environment. The primary_environment is the environment that the instruction will be run in; it is either a RealEnvironment object or an ImaginaryEnvironment object (see section 3.2.3.). The imaginary_environment is of type ImaginaryEnvironment and is used for execution of ImaginaryMachineCode.
For a ReverseCompiler object to compile assembly, certain constraints must be satisfied. For instance, if the forwards assembly is an addition, \( x + y \rightarrow z \), then at least two of the three variables must be concrete to find the other. If both of the inputs, \( x \) and \( y \), are known, then the instruction can be run in the forwards direction. If \( x \) and \( z \) or \( y \) and \( z \) are known however, then it must be run in the reverse direction.

A ReverseCompiler object must determine which direction, if any, it can compile an instruction into given the current primary_environment. If it is unable to compile the assembly line into real assembly, then it must compile it into an ImaginaryAssembly object and tell the dependency_graph (see section 3.2.4., Instruction Dependencies) of the imaginary_environment what values are needed by the un compilable instruction. If an instruction is compilable into real assembly, then a ReverseCompiler object will compute what instruction and what operands are required to perform the instruction. For instance, if the input assembly is \( r_0 + r_1 \rightarrow r_2 \) where \( r_0 = 3 \), \( r_1 = \text{symbolic} \), and \( r_2 = 7 \), then a ReverseCompiler object will determine that the instruction must be run as a subtraction operation: \( r_2 - r_0 \rightarrow r_1 \). If the input assembly is compiled into ImaginaryAssembly, then the ReverseCompiler object will return the ImaginaryAssembly object along with an instruction that moves a symbolic value into the input assembly’s output register indicating that the value is unknown for future execution. Also, the current values of the known registers, as well as the current sequence_number (see section 3.2.4.) must be saved into the imaginary environment.

After an instruction has been compiled into a ReverseAssembly object, it can be assembled by the assemble() function of ReverseAssembler. The ReverseAssembler assemble() function takes as input a processed ImaginaryAssembly object and decodes
what operation is needed. When the operation is determined, a pointer to a function that will accomplish the operation, along with any other required functions are bound in a lambda function. This lambda function is then returned by assemble() and acts as the ‘machine code’ for the emulator.

3.1.3 Code Execution

After assembling the assembly code, the emulator can call the resultant lambda function by passing the real and imaginary environments as arguments. The lambda function (i.e. ’machine code’) that gets returned by assemble() emulates the binary instruction. This lambda function may call several functions that will perform all of the necessary steps: a condition checker, pre-operator, the core instruction, and post operator.

First, it is determined if the instruction should be run given the conditions on the instruction. Then the pre-operations would be run. Pre-operations include any logical shifts that are specified by the instruction (these would be post-operations in forwards assembly). Then the core instruction gets run. In a typical instruction (e.g. the add instruction), several steps will occur in this phase. First, the actual operation is performed on the registers from the environment that was passed to the function. Then, any ImaginaryEnvironment object that was passed into the function will be notified of the new information that was obtained by running the instruction. Finally, the program counter will be updated and the primary_environment returned. After running the instruction a post-operation can be run. The post-operation will clear any register that became unknown as the result of the instruction. After the machine code has been run, the emulator can update the RealEnvironment’s sequence_number to the next value.
3.1.4 Imaginary Execution

After a real instruction has been fully executed, the *ImaginaryEnvironment* can be checked to determine if any imaginary instruction dependencies were satisfied. If the *ImaginaryEnvironment* has enough information to run at least one of the imaginary instructions cached earlier, then the emulation will start executing imaginary instructions. Before execution can occur, the instructions must be assembled by the *ImaginaryAssembler’s assemble()* function. Like *ReverseAssembler’s assemble()* function, the *ImaginaryAssembler’s assemble()* function will return a lambda function containing the instruction that will perform the assembly. However, unlike *ReverseMachineCode*, *ImaginaryMachineCode* determines what direction the instruction should be run in based on what constraints are satisfied. Then the *ImaginaryMachineCode* assembles the actual instruction into either *ForwardsMachineCode* or *ReverseMachineCode*. This real machine code is run with the *ImaginaryEnvironment* object as both the *primary_environment* and the *imaginary_environment* of the real machine code. The *ImaginaryMachineCode* can then save values of updated registers into the bound *RealEnvironment* object – thus propagating the results of the instruction into the main environment.

3.1.5 Reversing Loops

The *Complex Emulator* has a limited ability to handle loops. While executing a set of instructions the *Complex Emulator* attempts to detect loops and handle them as they are run across. The main purpose of this emulation technique is to detect
contradictions and break out of limitless reverse loops. A loop structure that has a well
defined end in the forward direction, may be undefined in the reverse direction. Consider
the simple case of Listing 2.

**Listing 2.** Simple while-loop.

```plaintext
1  int i = 0;
2  while(i < 10)
3      { array[i] = 0;
4          i++;
5      }
6...
```

In the reverse direction this code becomes problematic; \( i \) will continue to
decrement until it underflows and becomes a large positive number. This is clearly
nonsense since even without knowing line 1 since it could never have entered the
loop in the forwards direction if \( i \geq 10 \).

