Loop Detection via Symbolic Constraint Compression

Jacob Colton Henry
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.
Loop Detection via Symbolic Constraint Compression

by

Jacob Henry

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science and Mathematics

by

May 2019

NOT FOR PUBLIC RELEASE

APPROVED:

Prof. Dan Dougherty and Prof. Randy Paffenroth
Abstract

Symbolic analysis of a program involves the evaluation of assembly instructions with arguments that aren’t specific constants, but instead symbolic values defined by Constraints. As instructions are executed symbolically, these constraints are updated to reflect restrictions on what values a variable can hold, and how different variables relate to one another. On some level, the constraints upon the state of the program are a reflection of its behavior, capturing the relation between inputs and outputs as a complex mathematical expression. By using a modification of existing Frequent Pattern Mining techniques, it is possible to find repeating structures in these constraints, and from them discover characteristics of the original program’s behavior. In this report we present our definitions and algorithm, and demonstrate using it to detect loops in several example C programs. The method presented shows promise against binary obfuscation techniques, and with further development could be a valuable tool for analyzing binaries that are problematic for static analysis techniques.
Acknowledgements

Special thanks to the MITRE corporation for sponsoring this project, and specifically Pete Brown and Peter Lucia for overseeing it. Also, thank you to Karen Lamb, Stan Barr, and Samantha Comeau for helping to provide background leads into how to approach this problem. And of course, thanks to my academic advisers Prof. Dougherty and Prof. Paffenroth, without whom the project would not even have existed.
Contents

1 Executive Summary .............................................. 1

2 Introduction ...................................................... 3
   2.1 Report Organization ......................................... 6

3 Background ....................................................... 7
   3.1 Symbolic Modeling of Program State ......................... 7
       3.1.1 Expressions ........................................... 8
   3.2 Execution Behaviour ......................................... 9
       3.2.1 Example: Basic Arithmetic .......................... 10
       3.2.2 Example: Fixed Length Loops ....................... 13
       3.2.3 Path Branching ...................................... 18
   3.3 Initial State ................................................ 19

4 Methodology .................................................... 21
   4.1 Expression Data Structure ................................ 21
       4.1.1 Traits ............................................... 22
       4.1.2 Expressions as Trait Trees ......................... 26
       4.1.3 Benefits of Trait Nodes .............................. 28
   4.2 Pattern Data Structure ..................................... 30
A.1 Sum Fixed-Size Array ............................................. 72
A.2 Sum Integer Range ............................................... 72
A.3 Sum Fixed-Size Nested Array ................................. 73
A.4 Iterate Conditional .............................................. 74

B Example Constraint Results ................................. 75
  B.1 Sum Fixed-Size Array ............................................. 75
  B.2 Sum Integer Range ............................................... 77
    B.2.1 GCC and Clang ............................................. 78
    B.2.2 Optimized GCC ........................................... 82
    B.2.3 Obfuscated Clang .......................................... 84
  B.3 Sum Fixed-Size Nested Array ................................. 89
    B.3.1 GCC, Clang and Optimized GCC ..................... 89
    B.3.2 Obfuscated Clang .......................................... 92
  B.4 Iterate Conditional ............................................ 94
# List of Figures

3.1 A simple mathematical expression. .................................................. 9
3.2 A simple C arithmetic function ...................................................... 11
3.3 Constraints after each line of figure 3.2. Identical subterms are color-coded to show how each line builds upon prior constraints. ............... 11
3.4 Final simplified constraints of 3.2 .................................................. 12
3.5 Source code for the function sum_byte_array, which sums a fixed-size array with 5 32 bit integer elements. ................................. 13
3.6 Initial constraint for sum_byte_array, reflecting int x = 0; ............ 13
3.7 Constraint on x after 1 iteration of the loop in sum_byte_array.  
Extract here represents selectively taking bits 159 through 128 (from right, inclusive) from arr, an operation equivalent to arr[0] in C. ... 14
3.8 Constraint on x after 2 iterations of the loop in sum_byte_array.  
Subtrees from previous figures color coded. ............................... 15
3.9 Constraint on x after 3 iterations of the loop in sum_byte_array.  
Subtrees from previous figures color coded. ............................... 16
3.10 Constraint on x after 5 iterations of the loop in sum_byte_array.  
Subtrees from previous figures color coded. ............................... 17
B.1 Constraint on return value, pre-compression .............................. 76
B.2 Constraints on return value, post-compression. Each row of the “Re-
currence” trait represents one instance of the function $f$, chained onto
the next row. Since there are at least 4 rows, this was classified as a
proper loop by our algorithm. ........................................ 77

B.3 Constraints on one-iteration return value, pre-compression. However,
this was small enough that no compression was even possible. .... 78

B.4 Constraints on two-iteration return value, pre-compression However,
this was small enough that no compression was even possible. Note
how the previous constraint can be found as a sub-tree of this one. .. 79

B.5 Constraints on two-iteration return value, post-compression. Here
we see exactly one instance of the function $f$. ......................... 80

B.6 Constraints on twenty-iteration return value, pre-compression. Though
it is likely too small to read, the repeating structure is visible. .... 80

B.7 Constraints on twenty-iteration return value, post-compression. .. 81

B.8 Constraints on one-iteration return value, without-compression. How-
ever, the constraint is small enough that no compression was possible. 82

B.9 Constraints on two-iteration return value, without-compression. How-
ever, the constraint is small enough that no compression was possible.
Note how the previous constraint can be found as a sub-tree of this
one. ................................................................. 83

B.10 Constraints on twenty-iteration return value, without-compression.
A repeating pattern is visible, but unfortunately our algorithm was
unable to detect it. ..................................................... 84

B.11 Constraints on one-iteration return value, pre-compression. However,
this was small enough that no compression was even possible. .... 85

vi
B.12 Constraints on two-iteration return value, pre-compression However, this was small enough that no compression was even possible. Note how the previous constraint can be found as a sub-tree of this one. 86

B.13 Constraints on two-iteration return value, post-compression. Here we see exactly one instance of the function $f$. 87

B.14 Constraints on twenty-iteration return value, pre-compression. Though it is likely too small to read, the repeating structure is visible. It is not as “clean” as in previous examples, but is still there. 88

B.15 Constraints on twenty-iteration return value, post-compression. Unfortunately the entire loop wasn’t able to be captured due to differences in the iterations. However, they were similar enough to meet our threshold. 89

B.16 Constraints on return value, pre-compression. Repeating structure is clearly visible, even if the constraint is large enough as to be rendered partially illegible. 90

B.17 Constraints on return value, post-compression. Each row of the loop here represents the addition of an element of the two-dimensional arrays. The arguments to the pattern function encode both which row we are operating on (here represented as 6 separate symbolic variables), and which bits from the row are being added (representing the index in the row). 91

B.18 Constraints on return value, pre-compression. The individual repeating structures are each larger than in the non-obfuscated compiler, however the actual overall chain-like structure is more or less the same. 92

B.19 Constraints on return value, post-compression. Parameters serve identical function to those in non-obfuscated constraints. 93
B.20 Constraints of state #53 path. ........................................ 94
B.21 Constraints of state #168 path. ..................................... 95
B.22 Constraints of state #173 path. ..................................... 95
B.23 Constraints of state #383 path. ..................................... 96
Chapter 1

Executive Summary

Symbolic execution is a subject of great interest to researchers focused on the automated analysis of program behavior. It has a great deal of potential for finding vulnerabilities that more conventional techniques could easily miss. Symbolic analysis can also be used more generally to study the flow of data through a program, generating “constraints” modeling how the inputs to a program relate to its behavior and outputs, as a set of paths and equations. This project’s goal was to utilize the constraints generated by the ANGR symbolic execution framework [15] [17] [14] to recover underlying program structure without needing any other known binaries to compare against. Tree pattern mining is the process of finding frequent sub-structures in a set of related trees. The most common sub-structures can give insight into what the most significant features of the trees are. The structures of constraint trees produced by symbolic execution of a program are closely related to the behavior of the program. Previous work (e.g. [20] and [5]) has already explored the concept of using constraints to identify program behavior, but have been restricted to comparing constraints of unknown binaries to a labeled set of constraints. Ours methods work independently of any database of known binaries, and can identify
features in programs it has never seen before.

First, we propose a new data type by the name of a Trait Tree, the exact details of which are described in section 4.1. Similar to the attribute trees in [13], a Trait Tree is a tree in which each node is a set. Trait Trees have the important property that each node/set contains items called “Traits”. Traits in turn contain a mixture of “core” quantitative values and “progeny” child trait-tree nodes. Which Traits a node contains essentially determine its “type” - what features we know that trait contains. Knowing that a node contains a certain set of traits allows us to be sure that a trait tree has a valid structure (e.g. by guaranteeing the number and order of children). Throughout this report, we use Trait Trees (more specifically, Expression Trees) to represent the Expressions which form ANGR ’s constraints.

Along with the new data type Trait Trees, we provide a corresponding definition for patterns on these trees, outlined in section 4.2. Trait Tree patterns are basically the same as Trait Trees, except with the inclusion of a wildcard (⋆) to denote where any value or descendant is acceptable.

Finally, these definitions assist us in constructing our algorithm outlined in section 4.3. This algorithm is a variant of the pattern mining algorithms presented in reports [21] and [13]. Using this algorithm, we are able to somewhat reliably detect loops from constraints alone in a variety of binaries (outlined in section 5) by finding repeating sub-structures in a program’s constraints. Though the mining algorithm primarily takes inspiration from the classic tree mining techniques of gSpan [21], it additionally uses pseudo-random pattern extension as described in [10]. Our most significant contributions here are the advantages offered by the Trait Tree data structure to mining performance. Usage of Traits allows us to ignore evaluating trees that do not represent a valid constraint, increasing overall efficiency of the mining process.
Chapter 2

Introduction

Symbolic execution is a software analysis technique wherein all or part of a program’s assembly instructions are executed “symbolically”. This means that instead of having an entirely concrete set of inputs or arguments (such as 0, 1, True, False, "foo", "bar"), some values are instead represented by a symbolic variable, such as “x” or “y”. As the symbolic executor “executes” instructions, its internal state is updated to reflect relationships between the program’s inputs and its behavior, such as execution path and outputs. Notable symbolic executors include ANGR ([15], which was used for this project), KLEE ([3]), Xaandra ([11]), and Mayhem ([4]). For a general overview of symbolic execution techniques, we direct readers to the survey published here: [1].

At any point during execution of a program, the symbolic executor contains a record of how exactly the inputs to a program restrict the outputs or internal behavior of a program. These constraints have many useful properties for software analysis. For instance, they can help verify that a program behaves in a safe manner (e.g. will not crash). Constraints can also provide a description of the inputs required to get a certain execution path (e.g. a call to an administrator-only
function).

Perhaps the most important aspect of symbolic execution constraints is their viability in automation of binary analysis. Symbolic execution does not generally require the source code for its analysis target, and so this method is well suited to investigating programs about which nothing is really known aside from the binary instructions. This was demonstrated in the 2016 DARPA Grand Challenge [16], wherein researchers were tasked with creating software systems capable of autonomously analyzing and exploiting/hardening provided binaries. Nearly every finalist used symbolic execution as part of their strategy, due to its viability for modeling program behavior across many possible inputs.

As a more specific hypothetical case, suppose one was interested in some function $\text{foo}(\text{int } x, \text{ int } y)$, within some binary executable for which the source code was not available. If one wished to know for sure that this function was capable of causing the program crash, they would conventionally only have 2 options:

1. Analyze the assembly instructions $\text{foo}$ directly.

2. Exhaustively check every input to $\text{foo}$.

Option 1 is frequently done, but is not complete. Even with the source code available to the analyst, it isn’t guaranteed that they will spot a mistake if one exists - after all, such oversights are the only reason a problem would exist to begin with! This sort of analysis can be time consuming and requires an advanced understanding of the system under study.

Option 2 is complete, in that if we did check every possible input, we would know for sure whether or not a crash was possible. However, even if $\text{foo}$ is relatively simple, its two integer arguments have $(2^{32})^2$ possible input configurations, which would take a very long time to exhaustively test. Though it is complete, it is in
general not practical due to time constraints.

However, if one were to utilize symbolic execution, they could effectively test all possible inputs at once. Instead of using concrete arguments for $x$ and $y$, they would instead symbolically execute $\text{foo}(s_0, s_1)$, where $s_0$ and $s_1$ are symbolic variables. A more thorough explanation of how this execution works, including examples, can be found in Section 3.2.

Admittedly, symbolic execution is not a cure-all - it has its own set of problems, most notably an exponential blow-up in resource usage when executing conditional loops. The specifics of why this happens are explained in Section 3.2.3. Regardless, this issue makes the process of identifying and mitigating the problems caused by loops while executing symbolically of great interest to researchers. Perhaps the most significant work on this topic is the PhD thesis “Symbolic Execution and Program Loops” [19], which outlines a variety of strategies for identification and optimal symbolic execution of loops.

However, as observed in “Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping,”[20] this and many other approaches that look directly at the assembly instructions are brittle if the binary is designed with the intent of thwarting common assembly analysis techniques. In their report they take advantage of the fact that no matter how much the code is obfuscated, it still must perform certain operations to accomplish its goal (in their case, decrypting a file). They used symbolic execution to monitor constraints for the telltale signs of various looping cryptographic functions.

