FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Nested loops and path explosion in symbolic execution
Student:	Bc. Vojtěch Rozhoň
Supervisor:	Pierre Donat-Bouillud, Ph.D.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Symbolic execution is a program analysis technique that explores all the paths of a program and solve the various conditions along the paths using a SMT solver. It can create concrete examples of what leads to a particular path, what leads to some bug.

Symbolic execution must explore all the paths of the program to give sound results but the number of paths can increase exponentially, especially in the presence of loops, leading to "path explosion".
Some techniques to reduce the number of paths exist, such as loop summarization, path subsumption, state merging and leveraging static analysis techniques (see Baldoni, Roberto, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A survey of symbolic execution techniques." ACM Computing Surveys (CSUR) 51, no. 3 (2018): 1-39). Nested loops are particularly challenging.

The goal is to implement existing techniques that tackle path explosion due to loops on micro-c programs (https://courses.fit.cvut.cz/NI-APR/microc.html), adapt or even create new approaches for nested loops, and evaluate them. micro-c is an educational language with just a few programs so typical real world programs and nested loops heavy program will have to be written.

1) Research the principles of symbolic execution
2) Research the approaches to solve path explosion
3) Implement path explosion techniques for (nested) loops in Scala for micro-c programs

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 23 January 2024 in Prague.

4) Create representative micro-c programs

5) Compare and evaluate the approaches

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 23 January 2024 in Prague.

Master's thesis

NESTED LOOPS AND PATH EXPLOSION IN SYMBOLIC EXECUTION

Vojtěch Rozhoň

Faculty of Information Technology
Department of computer science
Supervisor: Pierre Donat-Bouillud, Ph.D.
May 9, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Rozhoň Vojtěch. *Nested loops and path explosion in symbolic execution.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my sincere gratitude to my supervisor Pierre Donat-Bouillud, Ph.D. for his valuable help, guidance and patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2024

Abstract

The thesis evaluates techniques for tackling path explosion in symbolic execution. There is a particular focus on the interaction of the techniques with nested loops. Path pruning, path subsumption, state merging, and loop summarization techniques are discussed using examples. Symbolic execution and these techniques are implemented to analyze programs written in an educational language called microc, a language inspired by C. The thesis describes experiments performed to compare the techniques, especially to find out how well the techniques deal with many loops or deep nested loops. A random microc program generator is developed to experiment with the symbolic executor.

Keywords symbolic execution, path explosion, nested loops

Abstrakt

Práce vyhodnocuje techniky pro řešení problému exploze cest v symbolické exekuci. Obzvláštní důraz je kladen na interakce jednotlivých technik se vnořenými cykly. Metody ořezávání cest, subsumpce cest, slučování stavů a shrnování cyklů jsou diskutovány na příkladech. Symbolická exekuce je spolu s těmito metodami implementována pro analýzu programů napsaných v edukačním programovacím jazyce microc, který je inspirován jazykem C. Práce představuje experimenty, které porovnávaly jednotlivé metody, obzvláště aby zjistili jak dobře se tyto metody vypořádávají s mnoha cykly a se vnořenými cykly. Generátor náhodných programů v jazyce microc byl implementován pro experimentování s implementovanou symbolickou exekucí.

Klíčová slova symbolická exekuce, exploze cest, vnořené cykly

Acronyms

AST	Abstract Syntax Tree
BFS	Breath First Search
CFG	Control Flow Graph
DFS	Depth First Search
FIT CTU	Faculty of Information Technology at Czech Technical University in Prague
IV	Induction Variable
NIV	Non-Induction Variable
PDA	Path Dependency Automaton

Chapter 1
Introduction

Testing program correctness is a crucial part of the software development cycle. Developers often manually develop test suites to check that a program behaves as expected for various kinds of inputs.

Automatic program testing seeks to find errors in the program without the developer having to write manual tests. Techniques based on random fuzzing do this by automatically generating numerous input values for the program.

Symbolic execution uses the source code information to enhance the analyses. Instead of generating concrete inputs and running the program with them, it reasons about the program abstractly, evaluating all the execution paths in the code symbolically.

The problem with this approach is that the number of execution paths in the program is typically infinite. Most programming languages have constructs, such as loops or recursive functions, that can exponentially increase the number of paths within the program. This problem is called path explosion.

There are many possible approaches to reduce the impact of path explosion and to allow the analyses to find the errors in the code faster.

This thesis examines different strategies for dealing with path explosion. The techniques include path pruning, path subsumption, state merging, and loop summarization. It examines loops, especially those that provide the biggest challenge for the symbolic executor. These loops contain another loop within their body, and they are called nested loops.

For this thesis, I developed a symbolic executor to analyze programs written in a programming language called microc from the NI-APR course at FIT CTU. Chapter 2 describes this language. Chapter 3 explains how symbolic execution works. Chapter 4 looks at different strategies to reduce the impact of path explosion and also discusses their effect on nested loops.

Chapter 5 examines the implementation of the symbolic executor and the path explosion techniques developed for this thesis. Chapter 6 discusses the experiment results after using the developed symbolic executor to analyze microc programs. The programs are randomly generated because there are not many programs written in microc since it is an educational language. The last chapter, chapter 7, summarizes the whole thesis.

1.1 Goals of the thesis

The thesis aims at analyzing different techniques for tackling path explosion. It discusses their effectiveness on loop-heavy programs.

A symbolic execution engine should be developed, and path explosion reduction techniques should be implemented.

An experiment that analyses the effectiveness of different techniques on loop-heavy programs should be performed, and its results should be analyzed.

Chapter 2
The microc programming language

The thesis uses the microc language that is used in the NI-APR course (Selected Methods for Program Analysis) at FIT CTU. The language is based on a TIP language from the SPA book [1], inspired by the language C [2]. This chapter provides an overview of the language. Section 2.1 defines the basic structure of microc programs. Sections 2.2 and 2.3 define all microc statements and expressions. Section 2.4 talks about errors that we can encounter in microc.

2.1 Overview of microc programs

The microc language is extended to support a few more operators for the purposes of this thesis. The grammar of microc is shown in the Extended Backus–Naur Form (EBNF) [3] format in 2.1.

The language supports numbers, arrays, records, pointers, and functions. The number type can hold only integers. In the original microc, variables can be of a function type, and thus functions can be stored as values of program variables. This feature was disabled for the purposes of this thesis to make code analyses easier. Since the language does not support the boolean type, any condition is expected to be evaluated to a number. If the number is zero, it is evaluated as `false`. Otherwise, it is `true`.

A microc program consists of a list of named functions. Each function takes a fixed amount of named parameters and returns a value. The function also contains two blocks of statements and a final `return` statement. A microc program has to contain the `main` function. The execution of the program starts with this function.

The first block of statements is the variables declaration block. This block contains variable declaration statements that start with a keyword `var`. A variable declaration statement can define multiple variables. All variables that are not function parameters must be declared here before they can be used in the function body. The second block of statements is the function body block. It contains the logic of the function. No variable declaration statement or `return` statements can be placed in this block. Thus, every function has one `return` statement that is always placed at the end of the function.

```
⟨Program⟩ ::=  { ⟨FunDecl⟩ }

⟨FunDecl⟩ ::=  Identifier ( [ ⟨IdentifierDecl⟩ { , ⟨IdentifierDecl⟩ } ] )

⟨IdentifierDecl⟩ ::=  Identifier

⟨FunBlockStmt⟩ ::=  { { ⟨VarStmt⟩ } { ⟨Stmt⟩ } ⟨ReturnStmt⟩ }

⟨VarStmt⟩ ::=  var ⟨IdentifierDecl⟩ { , ⟨IdentifierDecl⟩ } ;

⟨ReturnStmt⟩ ::=  return ⟨Expr⟩ ;

⟨Stmt⟩ ::=  ⟨OutputStmt⟩ | ⟨WhileStmt⟩ | ⟨IfStmt⟩ | { { ⟨Stmt⟩ } } | ⟨AssignmentStmt⟩

⟨IfStmt⟩ ::=  if ( ⟨Expr⟩ ) ⟨Stmt⟩ [ else ⟨Stmt⟩ ]

⟨WhileStmt ⟩ ::=  while ( ⟨Expr⟩ ) ⟨Stmt⟩

⟨OutputStmt⟩ ::=  output ⟨Expr⟩ ;

⟨AssignmentStmt⟩ ::=  ⟨Expr⟩ = ⟨Expr⟩ ;

⟨Expr⟩ ::=  ⟨LogicalExpr⟩

⟨LogicalExpr⟩ ::=  ⟨EqualityExpr⟩ { (&& | ||) ⟨EqualityExpr⟩ }

⟨EqualityExpr⟩ ::=  ⟨RelationalExpr⟩ { (== | !=) ⟨RelationalExpr⟩ }

⟨RelationalExpr⟩ ::=  ⟨AdditiveExpr⟩ { (> | >= | < | <= ) ⟨AdditiveExpr⟩ }

⟨AdditiveExpr⟩ ::=  ⟨MultiplicativeExpr⟩ { (+ | -) ⟨MultiplicativeExpr⟩ }

⟨MultiplicativeExpr⟩ ::=  ⟨UnaryExpr⟩ { (* | /) ⟨UnaryExpr⟩ }

⟨UnaryExpr⟩ ::=  ⟨Deref⟩ | ⟨Ref⟩ | ⟨Input⟩ | ⟨Alloc⟩ | ⟨Null⟩ | ⟨PostfixExpr⟩ | ⟨Not⟩

⟨Deref⟩ ::=  * ⟨UnaryExpr⟩

⟨Ref⟩ ::=  & ⟨Identifier⟩

⟨Input⟩ ::=  input

⟨Alloc⟩ ::=  alloc ⟨Expr⟩

⟨Null⟩ ::=  null

⟨PostfixExpr⟩ ::=  ⟨PrimaryExpr⟩  ⟨FieldAccess⟩ | ⟨ArrayAccess⟩ | ⟨Call⟩

⟨Not⟩ ::=  ! ⟨Expr⟩

⟨FieldAccess⟩ ::=  . ⟨Identifier⟩

⟨ArrayAccess⟩ ::=  [ ⟨Expr⟩ ]

⟨Call⟩ ::=  ( [ ⟨Expr⟩ ’, ⟨Expr⟩  ] )

⟨PrimaryExpr⟩ ::=  ⟨Number⟩ | ⟨Identifier⟩ | ⟨Array⟩ | ⟨Record⟩ | ⟨Paren⟩

⟨Record⟩ ::=  { [ ⟨Field⟩  , ⟨Field⟩  ] }

⟨Array⟩ ::=  [ [ ⟨Expr⟩  , ⟨Expr⟩  ] ]

⟨Paren⟩ ::=  ( ⟨Expr⟩ )

⟨Field⟩ ::=  ⟨Identifier⟩ : ⟨Expr⟩

⟨Number⟩ ::=  [ - ] (0 ... | 9)

⟨Identifier⟩ ::=  (_ | A ... Z | a ... z) { (_ | A ... Z | a ... z | 0 - 9) }
```

Code listing 2.1 The grammar of the microc language in EBNF format

2.2 Statements

This section lists all microc statements.

VarStmt The statement defines a list of variables.

OutputStmt The statement outputs the evaluated expression to the terminal.

AssignmentStmt The statement is used to update the value of a variable. The expression on the right side of the assignment is evaluated, and its value is assigned to a memory location specified by the expression on the left side.

The expression on the left side can be `Identifier`, `Deref`, `ArrayAccess`, or `FieldAccess` expressions.

ReturnStmt The statement is the final statement of a function. The expression in the statement is evaluated, and the value is returned from the function. The main function always returns a number.

IfStmt The statement contains at least one block of statements. If the `else` keyword is present, there is one more block of statements. It also contains a condition. The condition will sometimes be referred to as the guard. The condition is an expression that is expected to be evaluated to a number, If the condition does not evaluate to zero, the first block of statements gets executed. Otherwise, the second block of statements gets executed.