To handle loops the *Complex Emulator* saves the addresses of executed
branches that it comes across during execution. If a branch in a certain segment of
code is executed twice, then it is considered to be a loop. The loop is then
executed one more time, but this time the values are cached, and registers that are
changed in the loop are set to symbolic. The loop is then exited and the reverse
execution continues.

Using a simple dependency graph, the constraints for the loop’s starting
values are tracked during reverse emulation. If all of the constraints for all of a
loop’s starting values are satisfied, then the emulator will attempt to see if there
exists a loop based contradiction in the execution state. To do this, a side

*RealEnvironment* object is generated based on the *ComplexEmulator*’s main

*RealEnvironment*. A method is then run that will start executing the code at the

current address in the *forwards* direction. Once the loop is reached in this forwards
execution, the loop is executed to see if the registers changed by the loop ever all

equal the values saved from before. If the loop exits and the values never equal the

values found in reverse execution, then a contradiction is declared.

It should be noted that this method is somewhat experimental and is only
designed to work in limited cases. It also is not fully integrated with the rest of the

emulator, so it is prone to issues.

### 3.1.6 Contradictions

Some environment states are impossible to generate by executing code in the

forwards direction. If one of these impossible environments is used to execute the code in

reverse, then contradictions can occur. Detecting contradictions is a central feature of the

*ComplexEmulator*.

The *ComplexEmulator* can detect contradictions while executing reverse

instructions whose input/output combination is impossible. For example, a contradiction

will occur if the code segment in Listing 3 is executed in reverse starting at line 2 if r1 =

5 and r2 = 20. On line 2 the *ComplexEmulator* will execute the division r2 ÷ r1 and place

the result in r0. When line 1 is run, the reverse move operation will be executed. It will
notice that 5 is being moved into r0, but the value in r0 is 4; this means that there is a contradiction and the ComplexEmulator will raise an exception warning the user.

**Listing 3.** Pseudo Assembly Sample

```plaintext
1  r0 <- #5
2  r2 <- r1 * r0
```

### 3.2 Structural Overview

There are several main classes that are required for the Complex Emulator to emulate code. The primary component of the Complex Emulator is the ComplexEmulator class. This class has a function called `emulate()` which will run an assembly program in a specified direction with specified initial conditions. The other main components of the Complex Emulator are the family of classes related to the Compiler, Assembler, and the Complex Environment.

#### 3.2.1 Compilers

There are three compiler classes that are used by the Complex Emulator: ComplexCompiler, ForwardsCompiler, and ReverseCompiler. The primary use of these classes is in their `compile()` method. These functions are used by the emulator to convert raw assembly instructions into the proper type(s) of assembly.

**ComplexCompiler**

The ComplexCompiler’s `compile()` method takes as input three arguments; the assembly to compile, a ComplexEnvironment, and the direction to compile the assembly to. The ComplexCompiler checks the direction argument to determine whether to use the ForwardsCompiler’s or the ReverseCompiler’s `compile()` method to compile the assembly.
to. It then returns a tuple holding a \textit{RealAssembly} object and an \textit{ImaginaryAssembly} object. Depending on the exact circumstances the \textit{ImaginaryAssembly} object may be null.

\textit{ForwardsCompiler}

The \textit{ForwardsCompiler’s compile()} method simply copies a raw assembly instruction’s mnemonic and operands and returns it as a \textit{ForwardsAssembly} object. This object can then be passed to the \textit{ForwardsAssembler} to be assembled and then run.

\textit{ReverseCompiler}

The \textit{ReverseCompiler’s compile()} method is much more complicated than the \textit{ForwardsCompiler’s compile()} method. It takes as input an assembly instruction to compile; the primary environment the object is to be compiled in the context of; and, if relevant, an imaginary environment. It must examine what the operation of the instruction is, the operands of the instruction, and the current state of the environments to determine both what must be returned and how the dependency graphs should be updated. The \textit{ReverseCompiler’s compile()} method when complete will return a tuple of objects: a \textit{ReverseAssembly} object and an \textit{ImaginaryAssembly} object. In some cases when there is enough information in the environment to compile and run an instruction in reverse, the \textit{ImaginaryAssembly} object will be null.

How an instruction is compiled by the \textit{ReverseCompiler’s compile()} method depends greatly on the current state of the registers in the primary environment object that was passed to it. If enough registers in the instruction are symbolic, then the instruction cannot be executed in reverse. It must therefore compile the instruction as an \textit{ImaginaryAssembly} object and add the constraints to the \textit{ImaginaryEnvironment’s} dependency graph. If a forwards instruction modifies an output register then its value
before the instruction is run may be different from after it is run; therefore, when this type of instruction is reversed, even if there are too many unknowns to execute the instruction, the output register still must cleared. For example, if the following forwards instruction is come across $x + y \rightarrow z$ where $x = \text{symbolic}$, $y = \text{symbolic}$, and $z = 4$, then $x + y \rightarrow z$ will be returned as an \textit{ImaginaryAssembly} object and the reverse instruction, move $\text{symbolic} \rightarrow z$ Will also be returned. In this way, after the reverse instruction is executed, the register $z$ will be symbolic. Also, if the value of $x$ or $y$ is ever discovered, then the imaginary instruction can be run to find the value of the unknown register (or confirm that no contradiction exists).