That is not the only work focused on using constraints to recognize program behavior. A previous MITRE MQP report “Symbolic Execution for Function Matching” ([5]) tackled a similar problem of identifying functions based on their generated constraints. They utilized string similarity algorithms to compare constraints, for-
matting them as mathematical expressions and then processing similarity. Unfortunately, usage of string structures left their algorithm more sensitive to insignificant differences in constraint structure. For instance, essentially meaningless changes in variable names (e.g. “x_0_32” vs “y_3_32”) could harm similarity scores of identical constraints.

In our work, we focus on the more general topic of recognizing program structure via observing the structure of the constraints generated by symbolic execution. This avoids some of the problems faced by “Symbolic Execution for Function Matching”. This can hopefully be leveraged into identifying functions or optimizing symbolic execution around discovered loops - however, those are beyond the scope of this project.

2.1 Report Organization

In Chapter 3, we provide a more thorough overview of how most symbolic execution engines work, and how exactly their constraints manifest. In Chapter 4, we first outline our unique representation of constraints in Section 4.1. We then discuss how to describe patterns on these constraints in Section 4.2. Section 4.3 presents our contributions to quickly finding common structures between symbolic constraints, and Section 4.4 elaborates on how we use these common structures to compress constraints and discover loops. Finally, in Chapter 5 we show some practical demonstrations of our algorithm and provide closing thoughts on future work in Chapter 6.
Chapter 3

Background

3.1 Symbolic Modeling of Program State

In ANGR and most other symbolic execution engines, there is a concept of execution states.

**Definition 1** (Execution State). An execution state \((D, P)\) is a representation of the state of system data \(D\) if a particular execution path \(P\) is taken.

\(D\) includes simple things, like the state of CPU registers and RAM, but also includes more advanced information such as the current environment variables (such as the current directory or user privileges), stdio (the text input and output of a program), the filesystem, etc. In this report, we limit our observations to registers, RAM, and stdio, as they are directly modeled by the symbolic constraints described below.

\(P\) is the path taken to reach the state. In simple terms it is a record of which instruction blocks were executed, in order.
3.1.1 Expressions

The following two definitions establish the most basic types of data present in the ANGR symbolic execution framework. We will use them in a moment to more generally describe what constraints are.

**Definition 2** (Bitvector). A Bitvector is an integer in $\mathbb{Z}_{2^n}$, represented by a sequence of $n$ bits. They behave as one would expect a standard int, long, short, byte, etc. to behave in a C program.

**Definition 3** (Boolean). A Boolean is a value in $\{\text{True}, \text{False}\}$

To describe operations on these values, we require the definition below. Later on, this definition will be key to describing how our symbolic execution state is represented.

**Definition 4** (Expression). Fix a set of function symbols (operators) $\Sigma$, each with a fixed arity (number of operands). $\Sigma$ can include 0-ary operators, sometimes called constants. The operators in $\Sigma$ can return a value that is either a Boolean (e.g. “=” or “<”), or a Bitvector (e.g. “*” or “+”).

An expression $E$ is a term built from $\Sigma$ and variables.

An expression consisting of only a variable or a constant will be called a primitive.

An expression with no variables will be called concrete; an expression with variables will be called symbolic.

The operands of a function term can be referred to as a sub-term.

A constraint is an expression returning a boolean value.

Expressions can be represented with a labeled tree using labels in $\Sigma$, with primitives forming the leaves of the tree. As a trivial example, the expression $(3 +
\( x \times 4 \) could be expressed as shown in figure 3.1 below:

![Figure 3.1: A simple mathematical expression.](image)

In symbolic execution states, constraints are used to describe what values a given part of the state holds. Constraints can be as simple as a single equality operation between a symbol and a constant. For example, a variable \( \text{sym}_x \) might have the constraint \( (\text{sym}_x = 5) \). This constraint establishes that in this execution state, the variable \( \text{sym}_x \) must be exactly equal to 5. However, as mentioned previously, symbolic execution allows one variable to represent many possible values at once. An instance of this is the following constraint: \( (\text{sym}_x > 7) \). This establishes that \( \text{sym}_x \) can hold any integer value, so long as it is greater than 7. Multiple requirements can be established through use of the boolean operators such as \&\& and || (boolean AND and OR, respectively). For instance, to fix \( \text{sym}_x \) to be strictly between 0 and 10, we would use the constraint \( ((\text{sym}_x > 0)\&\&(\text{sym}_x < 10)) \).

### 3.2 Execution Behaviour

The inclusion of expressions in the data of an execution state has a significant effect on what exactly happens when instructions are "executed".
Definition 5 (Concrete). We refer to an expression $E$ as Concrete if either

- $E$ is a Primitive with a constant value (i.e. not symbolic), or
- $E$ is an Operation and all subterms of $E$ are Concrete.

Definition 6 (Symbolic). We refer to an expression $E$ as Symbolic if either

- $E$ is a Primitive representing a symbolic variable (i.e. not constant), or
- $E$ is an Operation and at least one subterm of $E$ is Symbolic.

Remark 1. An expression must be either Concrete or Symbolic.

If every constraint expression in a state is Concrete, then the behavior of that state executed symbolically is identical to the behavior of the program if executed normally. This type of execution (with purely concrete data) is more specifically referred to as concrete execution. The only real difference is that all CPU operations and memory are simulated in software, not hardware.

However, a state in a symbolic executor like ANGR will typically utilize a mix of symbolic data and concrete data. Some examples of how they are built and used are below.

### 3.2.1 Example: Basic Arithmetic

Constraints on memory and files will take the form of Expressions evaluating to Bitvectors. In simple enough functions, they can automatically form a complete model of the behaviour of a function. For instance, observe the C function defined below:
```c
int validate_commute(int x, int y) {
    int r = 0; // 1
    r += x * y; // 2
    r -= y * x; // 3
    return r;
}
```

Figure 3.2: A simple C arithmetic function

Going through line by line, we would get the constraints below, each corresponding to the state after executing the corresponding line(s):

![Constraints diagram](image)

Figure 3.3: Constraints after each line of figure 3.2. Identical subterms are color-coded to show how each line builds upon prior constraints.

Note that each new line’s constraint is formed simply by adding items to the previous line (observe the similarity between the color-coded nodes). Though ANGR
has very trivial simplifications built in, for the most part constraints are generated simply by repeatedly adding new Operation nodes with the previous constraints as a subterm. This is visible in constraint 3, where the entire + tree of constraint 2 is present as a subterm to the − of constraint 3.

However, even though ANGR cannot handle simplification on its own, it is integrated with Microsoft’s Z3 SMT solver (described here [7], [6]). Z3 is primarily used to check the satisfiability of a constraint. A constraint $C$ is satisfiable if and only if there exists a set of values for the symbolic variables in $C$ such that $C$ is true. The satisfiability of an execution state’s constraint determines whether the state is actually reachable. Beyond checking satisfiability, Z3 also has the useful feature that it can emit simplified versions of whatever constraints are fed into it. Using this feature on constraint 3 above, we get the following:

![Figure 3.4: Final simplified constraints of 3.2](image)

Figure 3.4 shows that the return value $r$ will always be zero, since $r$ has the constraint: $r = 0$. This constraint represents every result that validate_commute can produce based on any inputs $s_0$ and $s_1$. This is a fairly trivial example but is useful as a demonstration of how constraints are built up and their implications.
### 3.2.2 Example: Fixed Length Loops

```c
#define SIZE 5
int sum_byte_array(int arr[SIZE]) {
    int x = 0;
    for (int i=0; i<SIZE; i++) {
        x += arr[i];
    }
    return x;
}
```

Figure 3.5: Source code for the function `sum_byte_array`, which sums a fixed-size array with 5 32-bit integer elements.

The function above sums 5 32-bit integers from a fixed length array and returns the result. Note that since the parameter is a fixed size array, there is no risk of early exit due to a memory error or anything of that sort here. As such, the for-loop will never deviate in the number of iterations, since `i` is a concrete, meaning each check of `i < 5` has only one possible result. We will discuss what happens if the condition in the for-loop included symbolic values later on, in section 3.2.3. For now, we observe the following:

Initially, `x` has a fairly simple constraint:

```
=  \\
x  0
```

Figure 3.6: Initial constraint for `sum_byte_array`, reflecting `int x = 0;`
After one loop, we have the following slightly more complicated constraint. Between figures, we have highlighted identical subterms by color-coding the corresponding nodes:

Figure 3.7: Constraint on x after 1 iteration of the loop in sum_byte_array. Extract here represents selectively taking bits 159 through 128 (from right, inclusive) from arr, an operation equivalent to arr[0] in C.

The arr array is treated as a contiguous Bitvector with 5 items * 32 bits per item = 160 total bits. The Extract operation here is equivalent to what would normally be written arr[0] - it retrieves a specific sequence of bits from within a larger Bitvector. Bits as referenced by Extract are inclusive and zero-indexed from right to left. The least significant bit has index 0, and the most significant bit has (in this case) index 159. So, in this case, the Extract node represents the sequence of 32 bits (i.e. one 32-bit integer), from 159 to 128, 0-indexed from the rightmost bit of the rightmost bit of arr.
Figure 3.8: Constraint on $x$ after 2 iterations of the loop in `sum_byte_array`. Subtrees from previous figures color coded.

After two loops, the tree begins to be somewhat hard to parse at a glance. However, we can see that the $+$ subterm highlighted in green is in fact the same $+$ expression as in figure 3.7. The only addition is the $+$ adding the new rightmost `Extract` node, which represents $arr[1]$. In total, this constraint means $x = 0 + arr[0] + arr[1]$ - exactly what we would expect knowing the source code of the function!

Going even further, again marking the previous iteration with a green node, we see a pattern begin to emerge.
Figure 3.9: Constraint on $x$ after 3 iterations of the loop in `sum_byte_array`. Subtrees from previous figures color coded.

Similar to what was somewhat visible in the previous section (3.2.1), each time the line $x += \text{arr}[i]$; is executed, we see that the new constraint contains the old value that $x$ was equal to as a subtree. This is simply explained by the fact that $x += \text{arr}[i]$; is equivalent to $x = x + \text{arr}[i]$; . Since the prior constraint is establishing what $x$ equals, it makes sense that the expression $x + \text{arr}[i]$ would be formed into a constraint by substituting the prior value of $x$. 
Figure 3.10: Constraint on $x$ after 5 iterations of the loop in `sum_byte_array`. Subtrees from previous figures color coded.

Skipping to iteration 5, it becomes very clear that the structure of this loop is repeating. This structural similarity is something that this report attempts to take advantage of to identify loops. We will not go any further with this specific example, as it is already growing large enough that fitting it within the report is difficult. However, this is the constraint on $x$ at the end of our function call and
would become a constraint on the returned value if we were doing broader analyses outside of sum_byte_array.

### 3.2.3 Path Branching

In the above examples, one might note that the examples were constructed in such a way as to avoid all ambiguity as to which way each branch condition would go. This is because symbolic executors behave very uniquely when faced with a branch condition. Since a variable can have numerous possible values, something as simple as checking $if(x > 7)$ can become fairly complicated. When faced with two execution paths that are both viable, how does the symbolic executor decide which to take?

The simple answer is that both paths are taken. Whenever a state encounters a branch, it produces several *Successor* states. Each successor is an exact copy of the *Parent* that created it, except with the addition of a new constraint reflecting why that successor would be chosen over its siblings (e.g. the false case of an if branch). The state is reachable if and only if this constraint evaluates to True. A simple example is shown below:

```c
int cond(int x){
    if  (x > 7) {  // 1
        return 1;  // 2
    }  else {      // 3
        return 0;  // 4
    }
}
```
Running $\text{cond}(s_0)$, where $s_0$ is an unconstrained symbolic variable. The initial state $\text{State}_0$ is then at line 1, with no constraints. With both the true and false case of the if condition satisfiable by $\text{State}_0$, the executor produces two successor states $\text{State}_1$ and $\text{State}_2$ at lines 2 and 4, respectively.

This splitting behaviour repeats for each conditional branch encountered in the program, each one potentially doubling the number of states. This is especially problematic in loop control flow structures (e.g. a for loop), which can conditionally branch an infinite number of times. Research has been done into pruning states if they do not introduce any new behaviour (e.g. an if-statement that merely prints a debug output, then rejoins the main execution path) in [18]. Other reports such as [12] have proposed methods of looking ahead at future branch points condition to find interesting paths. However, there is not yet a general solution to the problem of dealing with branches and loops.

### 3.3 Initial State

Depending on what one is hoping to find via symbolic execution, the choice of the initial state for symbolic execution may vary. The program entry point must be specified, which can be main() if one wishes to see execution of the entire program. Alternatively, one can set the entry point to be a function, to analyze its behaviour in isolation. Arguments and data in memory can be concrete values or symbolic variables, and can be constrained to observe behaviour under specific input conditions. Furthermore, constraints on the state of stdin and arguments to the entry point can be specified, and arbitrary data can be placed in memory.