WhileStmt The statement contains a block of statements and an expression. If the expression is not evaluated to zero, the block of statements is executed. When the execution of the statements finishes, we repeat the check of the condition and the execution of the statements until the condition evaluates to zero.

2.3 Expressions

This section lists all microc expressions.

Binary expression The binary expression is used to compute a value from two input expressions. The original microc grammar supports the operators `==`, `>`, `+`, `-`, `*` and `/`. In this thesis the operators `<`, `<=`, `>=`, and `!=` are added too.

Not The expression is a unary operator. It expects a number and returns the number one if the input number is 0. Otherwise, it returns the number zero. It is not part of the original microc but was added to this thesis.

Input The expression is used to load a user input number. It can be any number.

Identifier The expression is used to load a value of a variable from the memory.

Null The expression is used to create a null pointer.

Alloc The expression evaluates a subexpression and adds the created value to the memory. A pointer pointing to this memory location is returned.

Varref The expression returns a pointer to the value of an identifier.

Deref The expression contains an inner expression. This expression is expected to be evaluated to a pointer value. The value at the memory location this pointer points to is returned.

Record The expression is used to create a record value. A record value can contain several fields. Every field has a name, and it can hold a value. Nested records are not allowed.

- **Record Field** A record field is defined by a name followed by an expression. A field has to hold values of the same type at any time of the execution.

FieldAccess The expression is used to access a field of a record. The accessed record is retrieved by evaluating a subexpression.

Array The expression produces an array with a fixed amount of elements of the same data type. Every element of an array is initialized by an expression.

ArrayAccess The expression evaluates a subexpression that is expected to return an array value. The other subexpression is expected to return a number. The element at the index of the number in the array is accessed.

Code listing 2.1 shows a simple program written in microc. The program consists of two functions called `main` and `f`.

Code listing 2.1 An example of a micro program

```
f(n) {
  var r;

  if (n == 0) {
    r = 1;
  } else {
    r = n * f(n - 1);
  }
  return r;
}

 main() {
   return f(5);
 }
```

The function `f` takes a parameter called `n`. The list of variable declaration statements consists of one declaration statement that declares a variable named `r`. The body contains a `if` statement that contains a block with one `assign` statement in both branches. The return statement of the function returns the variable `r`.

The `main` function contains just a `return` statement, meaning that both the list of variable declarations and the list of body statements are empty.

2.4 Possible errors in microc

In the microc language, there are several possible errors related to variable types. A variable needs to have the same type at any time during the execution, and every condition has to be evaluated to a number.

Other invalid programs are produced by using an identifier whose value is not a function for a function call or missing the `main` function in the program.

We leave the detection of these errors for a different sort of analysis, such as a type analysis. We will assume that the programs are correctly typed.

There are also situations when a program is syntactically valid and type correct, yet there is a problem that causes a crash of the program during runtime.

division by zero Division by zero is not a mathematically defined operation.

nullptr dereference A null pointer is a pointer that points to nowhere. Thus, dereferencing it does not make sense.

array access out of bounds If we try to access an element of an array at a negative index or index behind the end of the array, then there is nothing to retrieve, and an error occurs.

use of an uninitialized value It is forbidden to use a variable that has not yet been initialized.

The following code 2.2 snippet shows a microc program with several errors.

Code listing 2.2 An example of a micro program with errors

```
1
2  main() {
3    var arr, res, a, ptr;
4    arr = [1, 0, 2, 2];
5    ptr = null;
6    res = 1 / arr[1];
7    res = a + a;
8    res = arr[4];
9    res = *ptr;
10   return 0;
11 }
```

Line 6 contains a division by zero error, line 7 contains the use of an uninitialized value, line 8 contains an array access out-of-bounds error, and line 9 shows a null pointer dereference error.

Chapter 3

Basics of symbolic execution

This chapter discusses the basic concepts of symbolic execution. Section 3.1 discusses the control flow graphs. Section 3.2 defines symbolic execution. Section 3.3 explains the concept of symbolic states. Section 3.4 discusses the problem of constraint solving. Section 3.5 discusses the problem of unbounded loops. Section 3.6 discusses the problem of path scheduling, and section 3.7 lists other properties of symbolic execution.

An automated program testing technique takes a program as input and tries to find as many errors in the program as possible. The tested program is loaded into memory and analyzed. There are multiple ways of representing a program within a program testing framework, one of which is the control-flow graph (CFG).

3.1 Control flow graphs

The CFG is a graph whose nodes can contain one or more statements. The statements within one node are always executed sequentially. A directed edge is present between two vertices if the execution of the target vertex can immediately follow the execution of the source vertex. The entrypoint statement of the program is the only vertex with only outgoing edges in the graph, while the exitpoint statements only have ingoing edges. The nodes with several successors in the CFG are called *control* nodes. These nodes correspond to conditions in the source program.

Code snippet 3.1 shows an example of a small program written in microc.

Code listing 3.1 An example of symbolic execution analyses

```
main() {
  var y,z;
  z = input;
  y = 1;
  if (z == 0) {
   y = 2;
  }
  else {
   y = 3 / z;
  }
  return y;
12 }
```

Figure 3.1 shows the CFG of this program.

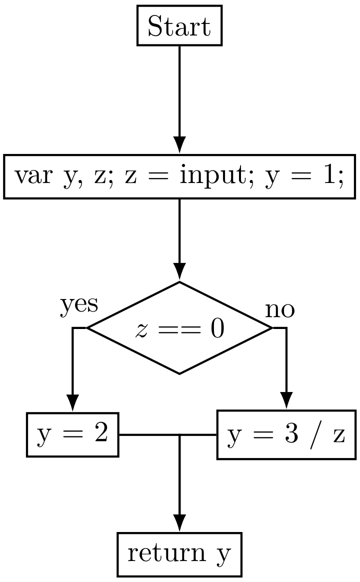


Figure 3.1 An example of a control flow graph for program 3.1

We can see that the CFG contains one *control* node because of the `if` statement in the source program. Also, note that the statements from lines 2, 3, and 4 are grouped into one node because they are always executed sequentially.

3.2 Symbolic execution

Concrete execution is the usual way of executing a program. All the values acquired from the environment, called input values (for example, a user input or a configuration file), hold a concrete value. The random fuzzing-based methods usually execute a target program with concrete execution. The executed program is called a fuzzed program. The methods try to find bugs by generating random concrete inputs for each input variable of the fuzzed program.

Symbolic execution is another way of evaluating a program. Instead of all variables holding a concrete value, they can also hold a symbolic value. A symbolic value represents multiple possible concrete values. The set of different concrete values of program input variables is typically extremely large, and thus, proving the presence of a bug by running the program for every possible combination of input values is impossible. Symbolic execution can help by analyzing the execution paths because it represents all these concrete values with one symbolic value.

An execution path in a program is a unique sequence of statements beginning at the entrypoint of the program and ending at the exitpoint. In a program represented as CFG, an execution path is a unique path through the graph from the entrypoint to an exitpoint.

Assume the program in code snippet 3.1 whose CFG is 3.1. Table 3.1 shows both the paths present in the program and the statements that are part of these paths.

Table 3.1 Paths present in a basic program 3.1 (CFG 3.1)

paths	statements
path1	var y, z; z = input; y = 1; z == 0; y = 2; return y;
path2	var y, z; z = input; y = 1; z == 0; y = 3 / z; return y;

A concrete execution of a program leads to one path being executed. Notice that different values of input variables can still lead to the same path being executed if all conditions are evaluated the same way. Symbolic execution explores a path abstractly using symbolic values, so the correctness of the program for all the different concrete input values is checked in one analysis of the path.

This means that in the presented example 3.1, we only have to analyze two paths listed in 3.1.

3.3 Symbolic state

The symbolic state represents the current situation of the symbolic executor.

It consists of the following parts.

Program location The symbolic state remembers the current location in the CFG.

Path condition A path condition is an expression that represents all the decisions made so far in executing the current path.

We define an update operation for a path condition. Updating a path condition with an expression results in a conjunction of the old path condition and the expression.

Symbolic store A symbolic store is used in the symbolic executor when we need to get a value of a variable, a record field, an element inside an array, or a value referenced by a pointer.

The symbolic execution engine starts the analyses by creating a symbolic state with an empty symbolic store, a path condition having a value `true`, and the program location being the entrypoint of the program.

Now assume that we start to symbolically execute the program 3.1, and we will try to show that it is correct. Table 3.2 shows the initial state we create.

Table 3.2 Symbolic execution of a basic program 3.1 (CFG 3.1) step1

state	program location	path condition	symbolic store
s1	start	`true`	

The basic workflow of a symbolic executor is such that the statements at the current program location are examined, the path condition and the symbolic store are modified based on the statements, and then we move to the next program location. Since the

Table 3.3 Symbolic execution of a basic program 3.1 (CFG 3.1) step2

state	program location	path condition	symbolic store
s1	var y, z; z = input; y = 1;	`true`	

entrypoint does not contain any statements, we can move to the next node in the CFG graph, as table 3.3 shows.

A variable declaration statement makes the symbolic store register new variables. An assign statement updates the value of a variable in the symbolic store. Thus, we can apply the effects of the statements `var x, z;`, `z = input;` and `y = 1;`. Notice that the value of the variable `z` has to be represented as a symbolic value. We will call it k. Table 3.4 shows the symbolic state after we move to the next statement.

Table 3.4 Symbolic execution of a basic program 3.1 (CFG 3.1) step3

state	program location	path condition	symbolic store
s1	$z == 0$	`true`	$y \Rightarrow 1, z \Rightarrow k$

We reached a *control* node. The condition can be evaluated either as `true` or as `false`. Thus, we will split the execution into two states. One state will continue executing the `then` branch with its path condition updated with the condition. The other state will continue executing the `else` branch, with its path condition being updated with the negation of the condition. The states can be seen in table 3.5.

Table 3.5 Symbolic execution of a basic program 3.1 (CFG 3.1) step4

state	program location	path condition	symbolic store
s1	y = 2	$z == 0$	$y \Rightarrow 1, z \Rightarrow k$
s2	y = 3 / z	$z! = 0$	$y \Rightarrow 1, z \Rightarrow k$

Assume that the state `s1` is explored first. The variable `y` is updated, and then we move to the exitpoint, so we can stop exploring this state and start exploring the state `s2`.

The statement `y = 3 / z` potentially causes an error if $z == 0$. However, the path condition $z! = 0$ makes this situation impossible, so we update `y` in the symbolic store and move to the exitpoint. The mechanism of checking the satisfiability of expressions is further discussed in 3.4.

There is no other unexplored path, proving that the program cannot crash for any input value.

Notice that every state corresponds to one path in the program. The benefit of this approach is that if two paths share the same sequence of instructions at their beginning, the sequence is analyzed only once.

3.4 Constraint solving

In the example 3.1, it is necessary to check the possibility of a division by zero error. The error occurs if the variable `z` equals 0. Recall that the path condition was `z != 0`. Thus, to check whether an error is possible, it is necessary to check whether the constraint `z == 0 && z != 0` is satisfiable.

There are various implementations of a constraint solver. Many of them utilize different strategies and work best for different kinds of constraints. A strategy proposed in the paper [4] tries to utilize different strengths of various constraint solvers by calling them in parallel and waiting for the first result.

Symbolic execution presents more cases when a constraint solver might be used. Assume an array index expression that accesses an element of some array. The index is a symbolic value. The constraint solver can be used to check whether an index can be out of bounds of the array. Other potential uses for the constraint solver will be discussed in chapter 4 and in chapter 5.

3.5 Unbounded loops

Unbounded loops are those that do not have a predetermined number of iterations.