\textit{ReverseCompiler’s compile()} function must respond differently depending on the type of instruction that is being run. For instance, if a noncommutative instruction is reversed, then depending on which input is known the order of operations, and even the instruction it gets translated to must differ. Table I shows the expressions that could be returned for different instruction. In each column of Table I it is assumed that all inputs and outputs are concrete except the input or output in the column header.

<table>
<thead>
<tr>
<th>Forwards Instruction</th>
<th>Symbolic Value:</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y \rightarrow z$</td>
<td>\textit{primary operation}</td>
<td>$z - y \rightarrow x$</td>
<td>$z - x \rightarrow y$</td>
<td>$x + y \rightarrow z$</td>
</tr>
<tr>
<td>Sequence Number: $t_0$</td>
<td>\textit{Dependency Graph}</td>
<td>\text{g.satisfy($t_0$, $x$)}</td>
<td>\text{g.satisfy($t_0$, $y$)}</td>
<td>\text{g.satisfy($t_0$, $z$)}</td>
</tr>
<tr>
<td>\textit{final operation}</td>
<td>\text{symbolic} $\rightarrow z$</td>
<td>\text{symbolic} $\rightarrow z$</td>
<td>\text{symbolic} $\rightarrow z$</td>
<td></td>
</tr>
</tbody>
</table>
3.2.2 Assemblers

The assemblers are divided into three classes: *ForwardsAssembler*, *ReverseAssembler*, and *ImaginaryAssembler*. These classes are used for their *assemble()* functions. The *assemble()* functions convert inputed assembly into forwards, reverse, or imaginary machine code, respectively. In each case, the ‘machine code’ that is returned is actually a lambda function binding the environments and operations required for the instruction. When this is called, the instruction will be executed in the context of the environments bound in the lambda function.

### 3.2.3 Environment

The environment with its components is the most elaborate piece of the *Complex Emulator*. The environment is where registers, memory, and instruction based dependencies are computed. The main environment class is the *ComplexEnvironment*; it

<table>
<thead>
<tr>
<th>Forwards Instruction</th>
<th>Symbolic Value:</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x - y \rightarrow z )</td>
<td><strong>primary operation</strong></td>
<td>( z + y \rightarrow x )</td>
<td>( x - z \rightarrow y )</td>
<td>( x - y \rightarrow z )</td>
</tr>
<tr>
<td><strong>Sequence Number:</strong></td>
<td><strong>Dependency Graph</strong></td>
<td>( \text{g.satisfy}(t_0, x) )</td>
<td>( \text{g.satisfy}(t_0, y) )</td>
<td>( \text{g.satisfy}(t_0, z) )</td>
</tr>
<tr>
<td>( t_0 )</td>
<td><strong>final operation</strong></td>
<td>( \text{symbolic} \rightarrow z )</td>
<td>( \text{symbolic} \rightarrow z )</td>
<td>( \text{symbolic} \rightarrow z )</td>
</tr>
<tr>
<td>( x \ XOR \ y \rightarrow z )</td>
<td><strong>primary operation</strong></td>
<td>( z \ XOR \ y \rightarrow x )</td>
<td>( z \ XOR \ x \rightarrow y )</td>
<td>( x \ XOR \ y \rightarrow z )</td>
</tr>
<tr>
<td><strong>Sequence Number:</strong></td>
<td><strong>Dependency Graph</strong></td>
<td>( \text{g.satisfy}(t_0, x) )</td>
<td>( \text{g.satisfy}(t_0, y) )</td>
<td>( \text{g.satisfy}(t_0, z) )</td>
</tr>
<tr>
<td>( t_0 )</td>
<td><strong>final operation</strong></td>
<td>( \text{symbolic} \rightarrow z )</td>
<td>( \text{symbolic} \rightarrow z )</td>
<td>( \text{symbolic} \rightarrow z )</td>
</tr>
<tr>
<td>( x \rightarrow y )</td>
<td><strong>primary operation</strong></td>
<td>( y \rightarrow x )</td>
<td>( x \rightarrow y )</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Sequence Number:</strong></td>
<td><strong>Dependency Graph</strong></td>
<td>( \text{g.satisfy}(t_0, x) )</td>
<td>( \text{g.satisfy}(t_0, y) )</td>
<td>N/A</td>
</tr>
<tr>
<td>( t_0 )</td>
<td><strong>final operation</strong></td>
<td>( \text{symbolic} \rightarrow y )</td>
<td>( \text{symbolic} \rightarrow y )</td>
<td>N/A</td>
</tr>
</tbody>
</table>
is composed of a RealEnvironment and an ImaginaryEnvironment. The RealEnvironment stores all register and memory values used in execution of real machine code. The ImaginaryEnvironment likewise is used for the execution of imaginary machine code, and it is also used to determine when dependencies are satisfied for real instructions.