For our test cases, we observed functions with all arguments set to be symbolic variables. However, wherever stdio was utilized, we ensured that there was sufficient
data to avoid early end-of-file behaviour, for the sake of simplicity. This is not a requirement of our method analysis, but was done for the sake of demonstration, as the constraints can otherwise become extremely complicated to describe, even with visuals.
Chapter 4

Methodology

The previous chapter (3) provided definitions and descriptions from previous research regarding symbolic analysis and expressions. Beginning in this section, the definitions and theorems put forth are unique to the work completed under this MQP. We do not claim that this work is entirely novel - large parts, in particular the pattern mining algorithm, are directly inspired by previous works that are cited where appropriate. However, it is our contribution, and can hopefully be used as a framework for future work related to mining patterns in expression trees, or other similarly structured trees including fixed numbers of children.

4.1 Expression Data Structure

As previously mentioned, expressions (and thus constraints) are a tree of terms, similar to what one would find in a functional programming language akin to LISP. For instance, the expression $x + 4 > y * x + 3$ is better described by the diagram below:
This representation is far more explicit with regards to the order in which operations are computed. However, there are some specific considerations we need to make in encoding our expressions as trees, specifically with regards to making them suitable for finding patterns. After all, it wouldn’t make sense to find a pattern like the below:

Even accepting that the above shows subtraction involving a symbol $x$, it is unclear which side of the operation $x$ is on, or even what $x$ is. In the following section we describe a very strict definition for how to construct an expression that allows strict definitions on number of arguments, argument order, and argument types, while also providing several other benefits with regards to pattern discovery.

4.1.1 Traits

Before going any further, we will introduce the concept of a trait tree node. Unfortunately, this definition depends on the definition of a trait, and vice versa. Definitions
for both terms will be provided in this section.

**Definition 7** (Trait Tree). A **Trait Tree Node** is an unordered set of Trait Instances. Within each trait tree node, there is at most one instance of each trait type.

A root trait tree node defines a **Trait Tree**, in which parent-child relationships between nodes in the tree are represented via progeny items (which we will define momentarily).

From this point forward we will occasionally just say “node” to refer to a trait tree node, or “tree” to refer to a trait tree. As far as what exactly the aforementioned traits and progeny are, see the following definitions:

**Definition 8** (Trait). Consider a tuple \( \langle v_1, v_2, \ldots, v_n \rangle \), where each item \( t_i \) can be of any data type (e.g. an integer, a string, a set, a trait node, etc.).

An item \( v_i \) is referred to as a **core** item if it is not a Trait Tree Node.

An item \( v_i \) is referred to as a **progeny** item if it is a Trait Tree Node.

With that terminology in mind, we can define a Trait Type as three values \([N, C, P]\) where:

- \( N \) is a unique string representing the trait type’s name/unique identifier.

- \( C \) is an integer representing a number of Core items in each instance of this trait type \((C \geq 0)\).

- \( P \) be an integer representing a number of Progeny items in each instance of this trait type \((P \geq 0)\).

Given a trait type \( T = [N, C, P] \), a **Trait Instance** \( t \) of \( T \) is a pair \( t = (N, v_1, v_2, \ldots v_{C+P}) \). \( N \), which we refer to as the **identifier** of \( t \), is used to
identify which trait type any given trait instance represents (since \( N \) is unique to each trait type \( T \)). The values \( v_1, v_2, \ldots \), which we refer to as the items of \( t \), represent the data specific to the trait instance \( t \).

The trait type parameters \( C \) and \( P \) are significant to the items of \( t \) in the following way: For each \( j \in 1..(C - 1) \), every item \( v_j \) is core. For each \( k \in C..\text{Size}(T) \), every item \( v_k \) is progeny.

In essence, a trait type \( T \) serves as a descriptor for the size and content of a trait instance tuple. For instance, given two trait instances \( (N', v_1, v_2, \ldots, v_n) \) and \( (N', w_1, w_2, \ldots, w_m) \), since both have the same identifier \( N \), they must be instances of the same trait type \( T' \). Since all instances of \( T' \) have the same number of items, then we also know that \( n = m \). We further know that for each \( i \in 1..\text{Size}(T) \), \( v_i \) is core \( \iff \) \( w_i \) is core, and \( v_i \) is progeny \( \iff \) \( w_i \) is progeny. In this way, trait types give guarantees about the organization of items in corresponding trait instances.

A useful analogy is to a struct datatype in C, or a class in Java, wherein each instance of that type is guaranteed to have certain member values. Note that we will sometimes refer to “a trait instance \( t \) of trait type \( T \)” as “an instance \( t \) of trait type \( T \).”

For later use, we introduce the following notation to allow more succinct mathematical definitions regarding trait types.

Suppose we have a given trait type \( T = [N, C, P] \).

We write \( \text{Size}(T) = C + P \) to represent the total number of items in any instance of \( T \).

We write \( \text{Name}(T) = N \) to refer to the name of a trait type \( T \). Inversely, we write \( \text{Lookup}(N) = T \) to refer to the trait type named by a given \( N \).

To express whether an item of a trait type is core or progeny, we have the following boolean functions:
• Core\((T, i) \iff i \leq C \iff \text{the } i\text{th item of } T \text{ is core.}\)

• Prog\((T, i) \iff i > C \iff \text{the } i\text{th item of } T \text{ is progeny.}\)

Suppose we have a trait instance \(t = (M, v_1, v_2, \ldots, v_n)\).

We write \(\text{Ident}(t) = M\) to refer to the identifier of a trait instance \(t\).

If \(t\) is an instance of trait type \(T\), we write \(t \sqsubseteq T \iff \text{Lookup}(M) = T\).

Conversely, if \(t\) is not an instance of the trait type \(T\), we write \(t \not\sqsubseteq T \iff \text{Lookup}(M) \neq T\).

We write \(\text{Size}(t) = n\). Note that \(t \sqsubseteq T \implies \text{Size}(t) = \text{Size}(T)\).

We refer to a specific item \(v_i\) of \(t\) as \(t[i]\).

We refer to the set of trait instance identifiers present in a trait node \(E\) via the function \(\text{NodeIdents}(E)\). That is, \(\text{NodeIdents}(E) = \{N \mid t = (N, \ldots), t \in E\}\). This basically just maps each trait instance in a set to its identifier, and is useful for comparing the trait types present in different trait tree nodes.

With the definition of traits established, it should be clearer how trait trees are structured. As defined earlier, each trait node \(E\) in the tree is a set containing trait instances. These trait instances in turn contain trait tree nodes in their progeny items (at least, if they are an instance of a trait type with \(P > 0\)). These nested trait tree nodes are effectively children to \(n\), and can in turn have their own children. And so the tree is formed: not as a graph with explicit edges, but as a nested data structure.

Though trait tree nodes are unordered sets, the requirement that there be at most one instance of each trait type per node means that the set union and intersection operations (\(\cup\) and \(\cap\)) aren’t particularly useful. Instead, we will mostly operate with trait tree nodes by adding new traits instances to them individually.
4.1.2 Expressions as Trait Trees

We believe that trait trees, as a data structure, are applicable to many different contexts. However, within this report we will primarily be focused on using Trait Trees to represent the expressions defined previously in definition 4 of chapter 3, using a specific set of traits chosen for this task. As such, we will henceforth refer to Trait Trees as an Expression Trees, and Trait Tree Nodes as Expression Nodes. They are in fact the same thing, just renamed to be more contextually relevant.

The traits we will be using are below, with examples following shortly after. Recall that the syntax of traits is \((N, C, P)\) where \(N\) is the name of the trait, \(C\) is the number of core items, and \(P\) is the number of progeny items.

- \((\text{Bitvector}, 1, 0)\) is a trait type used to represent a Bitvector-valued expression term. The single core item encodes the number of bits. As an example, a 32 bit valued node would include the trait instance \((\text{Bitvector}, 32)\).

- \((\text{Boolean}, 0, 0)\) is a trait type used to represent a Boolean-valued expression term. It has no items, and instead is meaningful simply by whether or not it is present in an expression node.

- \((\text{Constant}, 1, 0)\) is a trait type used to describe the value of constant term in an expression. The single core value represents the constant value. As an example, an expression term representing an integer constant 5 would include the trait instance \((\text{Constant}, 5)\).

- \((\text{Symbolic}, 1, 0)\) is a trait type used to describe the symbol of a symbolic variable in an expression. The single core value represents the symbolic variables name. As an example, an expression term representing the symbolic variable \(\text{var}_x\) would include the trait \((\text{Symbolic}, \text{"var}_x")\).
(Operation, 1, 0) is a trait type used to represent the function symbol of an expression node. The single core value represents the function symbol. As an example, an expression term representing addition would include the trait instance (Operation, "+").

Describing the arguments to a function is the first place where we use progeny items. Since each trait type has a fixed number of items, we need a trait type for each function arity. We enumerate these trait types as (Args<\(n\), 0, \(n\)), where \(n\) is the arity we want to represent. An instance of this trait will have \(n\) items, each encoding a child trait node, with the order (left, right, etc.) determined by the items position. As an example, an expression term that has two arguments \(L\) and \(R\) would include the trait instance (Args<2>, \(L\), \(R\)).

We write an empty trait tree node as Expr \{\(\emptyset\}\}, although this doesn’t represent a valid expression just yet. To do that, we need to add instances of the traits above, to represent all of the data in said expression. We claim that the above traits can fully represent the underlying semantics of any constraint produced by ANGR.

As a simple example, a node representing the 32 bit value 317, would need:

- A trait establishing that it is a constant value 317
- A trait establishing that it is a 32 bit integer.

which brings us to the node representation Expr\{(Constant, 317), (Bitvector, 32)\}.

As an example of how we would represent a full expression tree of \(3 \times 6\), we have the following:

\[
\begin{align*}
A & = \text{Expr}\{(\text{Constant}, 3), (\text{Bitvector}, 16)\} \\
B & = \text{Expr}\{(\text{Constant}, 6), (\text{Bitvector}, 16)\} \\
M & = \text{Expr}\{(\text{Operation}, "\times"), (\text{Bitvector}, 16), (\text{Args<2>}, A, B)\}
\end{align*}
\]
Though this form is more verbose than a simpler label-based tree, these definitions lend themselves well to pattern discovery as described later. Note that though it isn’t explicitly shown here, traits can have a mixture of core and progeny traits. This will be taken advantage of later, in section 4.4.1.

4.1.3 Benefits of Trait Nodes

Labeled Trees Unsatisfactory

As a first try, one might have tried to store these properties (operation, value, etc.) using a simple labeled tree, wherein each node has a single associated string value (as opposed to the sets of Trait Trees). However, there are several problems with this approach.

First of all, in order to quickly find frequent patterns in a tree, one must be able to generate candidates. With the immense search space that is every 32 or 64-bit integer, it is unlikely that exhaustive enumeration would make it past even a single node.

Even constraining the search space to only numbers that we have already seen while executing isn’t ideal. A simple labeled tree structure fails to take advantage of “type” similarity between nodes. For example, a primitive node representing the constant 32-bit integer 3 intuitively bears more resemblance to a similar primitive node representing 4, than to a + operation node in terms of implications on code structure. Furthermore, if we ever wished to explore new trees (e.g. by exploring new inputs or other binaries), it is unlikely that our existing set of labels/patterns would match up very well.

As a trivial example of how this could go wrong, consider a function that performs a repeated summation on its input variable. Suppose that it was fed the sym-
bolic variable \( x \), then the symbolic variable \( y \). By label matching, \( x + (x + (\ldots)) \ldots \) would not match on \( y + (y + (\ldots)) \ldots \) despite their structural similarity.

**Expression Node Type Complexity**

As previously established, there are Primitive and Operation nodes. But within those two broader categories there are further distinctions. Primitives can be symbolic variables or concrete constants. They can also be either a Boolean or a Bitvector value, and even between Bitvectors there are 32, 64, 128, etc. bit values.

Similarly, different Operation nodes will support different numbers of arguments. They will furthermore take and return different types of data (Boolean, Bitvector, etc.).

One quickly realizes that even though there are a multitude of unique types, most are a combination of a few unique ideas. For instance, the constant values 200 and 300, are both Bitvectors (of, say, 32 bits) and each is Constant. The only distinction between them is in the value of the constant.

As touched upon in 4.1.3, failure to account for similarity in types is a major weakness of a labeled tree system. The sheer number of types is another potential pitfall, as having to manually create them would hamper the usability of any algorithm. Finally, even between types there are varying levels of similarity - the difference between the addition of two 32 bit numbers and two 64 bit numbers is fairly minor.

The trait system neatly ties up both of these by expressing all possible combinations of types. One can draw a rough parallel to the concept of “product types” of type theory, with each Trait representing a type, to encode the Product Type of a node.

Furthermore, trees in which with each node being a set have fortunately already
had some research done on them, even with quantitative values, that we were able to take inspiration from.

**Edge ordering**

A simple labeled tree using edges to encode operation children does not account for ordering. An example of when this is of critical importance is the division operator 
\(/\). Without a distinct demarcation of which child edge points to which argument (left or right), the tree is left ambiguous as to which child represents the numerator, and which the denominator.