Their opposite, bounded loops, can be unrolled, meaning that the statements within the loop are extracted outside it. The loop is removed, and the block of statements from the body of the loop is copied as many times as the maximum number of iterations. However, we still might have to have conditions around each if it is a bounded loop that can finish earlier than the maximum number of iterations.

On the other hand, the unbounded loops present a big challenge to the symbolic executor. After an iteration of the loop body, we split the current state into two. One path executes the body of the loop again, and one leaves it. This process can be done an unknown number of times. Thus, there can be an infinite number of paths in the program when an unbounded loop is present.

Code snippet 3.2 shows an example of a small microc program that includes an unbounded loop.

Code listing 3.2 An example of microc program with while and if statements

```
main() {
  var i, n;
  n = input;
  i = 0;
  while (i < n) {
     if (i < 100) {
         output 1;
     }
     else {
         output 2;
     }
     i = i + 1;
  }
  return y;
}
```

The CFG of this program can be seen in figure 3.2.

Notice that if we go through the `yes` edge at node $i < n$, we always reach the node $i < n$ again. Thus, unless $i < n$ becomes unsatisfiable, we are stuck in the body of the

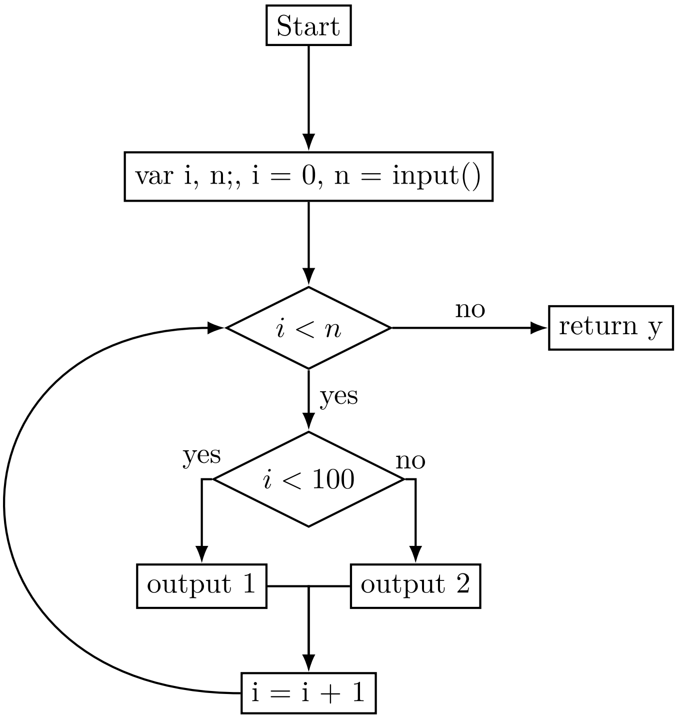


Figure 3.2 An example of a control flow graph of the example program with a while statement for program 3.2

loop for an unbounded number of iterations.

3.6 Path scheduling

The section 3.3 discussed that we sometimes have multiple discovered but yet unexplored states during symbolic execution. We add all such states into a data structure called worklist. Generally, since running an unbounded number of states in parallel is impossible, the worklist is always needed. If the number of states in the worklist is small, sufficient hardware allows us to explore them all in parallel.

The ordering in which the paths are explored is essential because a bad ordering might cause the executor to be stuck in the same place in the program and not explore other parts. This may happen with loop bodies, when a strategy may spend too much time executing a big number of iterations of a body of a loop and not paths that leave the loop. A search strategy is the function that picks the next state to execute from the worklist. The article [5] presents the following search strategies.

The DFS strategy chooses a new path to explore by choosing the last encountered unexplored path. The worklist can then be implemented as a stack. The strategy is memory efficient but runs into problems when dealing with loops and recursive functions.

The BFS strategy chooses the first encountered path, that is not yet explored. We can implement it with a queue. Despite the higher memory usage, the ability to not get stuck in a loop usually makes this strategy better than DFS.

The random strategy chooses the next state randomly from a set of discovered but not explored states. Usually, using a uniform distribution is possible.

Another strategy is used in Klee [6]. During the symbolic exploration, they create a graph whose vertices are the symbolic states created during the execution. A directed edge is added from a source node to a target node when the target state is created during the execution of the source state. The graph is a tree, and its root is the first state that the symbolic executor tries to explore. We call the graph an Execution Tree.

The execution tree is kept so that all leaves belong to states that have yet to be explored. The next state to explore is picked by traversing the tree randomly from the root until a leaf is reached. After the new state is taken, nodes are removed from the tree, so all the leaves in the tree are those still unexplored states.

This strategy favors states that are closer to the root of the symbolic execution tree and usually have simple path conditions. The symbolic execution engine that uses this strategy does not get stuck in loops and prefers states that do not repeat the loop body many times in a row.

A random strategy based on non-uniform weights is also possible. The algorithm for assigning the weights should assign bigger weights to the states that we consider more interesting. Some tools, such as [6], assign higher probabilities to states expected to increase coverage. Thus, the strategy should prioritize the paths that leave loops early.

Other works [7] try to prioritize states that have already found minor but not exploitable bugs. We can also remember how many times each branch was encountered and then choose the state that got split from the least encountered branch.

Moreover, all the above strategies can be easily parallelized, with small synchronization costs, and combined.

An important parameter for search strategies is how well they deal with a larger worklist. The smaller the number of states in the worklist, the less space and time the search strategy algorithm requires. Search strategies that perform more complicated work to pick a state from the list can achieve bad performance for large programs.

3.7 Properties of symbolic execution

Soundness is the ability of an analysis to prove the absence of errors. Thus, a sound analyzer can find all errors that are in the program. On the other hand, completeness is the ability of an analysis to prove the presence of errors. Thus, if the analyzer signals an error, this error is really present in the source code.

Notice that an analysis that never detects errors is complete but unsound, and an analysis that considers any piece of code to contain an error is sound but incomplete.

Theoretically, if we can explore all paths of the program and if our constraint solver can solve all constraints, the symbolic execution is both sound and complete,

However, in practice, symbolic execution is usually unsound because we cannot explore all paths in the program due to the path explosion. The symbolic execution is usually complete, but some design decisions can make it incomplete. For example, we can decide to detect a possible error if a constraint can not be solved.

The most significant advantage of symbolic execution is its precision. With standard symbolic execution, there are no false positives. It is also important to note that the set of execution paths is smaller than the set of different combinations of concrete inputs.

Symbolic execution can suffer from several different problems.

The first and the most important is the path explosion problem. This problem means that every conditional statement in the program can potentially increase the number of paths to explore twice. Unbounded loops can even generate an infinite amount of paths. Thus, the program generally contains an exponential number of paths compared to its size. The strategies to deal with the problem are further discussed in the chapter 4.

Constraint solving is a complex problem. Since our path conditions get bigger and bigger during the symbolic execution, the constraint solver is under more pressure and can become a bottleneck. Furthermore, using more complicated expressions than the basic arithmetic ones can be problematic. This problem will be further discussed in the chapter 4.

The other challenge is handling the interactions with the environment correctly. All the individual system calls must reasonably update the affected variable without sacrificing precision [5].

The presented challenges often force the symbolic executors to suffer from the lack of scalability.

Chapter 4

Nested loops and solving the path explosion

As discussed in chapter 3, loops provide a severe challenge for symbolic execution because a loop can generate an infinite number of paths. A loop can also have a complicated body with many different paths within its body itself, increasing the complexity of the analyses.

Similarly to loops, recursive functions also repeat the execution of the same set of statements, and the number of these executions may depend on input variables and be unbounded.

This chapter focuses on path explosion reduction techniques that might be useful for a loop-heavy program. Any recursion can be rewritten using loops, so the techniques are also relevant for solving the path explosion problem for recursive functions.

The article [5] suggests splitting the input program into parts, such as individual functions, and analyzing each part in isolation may greatly decrease the number of paths we need to check. The problem with this technique is the possibility of encountering false positive inputs. Some inputs can cause an error in our code fragment, but a deeper analysis of the whole code could show us that these values can never reach our code fragment.

Other works, such as [8], deal with the unbounded loops by unrolling the loop multiple times and trying different unrollings. However, this approach generally results in an incomplete analysis unless we try all the possible unrollings.

This chapter consists of four sections, each dedicated to a technique used to tackle path explosion. The section 4.1 talks about path pruning, the section 4.2 talks about path subsumption, the section 4.3 talks about state merging, and the section 4.4 talks about loop summarization.

Each technique is first discussed theoretically, and then an example showing the benefits of the technique is presented.

4.1 Path pruning

The most basic idea for discarding unnecessary paths is to remove those whose path conditions become unsatisfiable. The technique is discussed in the article [5]. Consider the following code snippet 4.1:

Code listing 4.1 An example of the pruning of unrealizable paths

```
 1 main() {
 2   var a, res;
 3 
 4   a = input;
 5 
 6   if (a > 1) {
 7      res = someFnc();
 8   }
 9   if (a < 0) {
10      res = someFnc2();
11   }
12 
13   return res;
14 }
```

For the sake of simplicity, assume that the functions `someFnc` and `someFnc2` perform some complicated computations but that the paths are not split during the execution of these functions.

The initial path reaches line 6 and encounters an `if` statement, which consists of two branches. The former is the `then` branch, and the latter is the `else` branch. In the presented example, the `else` branch is empty. The path gets split into two to explore both branches. Consider the `else` branch path first. Its path condition is `a <= 1`, and thus, the expressions `a < 0 && a <= 1` and `a >= 0 && a <= 1` are both satisfiable, and we split the path again on line 9.

The `then` path split on line 6 has the path condition of `a > 1`. An expression `a < 0 && a > 1` is unsatisfiable. A constraint solver is used to check the satisfiability. Thus, we can prune the `then` branch path on line 9 and continue only with the `else` branch path. In the presented example, it means one less call of a potentially computationally demanding function `someFnc2`.

4.2 Path subsumption

Subsumption is a relationship between two formulas. When formula `A` subsumes formula `B`, any interpretation that makes `A` true makes `B` also true. The path subsumption is a technique inspired by this concept.

The paths that have already been explored can be used to collect conditions associated with a program location. The conditions are created so that if the conjunction of the negations of all conditions is unsatisfiable, it is impossible for an error to happen in the later exploration of the path. Thus, the path can be pruned. Notice that the pruned path subsumes the condition since there is no combination of variable values for which the state would be valid, and the condition non-satisfiable.

The article [9] presents a possible implementation of path subsumption. It expects that the program has only explicit error locations, meaning that the only statement that can produce an error is a special error statement. When a path finishes its execution without finding an error, the program locations are annotated with conditions under which it is known that no further error would be reached from the program location.

The annotations are added during backtracking from a path that finished its execution towards the entrypoint of the program. The statements are visited in the reversed order compared to how they were evaluated. The initial annotation is `true`, meaning that all future paths should always stop the exploration when they encounter this statement. Other annotations are computed from the annotations of the successor statements.

To compute an annotation of a conditional statement, the annotation from the first statement in the `then` branch is combined with the guard of the condition. Combining expressions in this context means creating a conjunction. Similarly, the annotation from the first statement in the `else` branch is combined with the negation of the condition. The computed annotation is a disjunction of these two expressions.

To get an annotation of an assign statement, the annotation of the successor statement is modified by replacing all occurrences of the left side of the assign statement with the right side.

The annotation is copied from the successor statement for the other types of statements with only one successor.

Assume the following program 4.2 in microc. Notice that the program is written so it has only explicit error locations.