3.2.4 Registers and Memory

Registers are emulated using a dictionary class, and memory is emulated using a specialized dictionary class called ChronologicalDictionary. The values that are stored in the registers and memory are objects of a class called SymbolicNumber. A SymbolicNumber object can either have a concrete value or a symbolic value. Even if a SymbolicNumber is concrete, it is possible to have certain bits be symbolic. The SymbolicNumber class has only been partially implemented.

Due to the requirements of running imaginary instructions, old memory and register values must be accessible. To accomplish this for registers, a DependencyGraph is used. For memory, however, the values are stored in a ChronologicalDictionary object.

ChronologicalDictionary. A ChronologicalDictionary object stores key-value pairs. Like other dictionaries, keys can be used to access their corresponding value. The differentia of a ChronologicalDictionary, is that the same key can hold many values; which value is accessed is determined by a member variable holding an ordinal value.

When a ChronologicalDictionary object is instantiated, an OrdinalNumber object is passed to it (OrdinalNumber is a class that implements ordinal integers). When a key value pair is added to the dictionary, the value of the current OrdinalNumber is also saved with the key. If a key-value pair \((k_0, v_0)\) is placed into a ChronologicalDictionary at
a given OrdinalNumber, \( t_0 \), then \(((k_0, t_0), v_0)\) is saved into the dictionary. If a key-value pair with a new value \((k_0, v_1)\) is placed in the dictionary and the OrdinalNumber is still \( t_0 \), then the new key-value pair will overwrite the previous value in the dictionary. If, however, the OrdinalNumber has been progressed to \( t_i > t_0 \), then when a new key-value pair is saved \((k_0, v_2)\), it is saved as \(((k_0, t_i), v_2)\) and the previous value is not overwritten (because the key \((k_0, t_i) \neq (k_0, t_0)\)).

If a key-value pair \((k_a, v_b)\) is placed into a ChronologicalDictionary object, and the current OrdinalNumber object is at \( t_b \), and key-value pairs \(((k_a, t_a), v_a)\) and \(((k_a, t_c), v_c)\) exist in the ChronologicalDictionary object where \( t_a < t_b < t_c \), then the new key-value pair \((k_a, v_b)\) will overwrite \(((k_a, t_a), v_a)\). The rules for reading values out of the dictionary given a certain \( k_i \) and \( t_i \) are the same.

By saving an OrdinalNumber object in the environment and incrementing it at every instruction, the memory value at any OrdinalNumber value can be accessed. The RealEnvironment’s main OrdinalNumber object is called its sequence_number.

**Stack.** The stack is emulated in memory. When a reverse push occurs, instead of an item being pushed onto a stack, the top item is popped off of the stack and compared to the value that was being pushed into the stack. If the two values contradict each other, then the current environment has a contradiction and the emulation exits. When a value is popped, it is simply pushed onto the stack.

**Instruction Dependencies.** Instruction dependencies are managed and computed using the DependencyGraph class. A DependencyGraph object is at the core of the Complex Emulator; it is how it determines what instructions can and cannot be run at a
given point of time. When an instruction cannot be run as a real machine code due to missing dependencies, the dependency graph gets the instruction added to it.

The dependency graph is primarily implemented using a directed hypergraph. Each edge of the hypergraph represents an instruction, and the vertices of those edges are the inputs and outputs of the instructions. A list is maintained of all of the current vertices (i.e. inputs/outputs) that are visible to the reverse emulator but are unknown and have dependencies on them. These are called ‘leaves.’

When a new instruction is run and a register’s value has been discovered during execution, the dependency graph is notified of this change. The list of leaves is checked to see if the vertex representing that register currently exists. If it does not, then nothing must be done to the hypergraph. On the other hand, if the register is in the list of leaves then that means that the register is a constraint of one or more of the previously unrunnable instructions. The hypergraph is queried to determine if that edge has enough vertices to satisfy it. If it does not, then the value can be bound in the vertex but the instruction cannot be run. If however the edge does have enough vertices to satisfy it, then the instruction can be run. The direction a satisfied instruction should be run is the direction from the known registers to the unknown register relative to the original direction of the instruction.

When an imaginary instruction is able to be run due to constraints being solved, it is possible that the execution of that instruction will result in an internal vertex being satisfied in the hypergraph. If this happens then the satisfied values will be saved in the internals of the graph: a similar mechanism for the leaves exists for internal vertices in the graph.
3.3 Reverse Execution Details

For the Complex Emulator to emulate a binary, a user must first load the instructions, a reverse control flow graph, and any initial register or memory values the user would like to set as constraints. The user must also set the address the emulator should start emulating at, as well as the direction the emulator should execute the assembly.