Even if the order between child nodes is established, there is still be the issue of number of children. As mentioned later in the Pattern 4.4, in order to create templates from patterns we need to strictly define the number, parents, and order of the children to the pattern. Even though this is only strictly a requirement to perform collapsing, it is also a useful tool for ensuring our expression trees (and candidates for them) are sensibly structured. After all, it would not make sense to have a subtraction node with 3 children!

### 4.2 Pattern Data Structure

In the general context of Frequent Pattern Mining, a pattern is anything that can “occur” in a graph, in a similar way to how a regular expression can match on a string. Like the wildcard expressions found in most regex engines, our patterns are able to represent when *any* value will be accepted in a certain position. However, depending on what the position is (e.g. a core item vs a progeny item) the behavior of what a wildcard could or could not be differs. From this distinction we arrive at the following definitions:
**Definition 9** (Pattern Candidate). A *Pattern Candidate* is either a wildcard (⋆) or a *Pattern Node*.

**Definition 10** (Pattern Node). A *Pattern Node* is structurally identical to an Expression Node (or, equivalently, a Trait Tree Node), with one key difference in what items are found in its trait instances:

- Each core item is either a non-node value, or ⋆.
- Each progeny item is a Pattern Candidate.

In other words, each trait instance item of a Pattern Node or its descendants can optionally be ⋆ instead of a specific value.

Pattern Nodes are constructed similarly to basic Nodes, except with the introduction of ⋆. This change technically alters the definition of our trait types, however, we assume that for any trait type $T$, there is a roughly equivalent pattern trait type $T'$ that allows for wildcards, with the same identifier $N$. In practice, this means that we don’t really need to worry about a separate set of types between pattern nodes and expressions, provided we are explicit in what we are referring to in a given case. To that end, we distinguish expression nodes and pattern nodes by writing them as $\text{Expr}\{t_1, t_2, \ldots\}$ and $\text{Patt}\{t'_1, t'_2, \ldots\}$, respectively.

Pattern nodes and traits are otherwise identical, and in most cases we will interact with them in the exact same way. The introduction of ⋆ simply lets us be explicit about when an item is currently not specified by the pattern.

In the following example, we see a wildcard in place of a concrete core value in the Constant trait of the pattern node $A$:
This pattern matches any expression representing multiplication of the symbolic variable “x” by any constant.

In another the following example, we see a wildcard in place of a progeny value of the Args<2>trait of D:

This represents multiplying any sub-term by 3. The key distinction between this pattern D and the previous pattern M is that the wildcard in D would match on any child tree, since it is a wildcard on a progeny item. The wildcard in M would only allow for matches that varied in the specific constant value on the right hand-side of the ∗, since it is a core wildcard in the constant trait instance. This distinction in types of wildcards corresponds to the fact that Constant’s only item is core, while Args<2>’s items are progeny.

The most broad possible Pattern Candidate (e.g. the one that would match the most possible Expressions) is simply ∗. This candidate which matches any expression tree. However, such a vague pattern is somewhat useless to us, since by matching everything it essentially tells us nothing.

4.2.1 Match Conditions

As of yet we haven’t formally described what it even means for a pattern candidate to match an expression. The answer is not immediately obvious, due to the fact that Expressions have properties of both trees and sets. These definitions cover
different types of matching that are of interest to us, and are partially inspired by the definitions in the report “Frequent Pattern Mining in Attribute Trees” [13], where “attribute trees” are trees wherein each node is a set. However, unlike in that report, our definitions are very strict with regards to the structure of the pattern vs. the expression.

**Definition 11 (Support).** A pattern candidate $P$ **supports** an expression node $E$ (written $P \vdash E$) if and only if the following statement is true:

For all $t_P \in P$:

- NodeIdsents($P$) $\subseteq$ NodeIdsents($E$)
- $\forall t_P \in P$, $t_E \in E | Ident(t_P) = Ident(t_E) :
  - \forall i : Core(t, i) \& t_P[i] \neq \star \implies t_P[i] = t_E[i]
  - \forall i : Prog(t, i) \& t_P[i] \neq \star \implies t_P[i] \vdash t_E[i]$

In simpler terms, a pattern $P$ supports an expression node $E$ if $E$ has every trait in $P$, each core item in $E$ is equal to the corresponding item in $P$ (if exists and not $\star$), and each progeny item in $E$ is supported by the corresponding item in $P$ (if exists and not $\star$). Support on an expression is akin to being a sub-tree of that expression, except with missing children formalized as wildcards.

**Definition 12 (Capturing).** A pattern candidate $P$ **captures** an expression node $E$ (written $P \models E$) if and only if the following conditions are satisfied:

$\forall$ traits $T$:

- NodeIdsents($P$) = NodeIdsents($E$)
- $\forall t_P \in P$, $t_E \in E | Ident(t_P) = Ident(t_E) :$

33
∀i : Core(t, i) & t_P[i] ≠ ⋆ ⇒ t_P[i] = t_E[i]
∀i : Prog(t, i) & t_P[i] ≠ ⋆ ⇒ t_P[i] |= t_E[i]

The definition for Capturing is very similar to Support, but with the key difference that capturing **does not** allow for traits present in E to be missing in P. Capturing, on the other hand, requires the same set of trait types to be present in the expression and pattern.

However, since the definitions are so clearly similar, we easily arrive at the following proposition:

**Theorem 4.2.1.** If P is a pattern candidate and E an expression node,

- P \models E \implies P \vdash E
- P \not\models E \implies P \not\vdash E

In essence, capturing is a more strict relation than support for matching expression nodes.

**Proof.** As mentioned above, the only difference is that capturing requires the exact same set of traits between E and P. Let S_E and S_P refer to the trait types present in E and P, respectively.

\[
P \models E \implies S_E = S_P \implies S_P \subset S_E.\]

Since the conditions one core values are exactly the same between our definitions, this is sufficient proof.

P \not\models E can imply several things, depending on which conditions of support are failed. For simplicity’s sake let us assume that the “progeny of P support corresponding progeny of E” is not the failed requirement - if it is, then this argument would still hold if applied to that child instead. With that consideration in mind, we know that either
• The trait matching requirement of support failed, implying \( S_P \not\subset S_E \implies S_P \neq S_E \implies P \not\models E \).

• The core item matching requirement of support failed. Since this requirement is equivalent between our two definitions, it is clear that failing this condition for support would also fail for capturing.

Describing Matches

As of yet we don’t have a formal way of describing “how” a match (supporting or capturing) is formed. Since patterns can match more than one expression, it is useful to establish an explicit relationship describing how the match works out. The following definition provides a method of distinguishing how different expression nodes are supported/captured differently with regards to satisfaction of wildcard items.

\textbf{Definition 13} (Match Function). A \textbf{Match Function} \( M \) is a way of specifying exactly how an expression node \( E \) satisfies the support or capturing condition of a pattern candidate \( P \).

We denote the support match function as \( M_{\text{Supp}}(P, E) : W \rightarrow V \)

We denote the capturing match function as \( M_{\text{Capt}}(P, E) : W \rightarrow V \)

(Where \( W \) is the set of all \( \ast \) present in \( P \) and \( V \) is the set of values that a core trait item can hold, which depends on the use case but here is either a string or an integer).

Essentially, if \( P \) supports/captures \( E \), then there will exist a function \( M \) that maps each wildcard item in \( P \) to the item in \( E \) that “fills” it.

Though the definition may be verbose, here’s a trivial example.
We see that $P \vdash A, B$, and $P \models A$, but $P \not\models B$ since $B$ has an extra trait “OtherTrait”, and thus $\text{NodeIds}(B) \neq \text{NodeIds}(A)$.

Therefore, we only have the following three match functions:

1. $M_{\text{Supp.}} \{P, A\} : \star_1 \rightarrow 27, \star_2 \rightarrow 16$
2. $M_{\text{Supp.}} \{P, B\} : \star_1 \rightarrow 65, \star_2 \rightarrow 32$
3. $M_{\text{Capt.}} \{P, A\} : \star_1 \rightarrow 27, \star_2 \rightarrow 16$

**Remark 2.** We note that 2 and 3 have identical value mappings. Since all capturing offers over support is a stricter requirement on traits, there is no reason that these functions would ever be distinct.

However, the separate $M_{\text{Supp.}} \{P, E\}$ and $M_{\text{Capt.}} \{P, E\}$ labels can be useful for discussing the existence of the function, as an implicit method of implying $P \vdash E$ or $P \models E$, respectively.

For demonstration purposes, we have distinguished the wildcards in the above example by applying subscripts to them. Note that in the data structure, wildcards are not given a unique identifier like this. They are instead referred to by position, as described later in section 4.3.3

### 4.3 Pattern Mining Algorithm

Mining frequent subtrees is a fairly well studied problem. At the core of most variants on the process is the gSPAN algorithm, put forth in [21]. This algorithm
relies on the following principle: If $P$ and $P'$ are pattern candidates, and $P'$ is a more strict pattern than $P$ (e.g. $P'$ might have additional trait requirements), $P'$ will support only a subset of the nodes of $P$. It follows that we can ignore computing support on $P'$ if we already know $P$ does not have enough support to warrant investigation. In general, the more we can avoid computing support (which is expensive) without any loss of soundness, the better. We will formally show that our patterns satisfy this property in this section.

4.3.1 Evaluating Candidates

We must establish an empirical metric to describe the frequency of a pattern. The most common metrics for measuring the performance of a pattern candidate $P$ in FPM are:

- Percent of all nodes $N$ s.t. $P \vDash N$
- Percent of all trees (if target dataset includes multiple) s.t. containing at least one node $N$ s.t. $P \vDash N$

Neither of these are really what we’re looking for, as we eventually will care much more about capturing, and some specific criteria of the patterns themselves. However, since support is a prerequisite for capturing, we will for the time being use 4.3.1 for picking out promising candidates.

4.3.2 Extension

**Definition 14** (Extension). To **Extend** an existing Pattern Candidate $P$ is to do exactly one of the following actions:

1. Add a new trait instance $t$ to $P$, where $t[i] = \star \ \forall \ i \in 1..\text{Size}(t)$ (i.e. $t$ is
2. Replace a core \( \star \) item with a non-wildcard core value.

3. Replace a progeny \( \star \) item with an empty Pattern Node (i.e. \( \text{Patt}\{\emptyset}\) ).

4. Perform extension on a non-\( \star \) progeny Pattern Node item.

**Definition 15** (Descendants). Given two pattern candidates \( A \) and \( B \), \( B \) is a descendant of \( A \) (written \( B \succ A \)) \iff \( B \) can be created by any non-zero amount of Extension steps on \( A \). We refer to \( A \) as an ancestor of \( B \).

Extension is simply an explicit definition of all of the ways that we can make a pattern node more strict. Each of the 4 types of extension serve only to make a pattern more specific, by eliminating \( \star \) values or by adding additional trait requirements. The descendant relation is a useful way to describe how we reach one pattern by extending another. From these definition, we arrive at the following set of theorems:

**Theorem 4.3.1** (Transitivity). With three candidates \( P_a, P_b, P_c \),

\[ P_b \succ P_a \ \& \ P_c \succ P_b \implies P_c \succ P_a \]

**Proof of Theorem 4.3.1.** We do not yet have a formal definition for extensions. A notation will be formally introduced in section 20, but for the sake of understandability we’ll stick to more general definitions for now.

Suppose that \( S_{ab} = E_{ab.1}, E_{ab.2}, E_{ab.3}, \ldots , E_{ab.n} \) is the set of extensions required to form \( P_b \) by extending \( P_a \).

Suppose also that \( S_{bc} = E_{bc.1}, E_{bc.2}, E_{bc.3}, \ldots , E_{bc.m} \) is the set of extensions required to form \( P_c \) by extending \( P_b \). It follows then that by applying the exten-
Lemma 4.3.2. Any pattern node $P_s$ with no non-wildcard progeny items can be formed by repeated extension of $\star$.

Proof of Lemma 4.3.2. Showing this is fairly straightforward. Let $P'_s = \star$ be our working pattern, which we will build up to be equivalent to $P_s$ by extension. The first step is of course to extend $P'_s$ to replace $\star$ with an empty pattern node (an extension of type 3), resulting in $P'_s = \text{Patt}\{\emptyset\}$.

We then extend $P'_s$ by adding an all-wildcard stub of each of the trait types in $\text{NodeIdents}(P_s)$ (a series of extensions of type 1). Now we have that $\text{NodeIdents}(P_s) = \text{NodeIdents}(P'_s)$.

Finally, we need to make the core items match up between $P_s$ and $P'_s$. We do not care for progeny, because $P_s$ has only $\star$ children (by merit of our lemma’s conditions), and $P'_s$ has only $\star$ children (by merit of extension-stubbed traits having all items be initially $\star$). For each non-$\star$ core item in the traits of $P_s$, we extend $P'_s$ to have an equivalent core item in the corresponding position (an extension of type 2). After this, $P'_s = P_s$. \qed

Theorem 4.3.3 (Completeness). Any possible pattern candidate $P$ can be generated by repeated extension of $\star$. That is, $P \triangleright \star \forall P$.