Code listing 4.2 An example of the path subsumption optimization

```
main() {
  var x, y;
  x = 0;
  if (input) {

  }
  else {
    y = input;
    if (y < 0) {                // (x - y >= 0 and y < 0)
                                //      or (x + y >= 0 and y >= 0)
      y = 0 - y;                // x - y >= 0
    }
    x = x + y;                  // x + y >= 0
  }
  if (x >= 0) {                 // x >= 0

  }
  else {
    x = 1 / 0;
  }
  return 0;                     // true
}
```

Assume that when our symbolic executor encounters a conditional statement, it follows the `then` branch first and explores the paths in the worklist with a DFS search strategy. Notice that the `else` branch in the last conditional statement is unreachable because the variable `x` is always greater or equal to zero.

The initial path executes the `then` branch on line 5 and the `then` branch on line 16. On line 21, the path is stopped, and an annotation whose value is `true` is added to line 21. Then, the path backtracks to line 15, where it tries to execute the else path, but the path gets pruned because the value of the variable x in the symbolic state is 0, and thus, the condition `x < 0` is unsatisfiable.

Thus, the path proceeds to compute the annotation for line 15. The annotation from the `then` branch is combined with the guard of the conditional statement into a conjunction. The first statement in the `then` branch is line 21, and its annotation is `true`. By combining `true` with the guard `x >= 0`, we got an annotation `x >= 0`. Recall that the annotation signals to the future paths that no error can be found if `x >= 0` is always `true`.

After placing the annotation, the path backtracks to the conditional statement on line 4, and the `else` path starts its execution. The statements on lines 8, 9, 11, and 13 proceed to be executed. Notice that on line 13, the value of the variable `y` is always greater or equal to zero. Thus, after executing line 13, `x` is greater or equal to zero. After line 13, the line 15 is reached. This line already contains an annotation `x >= 0`. The annotation is always `true`, given our symbolic state. Thus, we stop the execution of the current branch, and we backtrack.

Line 13 has only one successor, with the annotation `x >= 0`. Since the statement updates the variable `x`, the occurrences of `x` in the annotation are replaced with the right side of the assign statement. Thus, the annotation that is placed on line 12 is `x + y >= 0`.

On line 9, the `else` branch starts its execution. The first statement encountered is line 13, which was recently annotated. In the current symbolic state, `x` is 0, while `y` is a symbolic value. The current path condition is `y >= 0`. Thus, `x + y` is always greater or equal to zero, and the current path can start backtracking.

The annotation for line 9 is computed as `(x - y >= 0 and y < 0) or (x + y >= 0 and y >= 0)`

Finally, the path backtracks through other statements to the program entrypoint, and there is no other path to explore, so line 19 is proved unreachable.

4.2.1 Subsumption and unbounded loops

However, the presented algorithm is not able to handle unbounded loops. This is because we would need to run infinite iterations of the body of the loop to collect the annotations.

A new variable `_t` is introduced to handle unbounded loops. This variable is decreased by one at the end of the body of the loop, and every program point within the loop is annotated with a temporary annotation `_t < 0`. Thus, all paths leave the loop body after `_t + 1` iterations. For example, the variable `_t` can be zero. Thus, all paths that reach the loop body perform one iteration and then leave the loop.

When the path that performed one iteration of the body of the loop and all the paths split from it after the loop finishes their execution, the path backtracks to the beginning of the loop. The computed annotations are an approximation of the fixpoint. The inductivity of the annotations of the statements within the loop is checked.

The algorithm described in [9] first plugs zero to variable `_t` in the annotation and checks whether the expression is satisfiable. If so, it checks whether the expression is always satisfiable after `n + 1` iterations of the loop if it is also satisfiable after `n` iterations, where `n >= 0`.

If one of the induction check steps fails, the annotation is removed. Then, the value of `_t` can be increased, and the algorithm is repeated.

If the annotation is kept instead, the annotations `_t < 0` are removed.

Now assume the following program 4.3.

Code listing 4.3 An example of path subsumption and loops.

```
 1 main() {
 2   var x, y, i, n;
 3   x = input;
 4   i = input;
 5   n = input;
 6   y = x;
 7   while (i < n) {
 8      x = x + 1;
 9      i = i + 1;
10   }
11   if (x < y) {
12      x = 1 / 0;
13   }
14   return 0;
15 }
```

The loop on line 7 is unbounded, and even though the error statement on line 12 can not be reached, the default symbolic execution would not terminate because the loop creates an infinite number of paths. Subsumption can be used to deal with such cases.

When we first reach the loop, we can try to execute it only a fixed number of times, collect these incomplete annotations, and check whether they are generally applicable for any number of iterations of the loop. This checking is done by induction.

The code snippet 4.4 shows the loop from the previous example with the variable `_t` added.

Code listing 4.4 An example of path subsumption and loops.

```
  _t = number_of_iterations_of_the_loop - 1;
  while (i < n) {
     x = x + 1;            // _t < 0
     i = i + 1;            // _t < 0
     _t = _t - 1;
  }
```

All statements within the loop are annotated with an expression `_t < 0`. At the end of the loop, the variable `_t` is decremented. After the fixed amount of iterations of the loop, the symbolic executor is forced to leave the loop.

After exploring the whole execution subtree, we backtrack. The annotation `x >= y` is computed on line 11. Then, we backtrack to line 7. We copy the annotation from line 11 to line 7. Only then do we backtrack to the loop body and annotate its statements based on the annotation from the successor states of each statement. Thus, every statement within the loop is annotated with `x >= y && _t >= 0`. When we reach the top loop statement on line 7, we take the annotations of all statements within the loop and replace all occurrences of the variable `_t` with zero. Thus, all annotations are transformed to `x >= y`. Then, we check whether the annotations are inductive. In our case, the annotations are inductive because they are satisfiable if only one iteration of the body was performed, and also it is true that if `x >= y` is satisfiable after `k` iterations of the loop, then it is definitively also satisfiable after `k + 1` iterations.

An example of an annotation that would not be inductive for our loop is `i < n`. It is possible that `i < n` is always true after `k` iterations but not after `k + 1` iterations.

Now, we check whether the computed annotation for line 7 is subsumed by our path condition. In our case, `x >= y` can not be evaluated to `false`, so the current path is stopped.

4.2.2 Path subsumption for nested loops

The construct we are particularly interested in is the nested loops. The code snippet 4.5 shows one.

Code listing 4.5 An example of path subsumption and loops.

```
main() {
  var x, y, i, j, n;
  x = input;
  i = input;
  j = input;
  n = input;
  y = x;
  while (i < n) {
     i = i + 1;
     while (j < n) {
         x = x + 1;
         j = j + 1;
     }
  }
  if (x < y) {
     x = 1 / 0;
  }
  return 0;
}
```

The nested loops are handled similarly to the simple loops. There are several different `_t` variables introduced. Each of them is for one loop. Thus, the statements that are part of multiple loops have multiple annotations of type `_t < value`.

4.3 State merging

As discussed in chapter 3, the paths that share the same sequence of statements at their beginning have these statements being analyzed together. That is the reason why the paths are split during the execution. There are probably paths that share the same sequence of instructions at different stages of the program other than its beginning. Those paths can be merged and analyzed together.

The following snippet 4.6 shows an example of a part of a program where state merging could be very beneficial.

Code listing 4.6 An example of beneficial state merging

```
if (input) {
    output 0;
}
else {
    output 1;
}
...
```

The `if` statement splits the execution into two parts. If multiple paths reach the `if` statement, the number of the paths leaving the statement is twice as big. However, the paths share the same statements that they must explore in the future, so it would be great if the exploration could be done once for multiple paths.

The paths to be merged must currently be at the same program point. When two states are merged, variables whose symbolic values are the same are reused in the merged state. An `ite` (if-then-else) expression is constructed for those whose symbolic values differ. The `ite` expression contains two values and an expression. If the expression evaluates to `true`, then one of the values is used. Otherwise, the second value is used. The path condition of the merged state is a disjunction of initial path conditions. The `ite` expressions may be hard for a constraint solver to handle.

As the article [10] discusses, the effect of merging states into a symbolic state can also be described in the following way. Without state merging, the states are kept as a disjunction. For example, imagine a symbolic executor that currently has two symbolic states. The overall state of the symbolic executor can be expressed as a disjunction of those states. In the following example, the states are called `s1` and `s2`. The field `loc` in a symbolic state is the program location of the symbolic state. The field `cond` is the path condition, and the field `store` is the symbolic store.

```
executionState = s1(loc1, cond1, store1) or s2(loc2, cond2, store2)
```

To merge the two states, the disjunction of two states can be replaced with a state whose fields are disjunctions of the fields of the original states.

```
executionState = merged(loc1 or loc2, cond1 or cond2, store1 or store2)
```

The program location must be the same in both to-be-merged states for the merged state to be executable. The merged path condition is a disjunction of the initial path conditions. The variables stored in the merged symbolic store are a disjunction of the values of the variables in the initial stores. Those variables that are the same in both states can be simplified to a simple value because `(a || a) == a`. Other variables in the new symbolic store must be encoded with an `ite` expression. The `ite` expression is essentially just a disjunction, where the expression condition adds additional knowledge about the value.

It can be observed that when a path is split upon reaching a branch and the two new paths are immediately merged, the path condition of the merged state is the same as before the split. Thus, the technique is particularly effective for conditional statements and loops, whose bodies do not affect the symbolic states. The code snippet 4.9 is an example of such a loop.

If the states are merged immediately after the `if` statement, the merged state does not contain any `ite` expression, and the path condition is the same as before the merge.

Generally speaking, while the technique can significantly reduce the number of paths, both the symbolic values and path conditions can become more complicated, thus making the state more computationally demanding for the symbolic executor. Thus, the optimization effect of state merging can even be negative.

One such problem arises when a variable whose value is symbolically represented with an `ite` expression is part of the state of the path and when the path reaches a branch containing this variable. In such a case, we would have to rely on our constraint solver and its ability to solve this constraint efficiently, which may be a problem.

Moreover, symbolic values can replace some concrete values during the state merge. This can happen, for example, when merging concrete values assigned to a variable in different branches of a conditional statement. In such a case, we might be forced to call the constraint solver more times than without the actual state merging. We can see the problem in the following example:

Code listing 4.7 An example of possibly unbeneficial state merging

```
a = 0;
if (input) {
    a = 1;
}
else {
    a = 2;
}
...
return 1 / a;
```

First, assume that no state merging is used. The execution is split into two states in the `if` statement. When the return statement is reached, the value of `a` is checked with a simple number comparison because `a` is a concrete number in both states.

If the states are merged directly after the if statement, the value of the variable `a` is an `ite` expression. In the small presented example, the values within the `ite` expression could be compared to zero, and thus, it could be decided whether an error is possible. However, for more complicated constraints, it is necessary to call the constraint solver to check the possibility of the error. One call to a constraint solver is always more expensive than two simple comparisons performed without the state merging.

An extreme case of state merging that occurs when the created states are merged right away after we perform a split is called static state merging. Symbolic execution then becomes similar to verification condition generation [10].

The other extreme case on the other side of the spectrum is the basic symbolic execution, which performs no state merging.

The tradeoff on this spectrum is between reducing the state space by merging many states and not having any complicated states with no states merging. When we have complicated states, we rely on the ability of the constraint solver to solve more complicated constraints efficiently.

4.3.1 Query count estimation

A technique presented in [10] does not pick its place on the merging spectrum globally. Yet, it automatically detects whether a few more complicated states or a bigger number of simpler states will likely be more efficient in a particular case. Generally, solving a constraint that contains conditions with variables consisting of `ite` expressions slows down the executor considerably. Thus, we can preprocess a program with a static analysis technique, which computes how often the program variables are used in a constraint solver after a program point.

The number of solver invocations for a variable and a program point is computed as a sum of solver invocations for a variable in the successor statements in the CFG. This number is increased by the number of solver invocations of constraints containing the variable the current line causes. Thus, for the program without loops, the algorithm is as follows.

```
compute_future_solver_invocations(variable, location) {
    sum = 0;
    for (successor in location.successors) {
        sum += compute_sum(variable, successor)
    }
    sum += number of solver invocations containing the variable
            at the current program location
}
```

The technique is controlled by several parameters. One of them is named κ, and it is used as a global limit for a maximal number of iterations of any loop.