3.3.1 Contradiction Detection

One of the major features of the emulator is to detect contradictions. If concrete integer values are used to represent memory and registers, then many false contradictions can occur. For instance, if the expression 0011 AND $x = 0001$ is come across, then $x$ could be many values. If one value has to be chosen to satisfy the equation, say $x = 0001$, then later on if it were discovered that $x = 1101$, no true contradiction would exist (because 0011 AND 1101 = 0001), but the emulator would say there was: $x$ cannot both equal 0001 and 1101.

To mitigate this issue, memory and register values are emulated using the SymbolicNumber class. SymbolicNumber objects can store binary numbers with both concrete and symbolic bits. This means that if only part of a value is known, it can be stored in a SymbolicNumber with some of its bits concrete and some symbolic.

When the emulator finds that the same register must equal two different SymbolicNumbers at the same time. It must perform a comparison of the two. Because a simple equality check is unreliable for comparing SymbolicNumbers, the emulator has a function that attempts to reconcile the two numbers. That is, the function attempts to find
the most general value that would satisfy both symbolic numbers. Table II shows the truth table for reconciling two symbolic numbers (X represents bits that are symbolic and C indicates that a contradiction would occur given the combination). For example, if XX01 and X0X1 were to be run through the reconcile function, then the resulting value would be X001.

### Table II

**Truth table for reconciling two symbolic numbers**

<table>
<thead>
<tr>
<th>rec.</th>
<th>0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

#### 3.3.2. Reversed Instructions

In the simplest case, an instruction can be reversed by performing simple algebra on the variables to solve for the unknown. In many cases however, this does not work, because there is no single solution; some operations are irreversible [2]. A simple example of an equation that is irreversible is 0011 AND \( x = 0001 \) where \( x \) is an unknown variable. If the variable \( x \) is four bits, then \( x \) could equal 0001, 0101, 1001, or 1101. There are also values that it could not equal; for instance \( x \neq 0000 \). As another example if the emulator were ever to come across the expression 0010 AND \( x = 0001 \), then the emulator will know that a contradiction has been reached.

In order to emulate reverse instructions, each operation is written as its own function. These functions are all members of the `ReverseMachineCode` class. When an
instruction is ‘assembled’ by the ReverseAssembler, the ReverseAssembler will choose which of these ReverseMachineCode functions is appropriate.

**Reverse Addition.** The addition instruction is one of the simplest instructions to reverse because the reverse of addition is subtraction. The ReverseCompiler compiles addition instructions to subtraction instructions. Listing 4 shows pseudo code for the subtraction instruction in ReverseMachineCode.


```c
environment subtraction(operands, environment, imaginary_environment)  
{ 
    environment[operands[0]] = environment[operands[-2]] - environment[operands[-1]];
    if(imaginary_environment != NULL) 
    { 
        imaginary_environment.satisfy_dependencies(environment.get_program_counter(), 
        operands[0], 
        environment[operands[0]]);
    } 
    environment.update_pc();
    return environment
}
```

The instruction in Listing 3 begins by performing the actual subtraction itself. The operands argument holds an ordered list of the operands used in the instruction, the environment argument references what ever primary environment the instruction should be run in the context with, and the imaginary_environment argument is an ImaginaryEnvironment object.

**Reverse Logical AND.** To reverse a logical AND instruction, the emulator makes use of SymbolicNumbers. It should be noted that, unlike reverse XOR (which is simply XOR) reverse AND is not commutative. Table III shows a truth table for a reversed AND function. It should be noted that Input is the known input of the forwards AND instruction and Output is the known output of the forwards instruction. Output is not the
output of the reverse AND instruction. X and C stand for symbolic bit and contradiction respectively.

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th><strong>Output</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

The truth table in Table III is not unique. Some input/output combinations have multiple possible values. For instance, if Output is 0 and Input is X, all that is known is that at least one of the inputs must be 0. This means that in some cases the emulator will claim a contradiction when no contradiction is present. Listing 5 shows pseudo code for the reverse AND instruction.

**Listing 5.** Pseudo Code for *ReverseMachineCode*’s Reverse AND Operation.

```c
environment reversed_and(operands, environment, imaginary_environment) {
    environment[operands[0]] = and_inverse(environment[operands[-2]], environment[operands[-1]]);
    if and_inverse_contradicts(environment[operands[-2]], environment[operands[-1]])
        throw Contradiction;
    if( imaginary_environment != NULL)
        imaginary_environment.satisfy_dependencies(environment.get_program_counter(),
        operands[0],
        environment[operands[0]]);
    environment.update_pc();
    return environment
}
```
**Reverse Move.** A move instruction copies a value to a register. The source value can either be a known constant or a register. The format of a move operation is therefore $\text{val} \rightarrow \text{reg}$ where $\text{val}$ is the source value and $\text{reg}$ is some register. When a move operation is reversed a couple of possibilities exist for what the reversed move operation will do. If $\text{val}$ is a register, then whatever is in $\text{reg}$ can be moved into the register of $\text{val}$. If $\text{val}$ is not a register but rather a known constant, clearly no information can be gathered about $\text{val}$. In either case however, the values of $\text{reg}$ and $\text{val}$ can be compared; if they differ, then that means there is a contradiction. If no contradiction exists, then $\text{reg}$ will be set to be a *SymbolicNumber* (since the value would be completely unknowable before that instruction); execution will then continue.