Proof of Theorem 4.3.3. This result stems from an extension of the method and result supplied in lemma 4.3.2. All that remains is to show that we can do the same to a pattern candidate $P$ with non-$\star$ progeny. We begin by constructing a pattern $P'$ by the same process as outlined in the proof of 4.3.2. Now, $P'$ and $P$ should differ only in that where $P'$ only has $\star$ progeny children, $P$ has a
mixture of $\star$ and pattern node (i.e. non-$\star$) progeny children.

To complete the construction, we go through each corresponding pair of $(P_c, P'_c)$, and extend $P'_c$ to match $P_c$ by the same processes that we used to extend $P'$ to match $P$. This process repeats recursively until we eventually reach pattern nodes with either no progeny children or only $\star$ progeny children, at which point 4.3.2 satisfies completeness. In essence, we just recursively match traits and core items until there is nothing left to do.

\[\square\]

**Theorem 4.3.4** (Descending Support). *Given any two pattern candidates $A$, $B$, such that $B \succ A$,*

- $B \vdash E \implies A \vdash E$
- $A \not\vdash E \implies B \not\vdash E$

*Proof of Theorem 4.3.4.* Under the definition of $\succ$, $B$ was formed by one or more extensions of $A$.

Each extension step either replaced a $\star$ within $B$ with a more specific value, or added another trait somewhere in $B$. In either case, the criteria for $B \vdash E$ are unchanged except that there is an added condition that $E$ must match on that specific item (where it did not before), or include that trait (where it did not before). As such, extension can only make a pattern’s requirements more strict, and thus can only reduce the set of nodes supported by a pattern.

\[\square\]

From the above, we have the following result, which follows naturally from the fact that descendants can never support that which their ancestors did not.
**Corollary 4.3.4.1** (Support Set Monotonicity). *Given two pattern candidates $A$, $B$ and the sets $S_A$, $S_B$ where*

- $S_A = \{ E \text{ s.t. } A \vdash E \}$
- $S_B = \{ E \text{ s.t. } B \vdash E \}$

*The following must be true: $B \succ A \implies S_B \subseteq S_A$.*

**Proof of Corollary 4.3.4.1.** This follows from the theorem 4.3.4, simply applied to each element of a the set.

For every $E \in S_B \implies B \vdash E$, by 4.3.4, $A \vdash E \implies E \in S_A$, and thus $S_B \subseteq S_A$. However, the reverse of this is not generally true, as there exist $E$ such that $E \in S_A$ & $E \notin S_B$. A simple example of this is if we let pattern $A = \star$, and $B = \text{Patt}\{\text{SomeTrait()}\}$. Letting $E = \text{Expr}\{\emptyset\}$ we see that $A \vdash E$ but $B \not\vdash E$, and thus $E \in A$ & $E \notin B \implies A \not\subseteq B$. 

From this we have that a pattern’s support in a set is monotonically decreasing as the pattern is extended. This is the key to gSpan’s pruning optimizations. However, there is an important caveat here. Though support is monotonic, capturing is not. The extension of a pattern to include a missing trait can very suddenly make a pattern that captured nothing capture many things.

### 4.3.3 Enumeration

Our mining algorithm works via enumeration of pattern candidates. **Enumeration** is the process of using extension in a well-defined order to generate patterns without duplication. Enumeration proceeds in a depth-first pattern, wherein each new pattern is generated by extending a previous one in a Depth-First-Search style...
pattern. From the monotonic property of extension established in 4.3.4.1, we know that all candidates that can be generated by extending the “current” candidate will have less support than their ancestor. This allows us to quickly prune patterns that are unlikely to be useful to us due to low or nonexistent support.

A key requirement of enumeration is ensuring that exploration of candidates does not ever repeat work. Though repeated work won’t break the behavior of the algorithm, it will slow it down while not providing any advantage. As such, we must establish a formal method for traversing the infinite set of possible pattern candidates. But first, some formalities:

**Trait Order**

In order to establish an ordering on how to enumerate pattern candidates, we need a total ordering on the items that make up a pattern candidate!

**Definition 16** (Trait Lexicographic Order). We say that for two traits $T_A$ and $T_B$, $T_A$ is **lexicographically greater than** $T_B$ if the string name of $T_A$ is lexicographically greater than that of $T_B$ lexicographic order.

In other words, $T_A > T_B \iff \text{Name}(T_A) > \text{Name}(T_B)$.

This establishes a (somewhat arbitrary) total ordering on traits and is useful to clarifying order of enumeration.

**Remark 3.** We do not need to concern ourselves with ambiguity on equality of strings due to the fact that traits are guaranteed to have unique names.

**Comparing Lists and Tuples**

Before going any further, to save time we will establish a total ordering on lists. This is to make comparing value pairs such as $(a, b)$ and $c, d$ simpler in future proofs, and
is additionally applicable to lists of varying lengths.

**Definition 17** (Sequence total ordering). *Given two sequences* \( a = (a_1, a_2, \ldots, a_n) \) *and* \( b = (b_1, b_2, \ldots, b_m) \), *We say that* \( a = b \) *if* \( n = m \) *and* \( a_i = b_i \forall i \in 1..n \).

*We say that* \( a < b \) *if either*

- \( a_i = b_i \forall i \in 1..n \) & \( n < m \), *or*

- \( \exists i \mid i \leq m \) & \( i \leq n \) *such that*
  - \( a_j = b_j \forall j \in 1..i - 1 \), *and*
  - \( a_i < b_i \)

Some simple examples are provided for clarity:

- \( ("a", "b", "c") = ("a", "b", "c") \)
- \( (1, 2, 3) < (1, 2, 3, 4) \)
- \( (2, 1, 6) < (3) \)
- \( () < (1) \)
- \( ("a", 2) < ("b", 2) \)

This will greatly simplify the process of establishing a total ordering for the types defined below.

**Directives**

It is useful to be able to refer to a specific item inside a pattern candidate, such as when specifying a match function. Though we have previously used sub-scripted wildcards (e.g. \( *_1, *_2 \), etc.), these aren’t really viable for auto-generating patterns, or nesting them. For this we have the following definition:
Definition 18 (Directive-Step). A **Directive-Step** is pair \( S = (T, i) \) used to refer to the \( i \)th item of instance of a trait \( T \). Directives can be used with reference to both pattern candidates or expression nodes, with the syntax \( P[S] \) or \( E[S] \), respectively.

Given two directives \( S_A = (T_A, i_A) \) and \( S_B = (T_B, i_B) \), \( S_A < S_B \iff (T_A, i_A) < (T_B, i_B) \) by the ordering defined in 17.

As the name implies, directive steps are a part of something larger:

Definition 19 (Directive). A **Directive** is a list of directive steps:
\[
D = (S_0, S_1, \ldots, S_n) \text{ s.t. } n \geq 0
\]
representing an item somewhere in an Expression node or Pattern candidate.

Similar to directive steps indexing operation, we write \( P[D] = P[S_0][S_1][\ldots][S_n] \) to refer to the item within a pattern candidate \( P \) reached by taking the sequence of steps \( S_0 \) through \( S_{n-1} \). Operation is identical for expression nodes.

It is necessary that all directive steps up to \( S_{n-1} \) return pattern nodes or expression nodes as appropriate, otherwise the next lookup will fail. In formal terms:
\[
\forall j \in 0..n-1, \ S_j = (T_j, i_j) \text{ satisfies } Core(T_j, i_j).
\]

The behavior of the empty directive is effectively a no-op:
\[
D = [] \iff P[D] = P.
\]

Directives are totally ordered according to the definition described in 17.

Directives are a formal way to describe the location of an item in a node. As a simple example, consider the following pattern candidate:
This candidate has three wildcards: one in $A$’s symbolic trait, one in $B$’s constant trait, and one in $P$’s operation trait. Unlike before, we haven’t given them “names” in the form of subscripts - doing so was useful for the sake of simplicity at the time, but isn’t viable long term. There are too many possibilities of name conflicts between wildcards, or getting two identical patterns that differ only by their wildcard names (which inhibits our ability to cache matches). Furthermore, even though here we have explicit names for both of $P$’s children, in most cases we will only have an explicit reference to the root of the pattern (in this case $P$). So, referring to wildcards as “the one in $A$” isn’t really viable for describing our algorithm.

Instead, we refer to them by their directives:

- The $\star$ in $A$ would be referred to as $P[D_A]$, where $D_A = ((\text{BinArgs}, 1), (\text{Symbolic}, 1))$.
- The $\star$ in $B$ would be referred to as $P[D_B]$, where $D_B = ((\text{BinArgs}, 2), (\text{Constant}, 1))$.
- The $\star$ in $P$ would be referred to as $P[D_R]$, where $D_R = ((\text{Operation}, 1))$.

### 4.3.4 Canonical Forms

Until now we have simply described extensions verbally. Here we will provide a formal definition representing extensions.
Definition 20 (Extension Formal Syntax). Any extension operation on a pattern \( P \) can be described as a couplet \( C = (D, A) \), where \( D \) is a directive and \( A \) is an action of one of the following types:

1. \( \text{AddTrait}(T) \): At \( D \), add an instance of Trait \( T \), with all items of the instance wildcard.

2. \( \text{StubProgeny}(T, i) \): At \( D \), replace ⋆ progeny item \( i \) of trait \( T \) with an empty pattern node.

3. \( \text{PutCore}(T, i, v) \): At \( D \), replace ⋆ core item \( i \) of trait \( T \) with \( v \).

\( D \) describes where within \( P \) we are performing the extension, and \( V \) describes what sort of extension we are performing of the types described previously.

When comparing two actions of different types, the order is based on where each items type falls in the above listing. For instance, an \( \text{AddTrait} \) would always be less than an \( \text{PutCore} \).

If \( A_1 \) and \( A_2 \) are of the same action type, then we proceed as follows:

1. If \( A_1 = \text{AddTrait}(T_1) \) and \( A_2 = \text{AddTrait}(T_2) \), \( A_1 < A_2 \iff T_1 < T_2 \) by the ordering on traits defined in 16.

2. If \( A_1 = \text{StubProgeny}(T_1, i_1) \) and \( A_2 = \text{StubProgeny}(T_2, i_2) \), \( A_1 < A_2 \iff (T_1, i_1) < (T_2, i_2) \)

3. If \( A_1 = \text{PutCore}(T_1, i_1, v_1) \) and \( A_2 = \text{PutCore}(T_2, i_2, v_2) \), \( A_1 < A_2 \iff (T_1, i_1, v_1) < (T_2, i_2, v_2) \). We assume the values in the tree to have a total ordering, even if it is as arbitrary as an ordered list of possible values where index determines order.
We call this pair of values an **Extension Couplet**.

To compare two couplets: \( C_1 = (D_1, A_1) < C_2 = (D_2, A_2) \iff (D_1, A_1) < (D_2, A_2) \).

We write \( P \oplus C \) to represent a pattern candidate \( P \) extended by an extension couplet \( C \).

For brevity, we use the following abbreviation for multiple extensions:
\[
P \bigoplus (C_1, C_2, \ldots, C_n) = P \bigoplus C_1 \bigoplus C_2 \ldots \bigoplus C_n.
\]

As an example of how one might use couplets to modify a pattern, take the pattern candidate \( P_0 \) below:
\[
P_0 = \text{Patt}\{\text{Args<1>}(\star)}\}
\]

To replace the argument wildcard with a stubbed pattern node value, as an example of 2), we would use the extension couplet \( C_1 \) defined below:
\[
C_1 = (D_1, A_1) \text{ s.t. } D_1 = () \& A_1 = \text{StubProgeny(Args<1>,1)}.\]

Applying \( C_1 \) to \( P_0 \) results in the new candidate \( P_1 \):
\[
P_1 = \text{Patt}\{\text{Args<1>}(\text{Patt}\{\varnothing})}\}
\]

To add a new trait (e.g. Constant) to the argument node, as an example of 1, we use the extension couplet \( C_2 \) defined below:
\[
C_2 = (D_2, A_2) \text{ s.t. } D_2 = ((\text{Args<1>},1)), A_2 = \text{AddTrait(Constant)}.\]

Applying \( C_2 \) to \( P_1 \) results in the new candidate \( P_2 \):
\[
P_2 = \text{Patt}\{\text{Args<1>}(\text{Patt}\{\text{Constant}(\star)})}\}
\]

To give the constant a specific value (e.g. 17), as an example of 3, we can use the extension couplet \( C_3 \) defined below:
\[
C_3 = (D_3, A_3) \text{ s.t. } D_3 = ((\text{Args<1>},1)) \& A_3 = \text{PutCore(Constant,1,17)}.\]

Applying \( C_3 \) to \( P_2 \) results in the new candidate \( P_3 \):
\[
P_3 = \text{Patt}\{\text{Args<1>}(\text{Patt}\{\text{Constant(17)})}\}
\]

Some things to note here. First of all, since all of the operations were extensions,
we know that $P_0 < P_1 < P_2 < P_3$. Second, we can describe each of $P_1, P_2, P_3$ as a list of extensions on $P_0$:

- $P_1 = P_0 \oplus (C_1)$
- $P_2 = P_0 \oplus (C_1, C_2)$
- $P_3 = P_0 \oplus (C_1, C_2, C_3)$

We formalize this way of describing nodes by repeated extensions under the following definition.