Assume the following code:

Code listing 4.8 An example of query count estimation with κ

```

main() {
  var r, arg, argc, argv, i;

  r = 1;                          // {}
  arg = 1;                        // {r -> 6}
  argc = input;                   // {r -> 6, arg -> 24}
  argv = [];                      // {r -> 6, arg -> 24, argc -> 9}
  if (arg < argc) {               // {r -> 6, arg -> 24, argc -> 9,
                                      // argv -> 7}
     if (argv[input]) {           // {r -> 3, arg -> 16, argc -> 6,
                                      // argv -> 5}
        r = 0;                    // {arg -> 8, argc -> 3,
                                      // argv -> 2}
        arg = arg + 1;            // {r -> 3, arg -> 8, argc -> 3,
                                      // argv -> 2}
     }
18   }
19   while (arg < argc) {          // {r -> 3, arg -> 8, argc -> 2,
20                                     // argv -> 2}
21      i = 0;                     // {r -> 2, arg -> 7, argc -> 2,
22                                     // argv -> 2}
23      while (argv[arg][i] != 0) {// {r -> 2, arg -> 7, argc -> 2,
24                                     // argv -> 2, i -> 5}
25         i = i + 1;              // {r -> 1, arg -> 4, argc -> 1,
26                                     // argv -> 1, i -> 3}
27         output argv[arg][i];    // {r -> 1, arg -> 4, argc -> 1,
28                                     // argv -> 1, i -> 3}}
29      }
30      arg = arg + 1;             // {r -> 1, arg -> 1, argc -> 1}
31   }
32   if (r) {                      // {r -> 1}
33      output 10;                 // {}
34   }
35
36   return 0;                     // {}
37 }
```

Assume that we want to merge two states whose current instruction is the beginning of the outer while loop on line 19. The states differ in the value of the variable `arg`. Thus, the maximal amount of future constraint solver invocations of constraints that contain the variable `arg` should be computed. The κ is set to one.

The constraint solver uses the variable `arg` on line 19 to check the satisfiability of the condition. The numbers computed for the successor lines 21 and 32 for the variable `arg` are added to this number. On line 32, the number of future invocations using the variable `arg` is zero. Thus, the value for line 19 would be one if no iteration of the body of the outer loop was performed.

We will now compute the number for the body of the loop. The value on line 21 is the same as on line 23. The condition of the loop on line 23 adds two solver invocations. This is because we need to use the solver to check a possible array access out-of-bounds error, and then we need to use the solver to check the satisfiability of the guard.

We also add the values from the successor states on lines 25 and 30. The value on line 30 is like the value on line 19 if no other iteration of the body of the loop was performed. We already computed this number as one. Thus, the value for line 23 would be three if no iteration of the inner loop body was performed.

The line 27 adds another invocation. However, the expression causing it on line 27 is the same as on line 23, so running an available expressions analysis could remove the need for this invocation. We also have to add again value for line 23 if no iteration of the body was performed. Thus, assuming no available expression analyses were performed, the number of possible future solver invocations on line 27 is 4. Thus, the number of line 25 is 3 is 4. The final number for line 23 is the sum of the values for line 30 (1), line 25 (4), and the number of solver invocations on line 23, which is 2. Hence, the value for line 21 is also 7, and on line 19 it is 8. Note that the real number of invocations can be higher, and the maximum is not bounded. A bigger value of κ would result in a bigger number being computed.

4.3.2 Dynamic state merging

Merging can coincide with some state search strategies. Merging benefits from repeating the body of a loop multiple times in a row and then merging the states that left the loop. However, a coverage-guided search strategy wants to cover the less explored parts of the program first. The dynamic state merging presented in the article [10] aims to solve this problem. The technique uses a merge strategy that chooses those states from the worklist that are expected to become mergeable with some states in the worklist soon.

Assume that we store the predecessors of each state. When we check whether a state `s1` can become mergeable with another state `s2` in the near future, we can check whether the state `s1` is mergeable with a predecessor of `s2`. If so, we can prioritize exploring `s1` because there is a high chance that we will be able to merge it with `s2` in the near future.

4.4 Loop summarization

We have already dealt with the fact that unbounded loops in a program can potentially generate an infinite number of paths. Notice that even though only one path reaches such a loop, there may be infinite ways of how many times the loop iterates and what path within the loop is taken in each iteration. The basic idea of loop summarization is to compress all the states that can emerge after leaving the loop body into a small set of states, so this set of states still represents all the original states without a loss of precision. This is not doable for all loops, as discussed later in this chapter.

Successful loop summarization analyses result in a loop summary. A requirement for computing a loop summary is that the effect of every statement in a loop can be captured as a function whose parameters are the initial values of the variable before the loop and the number of times the statement is invoked.s. The function returns a symbolic expression.

The summary is a structure that captures all possible patterns in which a loop can be executed. Every such pattern is called a trace. A trace contains a function that transforms the initial value of a variable into a symbolic expression that encapsulates the possible values of the variable after leaving the loop. A trace also has its own condition. It represents the constraints on the variables if the loop is executed in a pattern the trace captures.

To apply the loop summary to a symbolic state, the state is duplicated, so there is one state for each trace. The path condition of each state is updated, so the new condition is a conjunction of the old condition and the trace condition. The functions associated with a trace are used to update the variables in the symbolic store.

A state that emerges after applying a trace can be imagined as one created by merging many original states. The states are not merged by using `ite` expressions but by using symbolic values and specifying which real values the symbolic values can hold within the path condition.

After the summary is applied, all the created states are added to a worklist, and the symbolic executor continues.

Subsection 4.4.1 describes how to summarize loops contacting just one inner path.

Subsection 4.4.2 extends the technique to support conditional statements.

The last subsection 4.4.3 discusses the summarization of loops containing other loops.

4.4.1 Summarization of single-path loops

First, only assume loops that do not contain any statements, such as conditional statements or loops that could split the execution. Thus, there is only one path within the loop. In the case of the microc language, it means that the loop does not contain any `if` or `while` statements.

A simple concept of path counters [4] can be utilized to create loop summaries. A path counter is a variable assigned to a path within a loop that says how many times the path is iterated.

Statements can be split into multiple categories, given how they interact with the symbolic executor after multiple iterations of the statement within the loop body.

First, there are statements whose effect on the symbolic state is the same after any number of iterations of the path. An example of such a statement in microc might be the output statement, which does not affect the state. Another example is an assignment of a constant, which does not change within a loop. After multiple iterations, the effect of these statements can be summarized by applying the statement once.

Then there are the assign statements, whose effect can be summarized with a mathematical formula using the path counters. Statements that add a value to a variable can be summarized based on the initial value of the variable and the value of the path counter. The variables that are updated in such a way are called induction variables.

The following example shows a loop with several summarizable statements. The number of iterations of the body of the loop is unknown.

```
while (input) {
   a = a + inc;   ->   a = initial_a + inc * path counter of the path
   b = b - inc;   ->   b = initial_b + inc * -(path counter of the path)
   e = e;         ->   e = initial_e
}
```

The example shows how incrementation statements of various operators can be summarized. Each line within the loop contains a statement on the left side, while the right side shows a formula computing all possible values of the variable after the loop finishes. The `path counter of the path` is a symbolic value. For this to work, the variable `inc` has to behave as a constant within the loop and must not be updated within the loop.

The consequences of summarization of simple loops are shown in the following example 4.9.

Code listing 4.9 An example of loop summarization

```
main() {
  var a, i, n;
  a = 0;
  i = input; // v1
```

```
   n = input; // v2

   while (i < n) {
    a = a + 2;
    i = i + 1;
   }

   if (a == 15) {
    some_fnc();
   }
   else {
    some_fnc2();
   }

   return 0;
  }
```

The code snippet contains variables `i` and `n`, which are part of the guard of the loop. Assume their initial values are symbolic values `v1` and `v2`, respectively.

The loop is summarized into two traces. The first one does not iterate the loop body at all. A symbolic state that gets updated with this trace does not update its symbolic store since no statements are evaluated. The path condition of the state is updated with an expression `v1 < v2`.

The other trace is created to encapsulate iterating the loop body an unknown amount of times. The loop updates the variables in the following way. The variable `n` is not updated within the loop, and its value remains `v2`. The variable `i` is incremented until it holds the same value as the variable `n`. The formula of the update function is `i = i => i + path_counter`.

The last variable, `a`, increases by 2 in every loop iteration. Thus, the formula is `a = a => a + 2 * path_counter`.

Thus, when the formulas are applied with the initial values of the variables before the loop is summarized, the new value of `i` is `v1 + path counter`, and the new value of `a` is `2 * path_counter`.

Table 4.1 shows the update functions for the traces.

Table 4.1 An example of the traces of a summary computed for a loop 4.9

traces	**update functions**
t1	
t2	$i = i \Rightarrow i+$ path counter $a = a \Rightarrow a + 2*$ path counter

The path condition also gets updated with an expression `v1 + path counter == v2`.

The condition `a == 15` of the `if` statement is not satisfiable since `2 * path_counter == 15` is not satisfiable for integers.

4.4.2 Summarization of multi-path loops

This subsection discusses the summarization of loops that may contain a conditional statement. In such loops, there are multiple ways of executing the loop body. The analyses of such loops are discussed in the article [11].

Whether we can summarize a multi-path loop depends not only on our ability to summarize individual statements within the loop but also on the pattern of the interleaving of the paths within the loop and the type of all the conditions within the loop.

The loops are classified into four types, depending on how the paths within the loop interleave and how the conditions within the loops behave. Table 4.2 shows how the loops are classified.

Table 4.2 Classification of loops by combination of the interleaving pattern and the type of the inner conditions.

Interleaving pattern/Conditions	**only IV conditions**	**has a NIV condition**
Sequential	Type 1	Type 3
Periodic	Type 1	Type 3
Irregular	Type 2	Type 4

The subsection 4.4.2.1 discusses the path interleaving types. The types of conditions are discussed in 4.4.2.2.

4.4.2.1 Classification of path interleaving within a loop

To classify the path interleaving within a loop, we will build its path dependency automaton [11].

A path dependency automaton (PDA) is a tuple $(S, \mathrm{Init}, \mathrm{Accept}, E)$. S is a finite set of states. Every state corresponds to a path in a loop. The states remember the condition of its path and summarized statements of the path. The `Init` set is a subset of S containing the states whose path might be the first path in the loop that is executed. The `Accept` set of states is another subset of S, containing those states that are the last executed before the executor leaves the loop. The last member of PDA is a set of edges of vertices from S. A directed edge between a source vertex and a target vertex exists when an execution of the path associated with the target vertex might immediately follow the execution of the path associated with the source vertex.

To detect this, we assume that the source path is correctly executed and its path condition is evaluated as `true`. Then, we apply the changes to the variables performed by one iteration of the source path and check whether the path condition of the target path may be satisfiable.

We choose a symbolic value `k` for the value of the path counter of the source path. We assume that if the value is `k - 1`, the condition of the path associated with the source vertex evaluates to `true`, but if the value is `k`, the condition of the path associated with the target vertex evaluates to `true`. Thus, it would be the path associated with the target vertex that would be executed. The expression `k > 0` must also be true, so the number of iterations of the path is positive.

Thus, the constraint that detects an edge between two vertices is a conjunction of the source vertex path condition after the `k - 1` iteration of the source path, the target path condition after the `k` iterations, and an expression `k > 0`. The constraint will be referred to as `cond`. If `cond` is satisfiable, we create the edge. An edge remembers the `cond` and the changes of the variables after `k` interactions of the source path.