**3.3.3. Example**

**Table IV**

**Small Program**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>mov r0, #18</td>
</tr>
<tr>
<td>6</td>
<td>mov r4, #4</td>
</tr>
<tr>
<td>8</td>
<td>mov r3, #11</td>
</tr>
<tr>
<td>10</td>
<td>mov r1, r3</td>
</tr>
<tr>
<td>12</td>
<td>add r1, r0</td>
</tr>
<tr>
<td>14</td>
<td>add r0, r1, r2</td>
</tr>
<tr>
<td>16</td>
<td>sub r3, r4</td>
</tr>
<tr>
<td>18</td>
<td>mul r0, r0, r3</td>
</tr>
</tbody>
</table>

The program in Table IV is an example program that could be loaded into the *Complex Emulator* with the direction of emulation set to be reverse. If r0 is set as 49, the
program counter as 18, and all other data registers as SymbolicNumbers, then the emulator would begin executing at the address 18 in reverse.

The instruction at address 18 is the operation $r_0 \times r_3 \rightarrow r_{0_1}$. In reverse, there are a couple of possibilities for what this instruction would be. If $r_3$ and $r_{0_1}$ were known but $r_0$ was unknown, then the reverse instruction would be $r_{0_1} \div r_3 \rightarrow r_0$. If, however, $r_0$ and $r_{0_1}$ were known but $r_3$ was unknown, then the reverse instruction would be $r_{0_1} \div r_0 \rightarrow r_3$. In the case of this example however, there are two unknowns $r_0$ and $r_3$.

Therefore the emulator cannot run this instruction. The instruction is therefore compiled to ImaginaryAssembly and the register $r_0$ is set to Symbolic because the value of it is no longer known. The program counter is then updated to point to the next instruction, i.e. address 16. Again the next instruction cannot be run because all three variables $r_{3_0}, r_{3_1},$ and $r_4$ are unknown. Therefore that instruction is also compiled to ImaginaryAssembly.

The program counter will then be updated to point to address 14. The same process will happen with both the instructions at addresses 12 and 10.

As these instructions were being compiled to imaginary assembly their constraints were being added to the environment’s DependencyGraph object. Figure 2 shows an overview of the structure of the DependencyGraph object when address 10 is being executed. Shaded vertices in Figure 2 represent leaf vertices.
The instruction at address 8, in the forwards direction, would be a move operation. This operation would move the constant value 11 into the register r3. When this instruction is reversed, r3 will be cleared and the program counter be set to 6. This is not all that happens though; once r3 is known the DependencyGraph can have that value added to it. When this happens, the leaf vertex r3 is set to 11. Instruction 16 is not satisfied because r3 and r4 are still unknown. However, instruction 10 is satisfied because now one of the two instructions is satisfied. Because 10 is satisfied it will be removed from the graph and 10 will be added to the imaginary environment instruction schedule.

Because the address 10 is in the environment’s imaginary instruction schedule, the emulator will compile and execute the imaginary assembly at address 10. The direction that the instruction at address 10 will be executed is in the forwards direction, because that is the direction that will propagate the discovered information from the instruction at address 8. Because instruction 10 is a move instruction that copies the value of r3 into r1, the outer vertices of r1 on instruction 12 in dependency graph and r3 of the instruction

![Fig. 2. DependencyGraph Example (1).](image-url)
16 will contain the value 11. No new information however is known about the register values in the current reverse execution at address 6.

When the reverse instruction at the address 6 is executed, the instruction at 16 will be executable; this is because two of the three registers are known (the outer r3 and r4). The address 16 will therefore be added to the imaginary instruction schedule. Figure 3 shows the dependency graph at this program state.

After running the instruction at 6, the imaginary assembly instruction from line 16 will be run. In forwards execution, the instruction at address 16 would perform the subtraction \( r3_0 - r4 \rightarrow r3_1 \). Because the vertices that are known are \( r3_0 \) and \( r4 \), that means that the instruction will be executed in the forwards direction. This means that the resulting value of \( r3 \) will be 7; this will be placed into the \( r3 \) vertex of the instruction 18 of the dependency graph. Now that two of the three vertices of 18 are satisfied 18 will be run. Because one input (\( r3 \)) and the output (\( r0_1 \)) of instruction 18 are known, but the other input \( r0_0 \) is unknown, the instruction must be executed in reverse. Therefore the forwards
instruction is compiled from r0 x r3 → r01 to r01 ÷ r3 → r00. After this is executed the resulting value of 49 ÷ 7 = 7 is placed into r00.

Continuing on with reverse execution, the next instruction to be emulated is the instruction at address 4; this instruction moves 18 into r0. The current state is depicted in Figure 4.