**Definition 21** (List Form). Let $L = (C_1, C_2, \ldots, C_n)$, where all $C_i$ are extension couplets. We refer to $L$ as a List Form of a pattern node $P$ if $\text{Patt}\{\emptyset\} \oplus L = P$.

In fact, from of our prior observations on how one can form any pattern via repeated extensions on $\star$, we can observe every pattern candidate except for $\star$ can be represented as a list of extension couplets of this form!

**Theorem 4.3.5** (List Form Completeness). Any non-$\star$ pattern candidate $P$ can be written as a list form.

Note that this excludes the $\star$ pattern candidate because the simplest list form $()$ forms the pattern $\text{Patt}\{\emptyset\} \oplus () = \text{Patt}\{\emptyset\}$. There is no way to “go back” to $\star$.

**Proof of Theorem 4.3.5.** Since the list form syntax fully describes any sequence of extensions, this theorem follows naturally from the completeness theorem 4.3.3, which states that any node can be formed by repeated extensions to $\star$. 48
More specifically, the proof of that theorem established that for any pattern $P$, $P \succ *$. We hope to establish a similar result for $\text{Patt}\{\emptyset\}$.

\begin{corollary}
The extension-couplet elements of a list form $L$ of a pattern $P$ are unique.
\end{corollary}

\begin{proof}
Suppose $L = (C_1, C_2, \ldots, C_n)$. Let $P_i = P \oplus (C_1, C_2, \ldots, C_i)$, so $P_0 = \text{Patt}\{\emptyset\}$, etc. Suppose $\exists i < j$ s.t. $C_i = C_j = (D, V)$. Recall that extensions can replace $*$ with a specific non-$*$ value, or they can add a previously non-existent trait to some place within a pattern. Following this thread, $C_j$ is either trying to replace a wildcard that no longer exists for any of $P_i, P_{i+1}, \ldots, P_{j-1}$, or $C_j$ is trying to add a trait that already exists for all $P_i, P_{i+1}, \ldots, P_{j-1}$. Thus, the extension described by $C_j$ cannot be performed, and so the form $L$ is invalid.
\end{proof}

The above corollary formalizes the fact that extensions cannot be repeated on the same pattern node, since applying an extension couplet breaks conditions that would allow it to be applied a second time. For instance, a couplet $(D, \text{AddTrait}(T))$ will on its first application to some $P$ add a trait of type $T$ to $P$. If we attempt to do this a second time, it will fail because a trait of type $T$ already exists in $P$, and a node can only have at most one trait of each type. A similar effect happens for $\text{PutCore}$ and $\text{StubProgeny}$, which both replace the required $*$ on the first application, causing the second application to fail.

From the uniqueness property of canonical forms, we have a formal guarantee that our exploration will not repeat work (by examining the same pattern twice), so long as we take care to only ever generate canonical forms. An algorithm that generates the tree of all canonical forms in the following section below:
Enumeration Order Algorithm

Typical FSM techniques operate via "Rightmost path extension", whereby each newly enumerated candidate is created simply by taking a previous candidate and adding new elements to it "to the right" of any previous additions. "To the right" here relies on a definition of a path with a total ordering, as we have just established. Since each tree is then described as an ordered list of paths, duplication is impossible, by the following definition and theorem.

**Definition 22 (Canonical Form).** If an extension list $L$ describing a pattern candidate $P$ is defined as

$$L = (C_1, C_2, \ldots, C_n)$$

where $C_1 < C_2 < \cdots < C_n$, then we refer to $L$ as the **Canonical form** of $P$.

The canonical form of a pattern is equivalent to a list of rightmost path extensions, as seen in other pattern mining algorithms such as gSpan [21]. Rightmost path extensions are a critical part of gSpans operation, and since we base our algorithm heavily off of gSpan it is valuable to establish this similarity formally. To that end we have the following theorem.

**Theorem 4.3.6.** The canonical form of a pattern candidate is unique.

*Proof.* Let us temporarily introduce the concept of the **complexity** of a candidate. The complexity of a candidate $P$ is the total number of traits and non-wildcard trait items in $P$. Note that we include the root pattern node, if one exists, as a non-wildcard trait item. So, for example, $\star$ has a complexity of 0, $\text{Patt}\{\emptyset\}$ has a complexity of 1, $\text{Patt}\{\text{Constant}(\star)\}$ has a complexity of 2, and $\text{Patt}\{\text{Constant}(66)\}$ has a complexity of 3, and so on.
**Lemma 4.3.7.** The complexity of $P$ is equal to the number of extensions required to form $P$.

From this lemma we easily see the following corollary by equating the number of extension couplets to the number of extensions:

**Lemma 4.3.8.** If $L$ is a list of $n$ extension couplets representing a pattern candidate $P$, then $n$ is equal to the complexity of $P$.

The above follows naturally from previous results about list forms, and from corollary 4.3.5.1 guaranteeing that we will never have repeated elements that would mess with our metric of complexity.

Finally, since the elements of $L$ are extension couplets, which have a total ordering, we assert that there is at most one way to sort $L$ such that $C_1 < C_2 < \cdots < C_n$. This sorted $L$ is the canonical form of $P$.

With the concept of canonical forms established, the algorithm for enumerating them as a tree of descendants is fairly straightforward, as it follows fairly naturally from the ordering we defined on extension couplets.

We begin constructing the tree with the root $R = \text{Patt}\{\emptyset\} = ()$. Since there is only one possible ordering of $()$, it is a canonical form.

Each child of a node $P = (C_1, C_2, \ldots, C_n)$ in the tree is formed by extending $P$ with a new extension couplet $C_{n+1} > C_n$, forming $P' = (C_1, C_2, \ldots, C_n, C_{n+1})$. If $P$ was a canonical form, we have that $C_1 < C_2 < \cdots < C_n$, and $C_n < C_{n+1}$, and so $P'$ is also a canonical form. Since our original node $R$ in the tree was a canonical form, by induction every node in the tree is a canonical form.

In Algorithm 1 below, we have pseudo-code for a function describing how to generate the children of a node $P$. In the definitions of `DirectiveSuccessors` and `ActionSuccessors`, we omitted specific code as we did not wish to become overly

51
bogged down in the specific details of how to represent an extension couplet in code. However, the process of enumerating over each follows directly from the definition on their total ordering. Directives are slightly tricky for the reason that not only can a directive $D = (S_0, S_1, \ldots, S_n)$ become longer (e.g. $(S_0, S_1, \ldots, S_n, S_{n+1})$), it can also become shorter (e.g. $(S_0, S_x) \mid S_x > S_1$). Going too deep into how to generate these strays too far into the specifics of implementation.

**Remark 4.** A curious thing about the design of this algorithm is that it fails to enumerate core values as well. The main reason is that replacing core $*$ values with specific values will never improve the support/capturing potential of a pattern. Doing so only serves to make the pattern more specific, which can be useful for analysis but not for pattern mining. As such, core values are handled as a separate step later in Section 4.4.2.

### 4.3.5 Pattern Mining Algorithm

Now that we have established a generator function for the (infinite) set of canonical forms, we can begin actually mining them. Many algorithms such as gSpan are able to efficiently find the best patterns with a given level of support via a Depth First Search, pruning any subtree that lacks sufficient support. Our algorithm will do something similar, but with some caveats.

First, and most significantly, we will *not* be iterating over candidates in a Depth-First-Search fashion. Instead, we will proceed with a stochastic method similar to the one proposed in the “Hierarchical Pattern Discovery in Graphs” in [10]. In that report’s algorithm, a population of existing candidates were randomly extended for a period of time, and at the end of the time period the best result was yielded. We will combine this random-extension approach with the pruning techniques of gSpan. Instead of yielding one pattern at the end, we will continuously yield any
Algorithm EnumerateChildren(curr : CanonForm or None):
1  successors ←[];
2  if curr! = None then
3      // If extending an existing couplet, extend using
4      // couplets greater than the last couplet.
5      lastCouplet ←curr[-1];
6  end
else
7      // If starting from scratch, extend from a null-couplet.
8      lastCouplet ←(() , None);
9      // Empty Directive, no action
end
10  foreach newCouplet in ExtensionCoupletSuccessors(lastCouplet, curr) do
11     /* By nature of ExtensionCoupletSuccessors(), these
12     should all be ```greater than``` the current last
13     element of curr, meaning that the result is also
14     canonical. */
15     successors.append(curr ⊕ newCouplet);
16  end
17  return successors ;

Generator DirectiveSuccessors(d : Directive, P : CanonForm):
18     // In order, yield each directive d′ > d that points to a
19     // valid item in P
20  foreach d′ | d′ > d & P[d′] exists do
21     yield d′ ;
22  end

Generator ActionSuccessors(a : ExtensionAction or None):
23     // Go through each action a′ > a, or all possible actions if
24     // a is None
25  foreach a′ s.t. a is None || a′ > a do
26      yield a′ ;
27  end

Generator ExtensionCoupletSuccessors((d, a) : ExtensionCouplet, P : CanonForm):
28  foreach d′ in DirectiveSuccessors(d, P) do
29      foreach a′ in ActionSuccessors(a) do
30         yield (d′, a′);
31  end
end

Algorithm 1: Successor Enumeration Algorithm
non-pruned candidates. In our testing, this method performed significantly better than a simple DFS due to its ability to quickly find more complex patterns, albeit at a cost to thoroughness in limited time periods. Given an infinite amount of time however, both algorithms would produce the same results.

Another key difference between the cited algorithms and our own is the criteria we evaluate resulting patterns by. gSpan and other algorithms optimize for how many nodes a pattern $P$ supports, whereas we are interested in how many it captures. Luckily, as established in theorem 4.2.1, $P \models E$ is a prerequisite for $P \models E$, so support is still meaningful for pruning any candidates that couldn’t possibly be worth further examination. The exact nature of the heuristic by which we score $P$ is defined later in section 4.4.1, and will be ignored for the time being except to say that we are looking for more complex patterns where possible.

**Remark 5.** A key optimization is how we handle the descendants of each canonical form candidate. They are enumerated as described in algorithm 1. Suppose that when enumerating direct descendants of a candidate $curr$ we obtain two canonical forms $L_1 = curr \oplus C_1$ and $L_2 = curr \oplus C_2$, where $C_1 < C_2$ and thus naturally $curr < L_1 < L_2$. Suppose that all three of $curr$, $L_1$, $L_2$ support the same number of nodes in our input set. We claim that there is no reason to consider evaluating $L_2$ or its descendants. Our argument for this is because the couplet $C_1$ has no negative impact on our support, it should be included (since we are looking for larger patterns where possible). Since $L_2$ will never have the option of adding $C_1$ due to the order of enumeration, but $L_1$ will have the option of adding $C_2$, the choice to ignore $L_2$ in favor of $L_1$ is clear.

Our enumeration algorithm operates as described below, in Algorithm 2. It utilizes Algorithm 1 to enumerate candidates, and incorporates in the aforementioned pruning techniques.
Note that the redundant call to ComputeSupport is cached in our actual implementation of this algorithm, and is only included here for clarity. Also note that if we replace the pop_random on line 3 with a simple pop, this would be a normal Depth-First-Search search. In our experiments the random popping generally found desirable patterns much more quickly than DFS. We believe that this could potentially be improved even further by weighting the random selection by compression score (described in Section 4.4.3). However, we did not have sufficient time to thoroughly examine this possible improvement.

4.4 Folding Pattern Instances

Finally, we arrive at the meat of this report. This section outlines what we do with the patterns we discover, specifically how they are used to identify repeating behavior indicative of loops.

As a simple analogy, suppose someone was developing a piece of software, and found that a certain snippet of code was repeated many times in sequence. It would be sensible then to replace that section of repeating code with a loop with that snippet as the loop body. Any differences in the arguments handling any circumstantial differences.

Our pattern mining system is essentially an automated version of the above process. Instead of operating on the source code directly, it analyzes the constraints generated by symbolically running the compiled assembly. Repeating occurrences of certain types of a pattern candidate $P$ can be used to identify what are more than likely loops in our constraints. To accomplish, we first need a way to represent occurrences of $P$ in our expressions. We detail the definitions for how we do this “Folding” in section 4.4.1.
We would also like to deal with concrete $\star$ values in $P$ which could possibly be replaced with more specific values. A specialized procedure to accomplish this is detailed below in section 4.4.2.

Finally, how we actually identify loops once all of this is done is formally explained in 4.4.4.

4.4.1 Representing Pattern Occurrences

Taking the above analogy to its logical conclusion, we attempted to treat occurrences of patterns as calls to a function, with data corresponding to wildcards forming the arguments. We refer to the process of replacing these occurrences with function calls as **Folding** and use it in a very similar to the identically named process in the report [2]. **Unfolding** then is simply the process of undoing our function substitutions, replacing every instance of the function with the full pattern including proper substitution of arguments for wildcards. Though we do not ever unfold anything in this report, it is important to establish that Folding is an entirely reversible process, which preserves validity of the expression.

The main difference between the prior Fold/Unfold algorithm and our own is that Fold/Unfold focused more on deliberately unfolding known function definitions to find potential optimizations, then folding them back up. Our process instead focuses on an already unfolded tree, as formed by symbolic execution, and attempting to fold back up to something close to the original function without knowing anything about it.