We analyze the newly created graph as follows. We call the execution sequential if there is no cycle in it. An example of a sequential path interleaving pattern is also the loop in 4.9. If there are cycles, the execution is either periodic or irregular.

To decide whether a loop is periodic, we define the periodicity of a cycle. A cycle is periodic if all paths within it have a period. A period is a constant value that specifies, after how many iterations of the path start, the execution of the next path. If all cycles within a loop are periodic, the loop is also periodic. Otherwise, it is irregular.

The code snippet 4.10 shows an example of a loop with a periodical execution pattern.

Code listing 4.10 An example of a loop with an periodic path interleaving pattern

```
main() {
  var n, x, z;
  n = input;
  x = input;
  z = input;
  while (x < n) {          // p1, p2, p3
    if (z > x) {           // p2, p3
      x = x + 1;           // p2
    }
    else {
      z = z + 1;           // p3
    }
  }
  return 1 / (x - n);
}
```

The loop contains three paths. Table 4.3 shows the paths of the loop with their path conditions and the statements they execute. The table also contains the children of the paths in the PDA graph. The statements within the loop in the code snippet 4.10 are annotated with a comment that lists all the paths that the statement is part of.

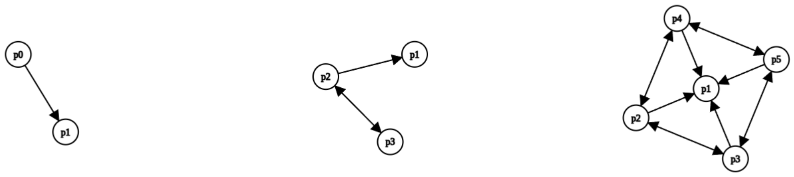
Table 4.3 An example of paths of a loop (paths graph 4.1b, code snippet 4.10) with a periodic path interleaving pattern

Paths in the graph	**Path condition**	**evaluated statements**	**children**
p1	$x >= n$		
p2	$x < n \land z > x$	$x = x + 1$	p1, p3
p3	$x < n \land z <= x$	$z = z + 1$	p2

The first path starts with the expression `x < n` being evaluated as false, and it contains no other statements. The second starts with `x < n` being evaluated as true and `z > x` as true. The statement `x = x + 1` is then executed. The last path has `x < n` evaluated to true and `z > x` evaluated to false. The statement `z = z + 1` is then evaluated. These three paths form the PDA graph.

The path `p1` has no children in the PDA graph. This is because the path condition can not be evaluated differently after an iteration of the path if the path contains no statements. The path `p2` has `p1` as a child. This is because `x < n && (x + 1) >= n` can be possibly true. Similarly, `p3` is also a child of `p2` because `z > x && z > (x + 1)` can be true. The path `p3` has the path `p2` as the child.

The figure 4.1b shows the graph. The graph contains only one cycle between paths `p1` and `p2`. The paths inside the cycle take turns after one iteration. Thus, the periods of both paths are one, and the execution pattern is periodic.



(a) A PDA graph of a loop with a sequential path interleaving **(b)** A PDA graph of a loop with a periodical path interleaving **(c)** A PDA graph of a loop with an irregular path interleaving

Figure 4.1 An example of graphs of paths in the loops

The code snipped 4.11 shows a loop with an irregular execution pattern that can not thus be summarized.

Code listing 4.11 An example of a loop with an irregular path interleaving pattern

```
main() {
  var i, j, a;
  i = input;
  j = input;
  a = input;
  while (i < 100) {
   if (a <= 5) {
     a = a + 1;
   }
   else {
     a = a - 4;
   }
   if (j < 8) {
     j = j + 1;
   }
   else {
     j = j - 3;
   }
   i = i + 1;
  }
  return 1 / (x - z);
}
```

Table 4.4 lists all the paths within the loop.

One of the paths has zero children in the graph, while all the others have four. As seen in the figure 4.1c, there are many connected cycles in the graph. The paths in the cycles can not be given constant periods. Thus, the execution pattern is irregular. The article [11] presented analyses performed on real-world programs that failed to find

Table 4.4 An example of paths of a loop (paths graph 4.1c, code snippet 4.11) with an irregular path interleaving pattern

Paths	Path condition	evaluated statements	children
p1	$i >= 100$		
p2	$i < 100 \land a <= 5 \land j < 8$	$a = a + 1; j = j + 1; i = i + 1$	p1, p2, p3, p4
p3	$i < 100 \land a <= 5 \land j >= 8$	$a = a + 1; j = j - 3; i = i + 1$	p1, p2, p3, p5
p4	$i < 100 \land a > 5 \land j < 5$	$a = a - 4; j = j + 1; i = i + 1$	p1, p2, p4, p5
p5	$i < 100 \land a > 5 \land j >= 5$	$a = a - 4; j = j - 3; i = i + 1$	p1, p3, p4, p5

any real-world connected periodic cycles. Thus, the graphs with connected cycles are regarded as irregular execution.

4.4.2.2 Classification of conditions

All conditions can be classified into two categories based on how much they complicate the analyses of the loop. A condition is converted to a form when an expression is compared to zero. If it is updated in the loop, as it was an induction variable, the condition is an IV condition. An induction variable (IV) is a variable that is increased or decreased in the loop predictably by a fixed number.

The only loop in the code snippet 4.10 contains conditions `x < n` and `z > x`.

The condition `x < n` can be converted to a form `E < 0`, where `E = x - n`. The variable `E` is updated predictably within the loop because one path increments it by one, and the other does not change its value.

The condition `z > x` can be converted to a form `E > 0`, where `E = x - z`. The variable `E` is updated predictably within the loop because one path increments it by one, and the other decrements it by one.

The following code snippet 4.12 shows a loop with a NIV condition.

Code listing 4.12 An example of a loop with a niv condition

```
main() {
  var i, j, k, n;
  i = input;
  k = input;
  n = input;
  while (i < n) {
     j = input;
     i = i + j;
     k = k + 1;
  }
  return k;
}
```

The condition `i < n` can be converted to a form `E < 0`, where `E = i - n`. The variable `E` is a non-induction variable because the value by which it is incremented can not be predicted.

4.4.2.3 Summarization of type 1 loops

To get a summary, we perform a DFS-based search on the PDA graph, traversing from the `Init` states through the transitions towards the `Accept` states. During each transition through an edge, we collect the path condition of the edge and the variable changes. A trace is constructed for each disjunctive path in the graph from an `Init` node to an `Accept` node.

First, we will look at the summarization of a single-path loop. We will reuse the example 4.9.

```
main() {
  var a, i, n;
  a = 0;
  i = input; // v1
  n = input; // v2

  while (i < n) {
    a = a + 2;
    i = i + 1;
  }

  if (a == 15) {
    some_fnc();
  }
  else {
    some_fnc2();
  }

  return 0;
}
```

The loop contains a path that executes the body of the loop and a path that leaves the loop right after the condition check.

Table 4.5 shows the edges within the PDA graph. There is only an edge from `p2` to `p1`. Both nodes in the graph are in the `Init` set, and the path `p1` is in the `Accept` set.

Table 4.5 An example of edges in a PDA graph of a sequential loop (paths graph 4.1a, code snippet 4.9)

source	target	edge condition	changes
p2	p1	$i + k - 1 < n \wedge i + k >= n \wedge k > 0$	$i = (it) \Rightarrow (init_i) \Rightarrow init_i + it$ $n = (it) \Rightarrow (init_n) \Rightarrow init_n{+}2{*}it$

The `it` parameter in the `changes` field in the table is a symbolic value representing the number of iterations of the loop. The `Init` value is the value of the variable before the loop was started to be summarized.

We collect two traces. One of them starts in `p1` and ends in `p1`. This trace captures the path where the body of the loop is not executed. Since we did not traverse any edge in the PDA graph, the path condition is updated just with the negation of the guard, and the symbolic store is not modified.

The second trace starts in the node p2 and ends in the node p1. This trace represents all the states that would be created after different numbers of iterations of the body of the loop. The path condition of the new state is updated with $i + k_1 - 1 < n \wedge i + k_1 >= n \wedge k_1 > 0$, which is the edge condition.

The values n and i are updated in the symbolic store based on the update functions associated with the edge. The changes field in table 4.5 shows these update functions.

4.4.2.4 Summarization of cycles in PDA

The Proteus framework supports the summarization of periodic cycles [11]. If a cycle is found in the PDA graph, and the execution of it has a pattern, we can summarize an iteration of the cycle.

The cycle can be interpreted as an actual state in the PDA graph. An iteration of the cycle is equivalent to sequentially applying the paths in the cycle. The number of times each path is applied depends on its period.

Assume the periodic loop in the example 4.10. The edges in the PDA graph of this loop are shown in table 4.6.

Table 4.6 An example of edges in a PDA graph of a periodic loop (paths graph 4.10, code snippet 4.10)

source	target	trace condition
p2	p1	$x + k - 1 < n \wedge z > x + k - 1 \wedge x + k >= n \wedge k > 0$
p2	p3	$x + k - 1 < n \wedge z > x + k - 1 \wedge x + k < n \wedge z <= x + k \wedge k > 0$
p3	p1	$x + k - 1 < n \wedge z <= x + k - 1 \wedge k > 0$

The only path in the Accept set is the path p1. All paths are in the Accept set. The loop has five traces. The first starts in p1 and ends in p1. Another trace starts in p2 and ends in p2. Another trace starts in p2 and then goes to p3. From p3, it returns to p2, and a cycle is detected. The periods of both states are one. The cycle p3, p2, p3 is periodic and can be abstracted into a state that performs an iteration of p3 followed by an iteration of p2. There can be any number of iterations of this cycle. After that, there is an iteration of p1, which is a state in the Accept states with no children in the graph. The traces that start in p3 can be computed similarly.

4.4.2.5 Summarization of loops of types 2, 3, and 4

Summarizing loops belonging to types 2, 3, and 4 is hard. The article [11] lists several approximation techniques that aim at summarizing at least some of these loops.

For example, input-dependent guard conditions can be evaluated as true in any iteration based on the actual input. Thus, approximating such a condition to be true is possible.

Interestingly, the authors of [11] performed an experiment on real-world programs and failed to find any actual type 2 loops.

4.4.3 Summarization of nested loops

In some cases, creating a summary of a loop containing another loop is possible. We first summarize the inner loop. The summarization of the outer loop proceeds as if the variable changes stored in the summary were actual program statements updating the variables.

The code snippet 4.13 shows an example of such a loop.

Code listing 4.13 An example of nested loop summarization

```
main() {
  var n, x, z, res, i, realX, realZ;
  n = input;
  x = input;
  z = input;
  res = 0;
  if (n <= 0) {
     n = 1;
  }
  if (x >= n) {
     x = n - 1;
  }
  if (z >= n) {
     z = n - 1;
  }
  realX = x;
  realZ = z;

  i = 0;
  while (i < n) {
     x = realX;
     z = realZ;
     while (x < n) {
        if (z > x) {
           x = x + 1;
        }
        else {
           z = z + 1;
        }
     }
     res = res + x;
     i = i + 1;
  }

  if (res == n * n) {
     res = 1;
  }
  else {
     res = 0;
  }

  return 1 / res;
}
```

Notice that the inner loop of the program is the same as in the example 4.10. Thus, we can summarize the loop in the same way. We know that if the loop body gets to be executed, then both x and z will be set to n. The initial values are initialized, so the body of the inner loop is always executed. The outer loop resets x and z to the original values and then calls the inner loop. Finally, it updates res, based on x, and increments i.

The following code snippet 4.14 shows how the structure of the outer loop can be imagined if the summarization of the inner loop is correct.