When this move occurs the imaginary instruction at 12 can be executed. Because the two values that are known are inputs (r10 and r0 of the instruction r10 + r0 → r1), that means that the instruction should be run in the forwards direction: this results in the addition 11 + 18 = 29. This leaves the imaginary instruction 14; it now can also be run. Again, because the two known vertices are an input and an output (namely r1 = 29 and r0 = 7), that means that the instruction should be run in reverse. Running the addition at address 14 in reverse results in the subtraction r0 - r1 → r2, i.e. 7 - 29 = -22. Because the r2 vertex is a leaf, the value of -22 will be saved in the register r2 of the RealEnvironment object. If reverse execution continued, the emulator would be able to use -22 as the value for r2.
3.4 Limitations

The Complex Emulator can execute programs in reverse in many cases. However, it also has many limitations. Given more development, the Complex Emulator could be used on a greater number of data samples.

Many of the features still need development to overcome some of their limitations and make them more useful. Reverse loop handling for instance, is a relatively new feature for the emulator, and only works under limited circumstances. Both reverse loops and memory would need to be more fully integrated in with the dependency satisfaction algorithms to improve performance.
Chapter 4

Demonstrations

In the following chapter the Complex Emulator has been used to execute example programs to demonstrate some of the features of the emulator.

4.1. Simple Example

Valid Starting State. If the program from Table IV is inputted into Complex Emulator and executed in the reverse direction with the starting state of r0 = 49 and pc = 18, then the emulator produces an output of r2 = -22, pc = 2, and all other registers are symbolic (aside from the stack pointer which is automatically assigned a value by the emulator).

When the emulator is run in the forwards direction with the input r2 = -22 and pc = 4, then after the emulator has executed the code, the register values are: r0 = 49, r1 = 29, r2 = -22, r3 = 7, r4 = 4, pc = 20, and all other registers (aside from the stack pointer which is an automatically assigned value and the current program status register) are symbolic. This confirms that the two are consistent.

Invalid Starting State. The starting state, r0 = 49, r2 = 5, and pc = 18, is invalid for the program in Table IV. If the emulator is run in the forwards direction with r2 = 5, then the result is that r0 = 238, r1 = 29, r2 = 5, r3 = 7, and r4 = 4. The register r0 cannot both equal 238 and 49.
When the Complex Emulator is run in the reverse direction with the invalid starting state $r0 = 49$, $r2 = 5$, and $pc = 18$, then the emulation ends with a contradiction. The contradiction it discovers is that when the instruction at 4 is executed $r0 = -9$ but the assembly implies that $r0 = \#18$ (mov r0, \#18).

4.2. Memory Access

The ARM32 assembly fragment of Listing 6 demonstrates the Complex Emulator’s memory storage capabilities.

**Listing 6. Memory Access Example.**

```
4  bne #100
6  mov r5, #20
8  mov r10, #0x80000
10 add r8, #40
12 str r8, [r10]
14 add r8, r5
16 mov r1, #0x80000
18 ldr r0, [r1]
```

To demonstrate the emulator’s memory handling capability, the following valid state $r0 = 2$, $r8 = 22$, and $pc = 18$ is passed into the emulator. The reverse emulation is run starting with with this state and it executes to completion. Table V shows the values after the reverse execution completes.

**Table V**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>SYMBOLIC</td>
<td>r8</td>
<td>-38</td>
<td>0x80000</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r1</td>
<td>SYMBOLIC</td>
<td>r9</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>SYMBOLIC</td>
<td>r10</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>SYMBOLIC</td>
<td>r11</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>SYMBOLIC</td>
<td>r12</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td>SYMBOLIC</td>
<td>sp</td>
<td>537427968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td>SYMBOLIC</td>
<td>lr</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>SYMBOLIC</td>
<td>pc</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>cpsr</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When the Complex Emulator is given the starting state r8 = -38 and pc = 6 (as well as sp = 537427968), the emulation completes with the state shown in Table VI.

**TABLE VI**

**FORWARDS EXECUTION VALUES FROM LISTING 6**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>2</td>
<td>r8</td>
<td>22</td>
<td>0x80000</td>
<td>2</td>
</tr>
<tr>
<td>r1</td>
<td>524288</td>
<td>r9</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>SYMBOLIC</td>
<td>r10</td>
<td>524288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>SYMBOLIC</td>
<td>r11</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>SYMBOLIC</td>
<td>r12</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td>20</td>
<td>sp</td>
<td>537427968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td>SYMBOLIC</td>
<td>lr</td>
<td>SYMBOLIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>SYMBOLIC</td>
<td>pc</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cpsr</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**4.3. Loop**

In the assembly fragment of Listing 7 there exists a small loop from lines 8 to 14.

Listing 7 can demonstrate the capability of the Complex Emulator to handle loops in reverse.