As an example, suppose we have the expression tree $e = 3 > ((x * 5) - (y * 5))$. Suppose that by some pattern recovery technique, we find the pattern $p := (\star \times 5)$. If as described earlier, we finalize the arguments of $P$ to be finite and with well-defined positions, we would obtain a pattern interface $P = f(a_0) \equiv (a_0 \times 5)$. Then,
collapsing $E$ with $P$, we get $\text{Fold}(E, P) = 3 > (f(x)) - (f(y))$.

Quite simple! This example is somewhat trivial, and the resulting constraint is about the same size as the original, and doesn’t really exhibit any loop-like behavior. We present more advanced examples later.

In order to properly represent these “calls to functions” and fold them, we need a way to represent a call to a function.

**Definition 23 (Pattern Instances).** An **Pattern Instance** is a special trait type that represents an occurrence somewhere in an expression $E$ of a pattern candidate $P$. There is a unique Pattern Instance $\text{Instance}<P>$ for every pattern $P$.

These trait type are somewhat novel for the fact that they are generated on-the-fly to represent mined patterns. However, other than that they follow all of the same rules as normal trait types, as described in definition 8. Pattern instance traits are represented as follows:

Suppose we have a pattern $P$. Let $W$ refer to the set of directives pointing to every $\star$ in $P$. Let the independent sets $W_p, W_c \mid W_p \cup W_c = W$ refer to the set of all core and all wildcard directives (respectively) in $W$. Let $L_P = (D_{p.1}, D_{p.2}, \ldots, D_{p.|W_p|})$, where refer to the ordered $(D_{p.1} < D_{p.2} < \cdots < D_{p.|W_p|})$ list of progeny directives in $W_p$. Let $L_C = (D_{c.1}, D_{c.2}, \ldots, D_{c.|W_c|})$, where refer to the ordered $(D_{c.1} < D_{c.2} < \cdots < D_{c.|W_c|})$ list of core directives in $W_c$.

Then we define the Pattern Instance Trait type for $P$ as $T_P = [N_P, |W_c|, |W_p|]$, where $N_P$ is some automatically generated unique identifier for this trait.

If $t$ is an instance of $T_P$ representing the expression node $E$ as matched by $P$, then $t$ will have the following properties:

- $t[i] = E[D_{p.i}] \forall i \in 1..|W_p|$
• $t[i + |W_p|] = E[D_{c,i}] \forall i \in 1..|W_c|$

Since the capture match function $M_{\text{Capt.}}\{P, S\} : W \rightarrow V$ fully describes this mapping of wildcards to values, we denote an instance of $E$ folded by $P$ as $\text{Instance}^{P>M_{\text{Capt.}}\{P, E\}}$.

In simpler terms, each item established by $T_P$ corresponds to a specific directive pointing to a wildcard in $P$. There are exactly as many progeny items in $T_P$ as there are wildcards in progeny item slots of $P$, and exactly as many core items in $T_P$ as there are wildcards in core item slots of $P$.

Within the categories of core and progeny, the items are ordered based on the natural ordering of their corresponding directives.

**Definition 24** (Folding). We write $E$ folded by $P$, written $\text{Fold}(E, P)$, to describe the result of recursively each subterm $S$ of $E$ and, if $M = M_{\text{Capt.}}\{P, S\} : W \rightarrow V$ exists, replacing $S$ with a new expression $S'$ formed of a single trait of type $\text{Instance}^{P>M}$.

Some examples of constraints pre- and post-folding can be found in appendix B.

### 4.4.2 Core Item Resolution

The candidates generated by our pattern mining algorithm (see Algorithm 2) do not have any non-⋆ core items. In fact, core items play virtually no role in mining common patterns as long as they are left ⋆. However, in order to maximize our compression factor (described below in 4.4.3) as well as to improve legibility of our graphs, we would like to replace core wildcards with specifics items where possible.

More formally, given a pattern candidate $P$ and a set of expressions $S_E = \{E_1, E_2, \ldots, E_n\}$ where $P \models E_i \forall i$, we wish to find a better pattern $P \ast \succ P$, such that $P\ast$ differs from $P$ only be replacing one or more core $\ast$ items with specific
values drawn from $S_E$. Our hope is that $P*$ still captures a significant subset of $S_E$. The algorithm we designed typically will, but also allows for producing a $P*$ that only supports a subset of $S_E$ if it means a significant decrease in the number of wildcards.

The algorithm is as follows:

Similar to our pattern mining algorithm, we select what extensions to make and which items to use stochastically. Though this slightly limits reliability, it avoids the issue of having to deal with the exponential number of possible variations on which core items to put where. In our tests, as long as one did this at least 3 or more times it would generally produce a desirable result, though we did 10 just for safety’s sake.

### 4.4.3 Compression Factor

The candidates generated by our pattern mining algorithm (see Algorithm 2) are selected to have high support/capturing. Unfortunately, the pattern does not discriminate between patterns that are good or bad for collapsing. But what defines a “good” a pattern for collapsing, specifically with regards to finding loops? An obvious criteria is whether or not it captures a sufficient number of nodes. Since the mining algorithm only checks support, we will have to evaluate this as an additional pruning step.

Another key factor is how complex the pattern is. If we allow patterns that are too simple, we could easily end up doing trivial folds that do nothing except rename existing operations (e.g. Folding on $f(x, y) = x + y$).

Eventually we arrive upon the following heuristic:
Definition 25. Given \( E \), an expression node that has been folded some number of times. Given \( (P_1, P_2, \ldots, P_n) \), the set of patterns which have folded \( E \) and for which trait instances exist within \( E \) or \( E \)’s extended progeny.

Let \( N_E \) be the set of expression nodes throughout \( E \) (essentially the total number of progeny items). Let \( N_P \) be the set of pattern nodes throughout all \( P_i \). Let \( T_E \) be the set of trait instances throughout \( N_E \). Let \( T_P \) be the set of trait instances throughout \( N_P \). Let \( W \) be the number of wildcards summed across all \( P_i \).

We say that \( E \) has an **Expression Complexity** of

\[
\text{ExpressionComplexity}(E) = |N_E| + |N_P| + |T_E| + |T_P| + W.
\]

Here we see the results of Algorithm 3 coming into play. Fewer wildcard core items results in a lower (and thus more desirable) complexity.

We say that given an expression \( E \) and a new pattern \( P \), that the **Compression Score** of \( P \) on \( E \) is equal to

\[
\frac{\text{ExpressionComplexity}(\text{Fold}(E, P))}{\text{ExpressionComplexity}(E)}.
\]

Lower compression scores indicate a greater reduction in complexity. A complexity score that is greater or equal to 1.0 is an increase in expression complexity, and generally undesirable.

**Remark 6.** This heuristic was decided upon without any formal reasoning to back it up. Many aspects of it, from the weighting of different elements, to whether the values should relate linearly or in a more complex fashion, were a subject of interest that we unfortunately did not have time to thoroughly explore.

However, as it stands it performed well enough in preliminary tests that we decided it was acceptable to use for our results. One major downside of the current formula is that it places excessive weight on the total size of the graph. This unfor-
fortunately means that sometimes multiple iterations of a loop will be recognized as
one or two exceedingly complex iterations, which unfortunately breaks our ability
to detect them. We think that further research into a heuristic that promotes small
yet still meaningful patterns is worthwhile.

4.4.4 Identifying Loops

Our loop identification technique is very simple. We write that an expression $E$
containing an instance of the Pattern Instance trait $t$ has a Recurrence if for exactly
one progeny item $p$ of $t$, $p$ also contains a Pattern Instance trait $t'$ s.t. $\text{Ident}(t') \equiv$ \text{Ident}(t').

We claim that a loop has probably occurred if there are $l$ successive recurrences,
where $l$ is an integer can be varied to increase or decrease sensitivity towards loops.
If a loop is detected, we fold all recurrences of the loop into a single, specialized node
representing the unique wildcard values at each “iteration” of the recurrence. This
improves our heuristic values on detected loops thus encouraging the mining algo-

This claim is not necessarily always true - loops of fewer than $l$ iterations will
be missed, but setting $l$ too low is likely to result in false positives. We lacked a
large enough corpus of test data to really warrant a more advanced identification
algorithm, and would advise that future research not cling unnecessarily to this
criteria for loops.

4.5 Algorithm Overview

The overall processing pipeline is outlined below.
1. Using ANGR, generate symbolic constraints for some set of input states.

2. Convert the generated constraints to our special tree representation, and denote the set of all root expression nodes as $E$.

3. For some time budget (in our tests 60 seconds), run our mining algorithm described in 4.3 on $E$. For each $P$ yielded from the algorithm, generate 8 core-tightened patterns $P_1, P_2, \ldots, P_8$ as described in 4.4.2. Select the resulting pattern $P^*$ with the best compression factor (see 4.4.3) across all $P_i$ yielded from the algorithm.

4. If $P^*$’s compression factor $F$ is better than a given threshold (in our tests, we used $F < 0.9$), then fold all $E$ with $P^*$, and try to mine another by returning to step 3.

5. Count the number of loops, via the criteria described in 4.4.4.

6. Return the set of folded nodes $E$, and a list of identified loops.
Data: $S_E = \{E_1, E_2, \ldots, E_n\}$ such that each $E_i$ is an expression.

Data: $l$, a minimum number of nodes that a candidate must support.

Result: A sequence $C = \{P_1, P_2, \ldots\}$ such that each $P_i$ supports at least $l$ nodes of $S_E$

/* We begin with the root canonical form, $()$, and continue until all possibilities are exhausted. */

1 candidates ← [()];
2 while candidates ≠ [] do
3     // Randomly select next candidate
4     curr ← candidates.pop_random();
5     // Compute number of nodes it supports from $S_E$
6     curr_sup ← ComputeSupport($S_E, curr$);
7     // If support is insufficient, move on to the next candidate
8     if curr_sup < l then
9         continue;
10    end
11    // Yield current candidate for further evaluation.
12    yield curr;
13    /* Next we handle the successors of the current candidate.
       They are greedily pruned as described above. */
14    successors ← EnumerateChildren(curr);
15    foreach $s \in$ successors do
16        candidates.append($s$);
17        // If $s$ is a "$\text{``perfect''}\$" successor, as described above 5,
18        // don't go further
19        if ComputeSupport($s$) = curr_sup then
20            break;
21        end
22    end
23 end

Algorithm 2: Pattern Mining Algorithm
Data: \( P \), an initial pattern candidate that we wish to constrain.

Data: \( S_E = \{E_1, E_2, \ldots, E_n\} \) such that each \( E_i \) is an expression captured by.

Result: \( P^* \) as described above.

1. Let \( W \) refer to the set of directives to core \(*\) values in \( P \);

2. foreach \( w \in W \) do

   // Select a captured expression at random to sample a core item from
   \[ e \leftarrow \text{random.sample}(S_E) \]

   // Get the core value from \( e \) corresponding to directive \( w \)
   \[ v \leftarrow e[w] \]

   // Copy \( P \) as \( P' \), and add the core item from \( e \)
   \[ P' \leftarrow P \]

   \[ P'[w] \leftarrow v \]

   // Find the new set of items captured by \( P' \)
   \[ S'_E \leftarrow \{E \mid E \in S_E \& P' \models E\} \]

   /* Decide whether to keep \( P' \) by checking a random number against the ratio kept. A higher ratio gives a higher probability, with a 1:1 ratio being guaranteed to keep the change to \( P \) */

   \[ \text{ratio}_\text{kept} \leftarrow |S'_E|/|S_E| \]

   if \( \text{random.random()} < \text{ratio}_\text{kept} \) then

   \[ P \leftarrow P' \]

   \[ S_E \leftarrow S'_E \]

   end

end

12. return \( P \);

**Algorithm 3:** Core Item Constraining Algorithm
Chapter 5

Results

To test our algorithm, we used a small suite of four simple programs in the C programming language, designed to showcase the specific cases where it should excel, or fail. The source code to these programs can be found in appendix A.

The algorithm was scored based on its ability to accurately detect the presence loops in the original program source code by analyzing the constraints produced by running that code symbolically. This was tested across different compilation settings and compilers, to highlight the techniques potential resilience against obfuscation. More specifically, we compiled the programs listed in Appendix A the settings listed below.

1. The GNU C Compiler (gcc) version 7.3.0, with no additional flags. This was to test the algorithm’s performance on minimally-optimized assembly.

2. The GNU C Compiler (gcc) version 7.3.0, with optimization level 3 (using flag -O3). This was to test performance of the algorithm against highly optimized assembly.

3. The clang Compiler version 3.6, with no additional flags. This was to test the
algorithm’s performance on minimally-optimized assembly (and to serve as a control for item 4.