Code listing 4.14 An example of outer loop after summarization of the inner loop

```
while (i < n) {
    x = realX;
    z = realZ;
    if (x < n) {
        if (z > x) {
            x = n;
            z = n;
        }
        else {
            x = n;
            z = n;
        }
    }
    else {

    }
    res = res + x;
    i = i + 1;
}
```

This can be simplified into the code snippet 4.15

Code listing 4.15 An example of an outer loop after summarization of the inner loop (simplified)

```
while (i < n) {
    x = realX;
    z = realZ;
    x = n;
    z = n;
    res = res + x;
    i = i + 1;
}
```

The first four statements, which just set the value of a variable to the value of a different variable, are easily summarizable. The last two both increase the value of the variable by a fixed value and thus we can summarize these statements too. The outer loop should update x, z, and i to n, and the value of res should be realX * path counter. Both the path counter and realX have the value of n. The res variable should thus hold the value n * n.

Chapter 5

Implementation

This chapter discusses the implementation of the symbolic executor and the techniques for tackling path explosion.

Section 5.1 discusses the overall design of the implementation. Section 5.2 describes how microc programs are represented. Section 5.3 describes the implementation of the general symbolic executor without path explosion optimizations. Section 5.4 describes the implementation of techniques for tackling path explosion.

5.1 Design

The programming language used for the implementation is Scala 2 [12]. The code is split into the following packages.

interpreter The implementation contains an AST interpreter of the microc language developed for the NI-APR course. The interpreter is completely independent of the symbolic execution part of the implementation. It does not support arrays.

generation The package offers to generate random microc programs. It is further discussed in the `experiments` chapter.

analysis The query count analyses 4.3.1 can be performed for state merging before the symbolic executor starts.

parser The implementation of the symbolic executor consists of several parts. The program is parsed into an internal representation using a slightly modified parser provided for the NI-APR class. The parser is modified to support the operators added to the language for this thesis as described in chapter 2. No type checking is implemented, and there are expected to be no type-related errors in the source programs.

ast Contains AST classes of the microc language. The `ASTNormalizer` class present in the package can normalize ASTs as will be discussed in 5.2.1.

cfg Contains the `Cfg` and `CfgNode` classes for representing CFGs. The `CfgFactory` transforms an AST into CFG.

symbolic_execution The package contains the code of the basic symbolic executor and the techniques for tackling path explosion.

util The package contains several utility classes that were provided for the NI-APR course.

cli

The package `cli` provides a command line interface for the various provided features. The program supports The following command line options.

symbolicallyExecute The command `symbolicallyExecute` is used to symbolically execute a program. There are several techniques for tackling path explosion, but enabling more than one is forbidden. The command supports the following options.

search-strategy The parameter specifies the search strategy that is used. The possible values are `bfs`, `dfs`, `random`, `tree`, `coverage`, and `klee`. The `bfs` search strategy is the default. The implementation of the strategies is further discussed in 5.3.2.2.

merging-strategy The parameter specifies what state merging strategy is used. The possible values are `none`, `aggresive`, `lattice-based` and `recursive`. The default value is `none`. The implementation of the strategies is discussed in 5.4.3

smart-merging-limit-cost The parameter is of type `Integer`, and it is used when merging `lattice-based` or `recursive` is enabled. The default value is `1`. The parameter is further discussed in 5.4.3.2.

kappa The parameter is of type `Integer`, and it is used when merging `recursive` is enabled. The default value is `1`. The parameter is further discussed in 5.4.3.2.

summarization The parameter is of type `Boolean`, and it specifies whether loop summarization is enabled. The default value is `false`. The implementation of loop summarization is discussed in 5.4.4.

subsumption The parameter is of type `Boolean`, and it specifies whether path subsumption is enabled. The default value is `false`. The implementation of path subsumption is discussed in 5.4.2.

timeout The parameter is of type `Integer`, and it specifies for how long can the symbolic executor run in seconds. The default value is `30`.

output The parameter specifies an output folder. If some values are specified, the program uses the folder to store there the achieved path coverage in file `coverage.txt`, the time that the symbolic execution took into the file `time.txt`, and the number of found errors into `error.txt`.

generateProgram The command `generateProgram` generates random microc programs. The program generator supports the following parameters, which are then further discussed in chapter 6.

program The parameter is a path to a file where the generated program should be stored.

forLoopGenProb The parameter specifies the probability of generating an iterative loop. Its value is a `Double`, and it must be bigger or equal to zero and lower or equal to one.

A variable of type number is incremented at the end of the loop until it is at least equal to another variable of number type from the program. The loop body is generated randomly based on the generator settings.

loopGenProb The parameter specifies the probability of generating a general loop. Its value is a `Double`, and it must be bigger or equal to zero and lower or equal to one. The sum of `forLoopGenerationProbability` and `generalWhileLoopGenerationProbability` must be lower or equal to one.

A general loop is a loop whose body and condition are random. The condition and loop body are generated based on the settings of the generator.

maxBlockDepth The parameter `maxBlockDepth` defines the maximal depth of a block of statements. This is necessary because, at some point, a `while` or `if` statements must only generate statements that do not create any blocks.

maxTopLvlStmtsCount The parameter specifies the maximum number of top-level statements in the *main* function. The actual number of statements to generate is taken with a uniform probability from the interval between 1 and `maxTopLevelStatementsCount`.

The variable declaration statements are not counted in this count.

maxStmtsWithinABlock The parameter specifies the maximum number of statements in a nested block. The actual number of statements to generate is taken with a uniform probability from the interval between 1 and `maxStmtsWithinABlock`.

errorGuaranteed The parameter is a `Boolean`. A random statement that produces an error is plugged at a random program location if the value is true.

generateDivisions The parameter is a `Boolean`. If the value is `false`, no divisions will be generated in the program, thus significantly reducing the probability of generating an error. However, it is still possible that the program contains an access out-of-bounds error or a null pointer dereference.

precomputeVariableCosts The command `precomputeVariableCosts` is used to precompute query count estimation. We can output the time that the analyses took to a file.

program The parameter is the input file with the program to be analyzed.

merging-strategy The parameter specifies what analyses will be used. The possible values are `lattice-based` and `recursive`. There is no default value. The implementation of the precomputations is discussed in 5.4.3.2

timeout . The parameter is of type `Integer`. If the time spent by the analyses exceeds the timeout, the analyses stop.

kappa The parameter is of type `Integer`, and it is used when merging `recursive` is enabled. The default value is `1`. The parameter is further discussed in 5.4.3.

output The parameter specifies an output file. The standard input is the default. The time spent doing the analyses is stored in the file.

export The `export` command takes a program and exports its AST as JSON. The functionality was provided for the NI-APR course.

- **program** The input file name
- **output** The output file name
- **indent** Number of spaces in the output.

run The `run` command executes the program with an AST interpreter using concrete execution. However, arrays are not supported.

- **program** The name of a file containing the source program.
- **input-file** The optional name of a file containing inputs to the file read by the `input` expressions.
- **input** If `input-file` is missing, the input data can be passed through the command line with the `input` parameter. If both parameters `input-file` and `input` are missing, the standard input is used instead.
- **args** Arguments for the main function.
- **time** If this flag is enabled, the elapsed program execution time is outputted.
- **ascii** If this flag is enabled, the outputted numbers are converted to ascii.

cfg The `cfg` command takes a program and returns it as a CFG. The CFG is outputted to the standard output.

- **program** The input file name
- **norm** The `norm` parameter is a flag. The AST is normalized if the flag is present.

5.1.1 Workflow of the executor

The parser outputs the program in the AST format. The parsed code proceeds to a normalizer, which transforms it so that it can be analyzed more easily. The normalizer takes an AST representation of a program and returns it normalized.

The normalized code is being transformed to CFG using the `CfgFactory` class from the `cfg` package.

The symbolic executor uses the CFG as input. The code of symbolic executor is present in the package `symbolic_execution`. It is further discussed in the section 5.3.

The execution is stopped after the first error is found, after all paths are explored, or after the time spent doing the analyses exceeds the timeout.

5.2 Program representation

The program is represented as a set of CFGs. A separate CFG is constructed for each function. Each node in the control flow graph is of type `CfgNode`. The node contains an ID number, one statement, and its predecessors and successors in the graph. Contrary to the CFG shown in chapter 3, every node in the CFG can not contain more than one statement.

The implementation supports all statements presented in chapter 2. All statements are implemented as classes inheriting from the trait `Stmt`. The statements contain expressions.

The implementation supports all expressions described in chapter 2. All binary operators are represented by the `BinaryOp` class. All other expressions have their own class that represents them. All expressions are implemented as classes inheriting from the trait `Expr`. The traits `Expr` and `Stmt` both inherit from the trait `Ast`.

5.2.1 AST normalization

The program AST is normalized so that complicated expressions are split into multiple assignments. Any expression containing a subexpression can be normalized if the subexpression contains another subexpression. A result of such a subexpression is assigned to a new variable. The name of such a temporary variable is such that it can not come from the source code. The new variable is then used instead of the subexpression. This is done recursively. In the final normalized program, all subexpressions do not contain any inner subexpression.

The left sides of the assign statements are not normalized.

Code snippet 5.1 shows an example of a normalization of a statement. The statements on the right side of the example are the normalized statements.

Code listing 5.1 An example of a cfg normalization

```
*x[0].field = *y + z[i + 1];    ->    _t0 = *y;
                                      _t1 = i + 1;
                                      _t2 = z[_t1];
                                      *x[0].field = _t0 + _t2
```

5.2.2 Supported values

In the symbolic store, there are several types of values, we can use.

UninitializedRef `UninitializedRef`

Stands for a defined variable that has not yet been assigned to.

Nullref `NullRef`

Stands for a null pointer.

PointerVal `PointerVal(address: Int)`

Stands for a non-null pointer. It takes a number as an argument, which points to a memory location.

ArrVal `ArrVal(elems: Array[PointerVal])`

Stands for an array value. The individual elements are pointers that point to a memory location, where the actual value of the element is stored.

RecVal `RecVal(fields: Map[String, PointerVal])`

Stands for a record value. The individual fields are named pointers that point to a memory location where the actual value of the field is stored.

SymbolicVal `SymbolicVal()`

Stands for a value that was acquired from an environment whose value is unclear. However, it must be a number.

ITEVal `IteVal(trueState: PointerVal, falseState: PointerVal, expr: Expr)`

Stands for an if-then-else expression variable. It can hold different values depending on the result of a condition. It takes three arguments. One of them is the expression that determines what value the expression evaluates. The other two arguments are called `trueVal` and `falseVal`. If the expression evaluates to `true`, the ITEVal evaluates to a value pointed to by the `trueVal`. Otherwise, it evaluates to a value pointed by `falseVal`.

All these values are implemented as classes inheriting from trait `Val`.

5.3 Symbolic executor

```
class SymbolicExecutor(
    program: ProgramCfg,
    subsumption: Option[PathSubsumption],
    ctx: Context,
    searchStrategy: SearchStrategy,
    executionTree: Option[ExecutionTree],
    covered: Option[mutable.HashSet[CfgNode]],
    createITEAtSymbolicArrayAccess: Boolean,
    printStats: Boolean
)
```

The class that performs the actual symbolic execution is called `SymbolicExecutor`. It has the following parameters.

program This parameter `program` is the program to be symbolically executed.

subsumption The parameter `subsumption` is further explained in 5.4.2.

ctx The parameter `ctx` is an instance of z3 `Context` class that is supplied for the constraint solver discussed in 5.3.2.1.

searchStrategy The parameter `searchStrategy`, is described in 5.3.2.2.

executionTree The parameter `ExecutionTree` is described in 5.3.2.2.

covered The parameter `covered` is described in 5.3.2.2.

createITEAtSymbolicArrayAccess The parameter `createITEAtSymbolicArrayAccess` is discussed in 5.3.3.2.

printStats The parameter `printState` controls whether the executor outputs the current number of explored paths to the standard output.

The most crucial method of the `SymbolicExecutor` class is `run`. It does not take any arguments. It finds the entrypoint of the program supplied to the class by the constructor. We use this entrypoint as the program location of the initial symbolic state.