**Listing 7.** Assembly Loop.

```
2  mov  r1,  #5
4  mov  r3,  #0
6  mov  r5,  #10
8  eor  r5, r3
10  add  r3,  #1
12  cmp  r3,  #10
14  blt  #8
16  add  r0, r1
```

To illustrate this feature, first the code will be executed in the forwards direction to see what the values would be if this code were run; then the code will be run in reverse with the values generated by the forwards emulation to see if the emulator reaches the beginning without a contradiction. After demonstrating this, the program will be run again in the reverse direction, but this time with an invalid state.
Valid Inputs. Rather than finding values by hand the emulator was run in the forwards direction to find a valid state for the small program in Listing 7. Table VII shows the state produced by this emulation. Now that a valid state is known for Listing 7 some of the values from Table VII can be used to demonstrate the reverse execution capabilities of the emulator.

To emulate Listing 7 in reverse, several things must be given to the emulator. First, the emulator must be given the size of each instruction and a reverse control flow graph of the program: this was provided manually, but the process could be automated. The emulator can also be provided with data to execute the program with. In this case, the emulator will be provided with a valid state: r0 = 8, r3 = 10, r5 = 11, cpsr = 12, and pc = 16. Once all of the input data has been provided to the emulator, it can begin executing the program in reverse.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>8</td>
<td>r8</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r1</td>
<td>5</td>
<td>r9</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r2</td>
<td>SYMBOLIC</td>
<td>r10</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r3</td>
<td>10</td>
<td>r11</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r4</td>
<td>SYMBOLIC</td>
<td>r12</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r5</td>
<td>11</td>
<td>sp</td>
<td>537427968</td>
</tr>
<tr>
<td>r6</td>
<td>SYMBOLIC</td>
<td>lr</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r7</td>
<td>SYMBOLIC</td>
<td>pc</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cpsr</td>
<td>0</td>
</tr>
</tbody>
</table>

The emulation proceeds as follows. The emulator emulates the program in reverse starting at address 16. During this emulation, the emulator detects that addresses 8 to 14
are a loop. It therefore begins to handle the loop. After detecting what values were modified by the loop, the emulator sets all of those values to symbolic and exits the loop. After breaking out of the loop, the emulator can proceed with reverse execution. Once the instructions from 6 to 2, are executed, the emulator detects that all of the registers have been satisfied for the loop at addresses 8 to 14. The emulator then begins forwards execution at address 2 to gather all of the necessary registers needed to emulate the loop. Afterwards, it emulates the loop. Eventually, the loop detects that registers r1, r3, and r6 all equal values that they equaled during reverse emulation, so the loop emulation ends: it assumes no contradiction exists. The reverse emulator then emulates the instruction at address 2; it finds that r1 = 5. This satisfies the constraint for instruction 16. Therefore the imaginary instruction at address 16 is run. Table VIII shows the resulting register values after reverse emulation. No contradiction was found.

Table VIII

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>3</td>
<td>r8</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r1</td>
<td>SYMBOLIC</td>
<td>r9</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r2</td>
<td>SYMBOLIC</td>
<td>r10</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r3</td>
<td>SYMBOLIC</td>
<td>r11</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r4</td>
<td>SYMBOLIC</td>
<td>r12</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r5</td>
<td>SYMBOLIC</td>
<td>sp</td>
<td>537427968</td>
</tr>
<tr>
<td>r6</td>
<td>SYMBOLIC</td>
<td>lr</td>
<td>SYMBOLIC</td>
</tr>
<tr>
<td>r7</td>
<td>SYMBOLIC</td>
<td>pc</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cpsr</td>
<td>SYMBOLIC</td>
</tr>
</tbody>
</table>

**Invalid Inputs.** The state r0 = 8, r3 = 10, r5 = 100, and cpsr = 12, on the other hand, is invalid. This is because the register r5 must equal 11 after the loop, regardless of
other register values (it is dependent only on the instructions from the addresses 4 to 14; this loop always generates the same value when completed). When the *Complex Emulator* is given these inputs and the code is run in reverse, it discovers that a contradiction exists. It finds this when it attempts to run the loop in the forwards direction. It finds that the loop variables never all equal the values they did during reverse execution, implying that a contradiction may exist.
Chapter 5

Discussion

The Complex Emulator is able to detect if program states are possible. However, in the general case it is not able to ever completely prove that a given program state is not possible. To make the Complex Emulator more robust, multiple emulations would generally be required; also reverse control flow graph traversal algorithms would also need to be implemented. To make the Complex Emulator a more useful tool would require major additions.

One of the most essential additions that is needed is a control flow graph traversal algorithm. This would require reasoning about reverse branches. Further, multiple emulations would be required to handle the many possible execution paths.

Another major addition would be greater symbolic dependency handling. Currently, no dependencies may exist on symbolic bits. Symbolic bits in variables are used by Angr [1] and would have to be added to the Complex Emulator to handle a larger variety of input cases.
Chapter 6

Conclusion

The Complex Emulator is able to emulate programs in reverse in order to determine if certain states are valid or invalid. Using small handmade programs we have demonstrated that the Complex Emulator can successfully emulate assembly programs in reverse and detect contradictions. If the Complex Emulator were expanded to cover more possible scenarios, it could handle larger and more complicated programs.
References