4. The clang Compiler version 3.6, using the llvm-obfuscation tool described in the paper [8]. This was to test the algorithms performance on heavily obfuscated code, including arithmetic obfuscation, control flow obfuscation, and control flow flattening. The specific options used for this were

\[
\text{clang src.txt -o bin_main -mllvm sub -mllvm bcf -mllvm -fla.}
\]

The number of loops detected versus the number that actually exist in the source is presented in the table below. A selection of the constraints generated by analyzing each binary, before and after compression, can be found in Appendix B. Note that the number of loops is not simply the number of recurrences as defined in section 4.4.4, but rather the number of unique loop bodies identified. That is, loops on each pattern \( p \) are only counted once even if we discover many recurrences of sufficiently large size on \( p \). This decision was made because, due to the branching nature of symbolic execution, in many cases we ended up with hundreds of states (and thus sets of constraints) that each contained a loop, but on the same pattern.

### 5.1 Scores

<table>
<thead>
<tr>
<th>Program</th>
<th>gcc</th>
<th>optimized gcc</th>
<th>clang</th>
<th>obfuscated clang</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum Fixed-Size Array</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sum Integer Range</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sum Fixed-Size Nested Array</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Iterate Conditional</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Detected Number of Loops for each Compilation Target.
5.2 Analysis

5.2.1 Sum Fixed-Size Array

*Source code found in appendix A.1.*

This program’s simple repeating arithmetic is the best case situation for our algorithm to find loops. The constraints has a very clear repeating pattern that is small enough to be detected extremely quickly, as can be seen in the Appendix B. The loop performs a fixed number of iterations, so there was only one constraint produced per binary.

Furthermore, due to the simplicity of the function, no notable optimizations (which can hinder automated analysis) were performed by any of the compilers. Even the obfuscation techniques used in compiler 4 did not have any effects on the generated constraints. As a result of this, the constraints generated across all 4 compiler configurations were in fact identical, lending credence to earlier claims of symbolic executions resilience against optimization.

5.2.2 Sum Integer Range

*Source code found in appendix A.2.*

This program is essentially slightly more complicated case than the one above. At its core, this program is still just repeated addition in a for loop. However, there are two notable distinctions between this and the previous test.

One, the number of iterations the loop might perform is variable (depending on our passed argument). This means that we actually ended up our execution branching into a large number of unique states, each representing a different number of iterations of the program. We limited the total number of states generated to 30, under the assumption that any pattern of note would manifest by then.
Two, unlike the previous example this program demonstrated different behaviour (in terms of generated constraints) for each compilation configuration. The optimized gcc version is especially noteworthy here since it made extensive use of bitwise operations that one wouldn’t normally expect from such a simple sum function.

Nevertheless, our program was able to recognize loops in all cases except the optimized gcc version. This is disappointing, as in the provided constraint examples from the appendix, it is clear that a repeating pattern does exist, just that our algorithm was unable to find it.

5.2.3 Sum Fixed-Size Nested Array

Source code found in appendix A.3.

This program served as a more complicated version of “Sum Fixed-Size Array,” described above. The loop structure and arithmetic are still very simple, and in theory should pose no special issue to our algorithm.

However, as can be seen in the score table, our algorithm only recognizes the presence of one for loop, despite the presence of two. Looking at the constraints in the appendix, it becomes clear as to why. In spite of the flow of execution going through two loops, the only arithmetic operations performed are repeated additions on elements of an array. Each addition is structurally identical in the constraint, and so our algorithm was unable to detect a clear indication as to when each sub-loop ended.

The biggest indication in the constraints that a second loop exists is the fact that the loop has 6 separate segments: one using ROW_1, the next using ROW_2, then ROW_3, etc. Unfortunately, the heuristic of our algorithm prioritized making a general collapsing function for all 6 subsections, rather than a unique one for each (since more patterns count negatively against the heuristic, to prevent frivolous
collapsing). With a better heuristic function, we believe that our algorithm would be able to identify the second loop (perhaps one that more heavily prioritized having fewer wildcards). However, this does indicate a broader weakness in our algorithm in that it is unable to distinguish sufficiently similar loops.

5.2.4 Iterate Conditional

*Source code found in appendix A.4.*

This program was designed to test the capabilities of our algorithm in the face of one of the worst-case situations for symbolic executors - that of an exponentially branching loop. Each time the if statement inside of the loop was checked, ANGR split the execution state into two. This means that even with only 10 iterations, we ended up with a total of $2^{10} = 1024$ states/constraints to analyze. Unfortunately, this proved almost impossible for our algorithm to analyze in a meaningful amount of time, with approximately 500 times more expressions that we needed to check for support slowing the algorithm to a crawl.

It is disappointing that our algorithm faltered here, as it is perhaps the biggest issue limiting usage of symbolic execution. However, with improved caching techniques, and a more intelligent enumeration process, it is possible that the performance of our algorithm could be vastly improved on highly-similar constraints such as the ones generated here.

**Remark 7.** Due to the differing nature of constraints and the code itself, we cannot formally claim that the loops detected are in fact the loops present in the code. The reader will have to make that call for themselves based on what they see in the appendix. We do not claim that this algorithm will work in a general case – in our testing we found many programs that ANGR could not even analyze, and were thus unable to test our algorithm against.
Chapter 6

Conclusions

6.1 Accomplishments

We were able to identify the presence and number of loops present in our sample programs, although with a low level of accuracy. Though identifying the presence of loops using constraints isn’t enough to actually understand what a function is doing, it serves as a starting point towards future structural identification techniques. Furthermore, it acts as a proof of concept of the constraint-pattern mining algorithm put forth in this report, even if the heuristic we came up with wasn’t really optimal. Though there are plenty of existing pattern mining algorithms, very few include the property that instances of the pattern can be compressed without loss of semantics. We believe that this algorithm could easily be modified to perform more advanced analysis, such as recognizing similar functions between binaries, without any loss in soundness. Unfortunately, we were unable to test these ideas due to time constraints.

As a side benefit, the folded constraints are typically much more easily visually understood. The function instances highlight similarities between different expressions, and dramatically reduce the number of nodes to visually parse and compare.
We cannot claim this as an objective improvement without doing user studies, but if nothing else its a nice bonus to our work.

6.2 Current Limitations / Future Work

There are several main issues with the project as it stands.

First of all, the resulting constraints are incompatible with ANGR, since ANGR provides no method of declaring custom operations at run-time. Though Unfolding instances is fairly simple, it might be useful to experimentally support pattern instances in ANGR.

Second, the algorithm is unfortunately fairly slow. Beyond the inherent slowness of a single threaded python implementation, there are undoubtedly a multitude of potential optimizations that could be performed. We already cache the results of pattern matching via hashing in a similar fashion to [9], but there is undoubtedly a lot more that we can do.

Third, in many cases over-eager simplification by ANGR (e.g. replacing (((((1 + 2) + 3) + 4) + 5) with 15) destroys structure/loops in the constraints. This kind of on-line simplification is default behavior of ANGR and most other symbolic executors, as without it RAM usage quickly grows to untenable levels.

Finally, though there are undoubtedly countless possible improvements, one of the most obvious is to improve the heuristics for identifying which common substructures are “the best”. This would probably require amassing a larger body of test programs, but is very simple to prototype around as it is essentially just a single line of arithmetic. Though our measure of ExpressionComplexity does a serviceable job, it frequently produces patterns that poorly match what a human being would identify as the most obvious repeating element of the tree.
Appendix A

Test Program Source Code

A.1 Sum Fixed-Size Array

#define SIZE 5

int sum_byte_array(int arr[SIZE]) {
    int x = 0;
    int i;
    for(i=0; i < SIZE; i++) {
        x += arr[i];
    }
    return x;
}

A.2 Sum Integer Range

int sum_natural_numbers(int zero, int max) {

int i;
int x = zero;
for(i=1; i<=max; i++)
{
    x = x + i;
}
return x;

A.3 Sum Fixed-Size Nested Array

// We performed our tests with a fixed 6x6 array.
// Due to angr's limitations with nested loops,
// (or perhaps just our unfamiliarity),
// we were unable to produce a satisfying test case
// using a symbolic width/height
int sum_2d_byte_array(int** arr, int width, int height) {
    int x = 0;
    int r;
    int c;
    for(r=0; r < height; r++) {
        for(c=0; c < width; c++) {
            x += arr[r][c];
        }
    }
    return x;
A.4 Iterate Conditional

void looped_conditional(int interval) {
    // This function prints a 'Y' every
    // interval characters, and otherwise prints 'N'
    // At most 10 characters will be printed
    int i;
    int f;
    for(i=0; i<10; i++) {
        if(f == interval) {
            printf("Y");
            f = 0;
        } else {
            printf("N");
            f += 1;
        }
    }
    printf("\n");
}
Appendix B

Example Constraint Results

This appendix contains a selection of example constraints before and after our constraint compression/loop detection algorithm is run on them. Note that if a constraint $C$ is stated as being representative of the return value, there is an implicit $return = C$ wrapping it (this was simply done to be consistent with how ANGR stores its constraints, internally).

B.1 Sum Fixed-Size Array

The constraints on this program are actually the same between different compilation targets, perhaps because of the simplicity of the function.
Figure B.1: Constraint on return value, pre-compression
Figure B.2: Constraints on return value, post-compression. Each row of the “Recurrence” trait represents one instance of the function f, chained onto the next row. Since there are at least 4 rows, this was classified as a proper loop by our algorithm.

B.2 Sum Integer Range

For this program, only the non-optimized gcc and non-obfuscated clang compilation targets had identical constraints. The other 2 targets generated unique constraints. Since each generate 30 unique states (and thus 30 constraints of interest), we have limited our inclusions to the first 3 and the final constraint of each set.
Figure B.3: Constraints on one-iteration return value, pre-compression. However, this was small enough that no compression was even possible.
Figure B.4: Constraints on two-iteration return value, pre-compression. However, this was small enough that no compression was even possible. Note how the previous constraint can be found as a sub-tree of this one.
Figure B.5: Constraints on two-iteration return value, post-compression. Here we see exactly one instance of the function $f$.

Figure B.6: Constraints on twenty-iteration return value, pre-compression. Though it is likely too small to read, the repeating structure is visible.
Figure B.7: Constraints on twenty-iteration return value, post-compression.
B.2.2 Optimized GCC

As mentioned in results, our program was unable to find any loops in these constraints. As such we only show the uncompressed versions.

Figure B.8: Constraints on one-iteration return value, without-compression. However, the constraint is small enough that no compression was possible.
Figure B.9: Constraints on two-iteration return value, without-compression. However, the constraint is small enough that no compression was possible. Note how the previous constraint can be found as a sub-tree of this one.
Figure B.10: Constraints on twenty-iteration return value, without-compression. A repeating pattern is visible, but unfortunately our algorithm was unable to detect it.

B.2.3 Obfuscated Clang

In the obfuscated clang, we were unable to actually recover the constraint on the return value. Though we were unable to find it, the algorithm found a loop in the path constraints that seems to represent the value as well.
Figure B.11: Constraints on one-iteration return value, pre-compression. However, this was small enough that no compression was even possible.
Figure B.12: Constraints on two-iteration return value, pre-compression However, this was small enough that no compression was even possible. Note how the previous constraint can be found as a sub-tree of this one.
Figure B.13: Constraints on two-iteration return value, post-compression. Here we see exactly one instance of the function $f$. 
Figure B.14: Constraints on twenty-iteration return value, pre-compression. Though it is likely too small to read, the repeating structure is visible. It is not as “clean” as in previous examples, but is still there.
Figure B.15: Constraints on twenty-iteration return value, post-compression. Unfortunately the entire loop wasn’t able to be captured due to differences in the iterations. However, they were similar enough to meet our threshold.

B.3 Sum Fixed-Size Nested Array

The constraints for this program ended up being almost identical between all compilation targets except for the obfuscated clang compiler. Since there was a fixed number of loop iterations for each target, we have only a single constraint per program to show.

B.3.1 GCC, Clang and Optimized GCC

Similar to our first program, these simplified very cleanly.
Figure B.16: Constraints on return value, pre-compression. Repeating structure is clearly visible, even if the constraint is large enough as to be rendered partially illegible.
Figure B.17: Constraints on return value, post-compression. Each row of the loop here represents the addition of an element of the two-dimensional arrays. The arguments to the pattern function encode both which row we are operating on (here represented as 6 separate symbolic variables), and which bits from the row are being added (representing the index in the row).
B.3.2 Obfuscated Clang

The compression algorithm actually found two patterns it decided were worth compressing. However, only $f(x)$ one actually had a recurrence. The other simply was used to make the constraint smaller and more quickly analyzed, as less nodes means less time to check support.

Figure B.18: Constraints on return value, pre-compression. The individual repeating structures are each larger than in the non-obfuscated compiler, however the actual overall chain-like structure is more or less the same.
Figure B.19: Constraints on return value, post-compression. Parameters serve identical function to those in non-obfuscated constraints.
B.4 Iterate Conditional

As mentioned in results, this program produced 1024 constraints in 10 iterations. No compression was found, though the constraints are similar enough that perhaps one could have been. It would be unfeasible to show them all, and we have instead selected 4 gcc state constraints at random to show the structural similarity between them. The other compilation targets are omitted because they are more or less the same, structurally, and because none of them compressed successfully anyways.

![Figure B.20: Constraints of state #53 path.](image-url)
Figure B.21: Constraints of state #168 path.

Figure B.22: Constraints of state #173 path.
Figure B.23: Constraints of state #383 path.
Bibliography