Since the language does not support booleans, the initial path condition is represented by number one instead of `true`.

The initial state is added to the empty worklist of states, a field of the `SymbolicExecutor` class. The basic workflow of the executor is such that a path is taken from the worklist in each iteration and executed. The execution of a path starts with calling the `step` method. The `run` method stops executing when there is no other state to take from the worklist or when an error in the program is found.

Before explaining the `step` method, we will explain the implementation of symbolic states.

5.3.1 Symbolic state

```
SymbolicState(
    programLocation: CfgNode,
    pathCondition: Expr,
    symbolicStore: SymbolicStore,
    callStack: List[CfgNode]
)
```

The symbolic state is implemented in class `SymbolicState`. Parameters of the class are a program location of type `CFGNode`, the path condition of type `Expr`, the symbolic store of type `SymbolicStore`, and call stack which is a list of elements of type `CfgNode`.

The call stack is needed to evaluate function calls and will be later discussed in subsection 5.3.3.3.

The `SymbolicStore` contains member `Storage`, and member `frames`.

The storage is a growable array whose values have type `Val`. It is used to store the values of the variables. If a value is a `PointerVal`, the address of the pointer value is used as the index at which the value that is pointed to lays.

The frames are a list of hash maps. The keys in the maps are variable names, and the values are of type `PointerVal` that are used as pointers to the storage. Each element in the list stores variables of one function invocation on the call stack.

Assume the following code snippet 5.2.

Code listing 5.2 An example of a simple program for the visualization of the symbolic store.

```
foo() {
  var x,y,z;
  x = 2;
  x = 4;
  y = &x;
  z = &x;
  return -1;
}


main() {
  var y;
  y = 1;
  return foo();
15 }
```

Assume that the program already called the `foo` function, and the current program location is line 7. The path condition is `true`. The following example shows 5.1 what the symbolic store looks like.

frame1

Variable	Pointer
y	ptr(0x00)

frame2

Variable	Pointer
x	ptr(0x02)
y	ptr(0x03)
z	ptr(0x03)

storage

Address	Value
0x00	num(1)
0x01	num(2)
0x02	num(4)
0x03	ptr(0x02)

Figure 5.1 An example of a symbolic state

There are two frames. The one called `frame1` contains the variables defined in the `main` function. The one called `frame2` contains the variables defined in the `foo` function. The pointers stored in both frames point to the same storage.

The class offers many methods to modify the symbolic store or retrieve some information from it:

addToPathCondition The method is used to add another expression to the path condition. It takes an expression of type `Expr` as the parameter. It updates the path condition of the state to be a conjunction of the old path condition and the expression.

The class also offers several methods to allow modifying the symbolic store.

addVar The method takes a value of type `IdentifierDecl` and registers the variable in the symbolic store. The initial value of a registered variable is `UndefinedVal`.

updateVar The method takes a variable name and a value of type `Val` as parameters and updates the value of the variable name in the symbolic store with the provided value.

updateMemoryLocation The method works similarly to `updateVar`, but it takes a pointer of type `PointerVal` as a parameter instead of the name.

addVal This method takes a value of type `Val` as a parameter and adds it to the symbolic store. It returns the `PointerVal` pointing to the address to which the value was added.

The class supports several methods to modify the current program location.

step The method changes the program location to the successor of the current program location. The method should be called only if the current statement has only one successor.

getIfTrueState This method is expected to be used only if the current program location is a control node. In microc the control nodes are the `if` and `while` statements. The method returns a symbolic state for the path that executes the branch that gets executed if the guard is evaluated to `true`. The method changes the program location accordingly and updates the path condition with the guard.

getIfFalseState This method is expected to be used only if the current program location is a control node. The method returns a symbolic state for the path that executes the branch that gets executed if the guard is evaluated to `false`. The method changes the program location accordingly and updates the path condition with the negation of the guard.

There are several methods for retrieving values from the symbolic state.

getValOnMemoryLocation The method takes a pointer value of type `PointerVal` and returns a value present on that address from the symbolic store.

getValueOfVar The method takes a variable name as a parameter and returns the value associated with this variable in the symbolic store.

associatedPathsCount Each symbolic state is associated with one execution path, so the method returns the number one.

The class also supports several methods related to path explosion reduction techniques.

mergeStates The method takes another symbolic state of type symbolic state as a parameter. The result of the function is a merged symbolic state of type `MergedSymbolicState`. The algorithm for merging symbolic states is further discussed in section 5.4.3.

5.3.2 Evaluation of statements

The `step` method takes a symbolic state as an argument. It looks at the current program location of this state and decides what to do based on the type of the statement.

The `assign` statement evaluates the right part of the statement and updates a memory cell defined by the left side of the statement. The method `getTargetMemoryCell` is used to determine the memory location of the left side of the statement.

The `output` statement does not update the symbolic state. However, an error might be present in the expression, so we have to evaluate it anyway.

When an `if` statement is encountered, the current path splits in two. The feasibility of the `then` branch is checked by calling the method `solveCondition` of the constraint solver that is discussed in 5.3.2.1. The method checks whether a constraint made of the current path condition and the guard condition is satisfiable. If so, we get a new state by calling the `getIfTrueState` of the symbolic state and continue exploring the `then` branch.

When the exploration of this path finishes or if the `then` path is unfeasible, we move to the `else` branch. The feasibility of the `else` branch is checked by calling `solveCondition` for the path condition and the negation of the guard. The method `solveCondition` calls the constraint solver. If the method returns that the condition is satisfiable, we call the `getIfFalseState` of the symbolic state to get a new state that we add to the worklist and backtrack back to the `run` function.

The treatment of `while` statements is similar. The difference is that the state that executes the body of the loop is added to the worklist, while the state created for the path that leaves the loop gets executed immediately. If the path executing the body was explored right away, it could lead to the executor being stuck exploring a body of one loop. This functionality is implemented in the method `stepOnLoop`.

5.3.2.1 Constraint solving

The implementation utilizes the z3 solver [13] developed by Microsoft to solve the satisfiability of the constraints. Class `ConstraintSolver` provides an interface between the z3 solver and the present symbolic execution engine.

The class provides the following methods:

createConstraint The method gets an expression of type `Expr` and a symbolic state of type `SymbolicState` as parameters and converts it to a z3 representation of the expression. The symbolic state is used to get the values of the variables present in the expression.

solveConstraint The method takes a z3 constraint as a parameter, checks its satisfiability using the z3 API, and returns the satisfaction status.

solveCondition The method takes a guard of type `Expr`, a path condition of type `Expr`, and a symbolic state of type `SymbolicState`. The guard and path condition are combined into a conjunction, and a z3 constraint is created with the function `createConstraint`. The function returns the result of the method `solveConstraint` with the constraint as the argument.

5.3.2.2 Search strategies

The strategies for picking a state from the worklist are implemented as classes that inherit from the `SearchStrategy` trait. The trait offers the following methods:

addState Takes a symbolic state of type `SymbolicState` as a parameter. The state is added to the worklist. Each strategy implements the worklist differently.

getState It takes no parameter and retrieves a state from the worklist.

statesCount Returns the size of the worklist as an integer.

updateExecutionTree Takes two symbolic states of type `SymbolicState` as parameters. The only strategy that implements this method is `Tree`.

The implementation supports the following search strategies.

BFS The oldest state in the worklist gets picked. Implemented with a queue.

DFS The newest state in the worklist gets picked. Implemented with a stack.

Radnom A random state is picked from the worklist. Implemented with a stack and an array with the same elements. A random valid index of the array is generated, and the element on the index is removed from both the array and the set.

Tree This strategy uses a symbolic execution tree implemented in the `ExecutionTree` class discussed in chapter 3.

It implements the `updateExecutionTree` method so that an association between a parent state and a child state is added to the `ExecutionTree`.

The state is picked by traversing the tree randomly from the root to a leaf and returning the leaf.

Coverage When this search strategy is enabled, the field `covered` of the `SymbolicExecutor` class is used to collect statements that were already encountered in the execution of some path. For each state in the worklist, the strategy computes a cost as an inverse of the distance to the nearest statement that the symbolic executor has not encountered yet. A random state is picked so that the states with lower costs have a bigger chance of being picked.

Assume that we computed the following costs 5.1 for the states.

Table 5.1 Costs and probabilities of being picked in the coverage strategy

name of the state	cost	probability of being picked
a1	1/10	1/13
a2	1/5	2/13
a3	1	10/13

The state `a1` have cost `1/10`, the state `a2` have cost `2/10`, and the state `a3` have the cost `1`. We sum all the costs. The probability of being picked for every state is the cost of the state divided by the sum of all costs.

Klee The strategy was inspired by a default Klee [6] strategy and combines the `Tree` and the `Coverage` strategies. In every even invocation of the `getState` method, the `Tree` search strategy is used. Every odd invocation of the `getState` method of the `coverage` search strategy is used.

The strategies can also be combined with state merging, as will be further discussed in the subsection 5.4.3.

5.3.3 Evaluation of expressions

Statements usually consist of expressions. The method `evaluate` evaluates an expression. The main parameters of the method are an expression and a symbolic state. A boolean parameter `ignoreUncertainErrors` disables asking the constraint solver to check a potential error. By default, the parameter is set to false, and the use cases when it is set to true are discussed in 5.4. The return value is a symbolic value.

The expression of types `Number`, `Array`, `Record`, and `Null` are dealt with by simply returning these expressions as values. The expression `Input` returns a new symbolic value.

If the expression is `Identifier`, we return the value of the variable from the symbolic state.

The expressions such as `Deref`, `BinaryOp`, `FieldAccess`, or `Not` contain inner expressions. The evaluation of the inner expression is used to compute the result of the outer expression.

The evaluation of the expressions `ArrayAccess` and `CallFuncExpr` is more complicated, and it is further discussed in 5.3.3.2 and 5.3.3.3, respectively.

5.3.3.1 Error detection

Some expressions can cause an error. In some cases, we can prove the absence of an error by a simple number comparison. For example, in the case of division by a number, we check whether the number is zero.

For other cases, the constraint solver is needed to prove the absence of an error. We check for this possibility if the parameter `ignoreUncertainErrors` is set to false. The constraint solver is invoked when we evaluate a division when the divisor is a symbolic value. We check if the divisor may be zero. Similarly, the constraint solver is invoked when an element of an array is accessed with a symbolic index. We check whether access out of the bounds of the array is possible.

5.3.3.2 Evaluation of array indexing

The `ArrayAccess` expression consists of an `array` subexpression and `index` subexpression.

```
ArrayAccess(array: Expr, index: Expr)
```

We evaluate the array subexpression and expect the result to be an array. We also evaluate the index subexpression. If it is a number, we use it to access the element at the index of the array.

If the value is symbolic, we must call the constraint solver to check whether an out-of-bounds array access is possible. We check this possibility with the constraint solver. If the error is impossible, we iterate through array indices and check with the constraint solver whether it is possible that the element at the index was accessed. The parameter `createNewStateAtSymbolicArrayAccess` of the `SymbolicExecutor` class decides the strategy for continuing the analysis for all the possible indexes.

If createNewStateAtSymbolicArrayAccess is false, the elements at indices of the array that could be accessed are combined into an ITE expression. The ITE expression is returned as the result of the evaluation.

However, the ITE expression could slow down the executor. Thus, another strategy is used if createNewStateAtSymbolicArrayAccess is true. Notice that since the AST is normalized, the only expression that can evaluate to a symbolic value is Identifier.

Recall that all expressions that contain an expression within themselves are normalized so that the inner expression does not contain any inner expression themselves. As the following example shows, if the expression gets normalized, the final index is an always Identifier expression.

```
a = arr[k + 2];                ->                  _t1 = k + 2;
                                                   a = arr[_t1];
```

There are three expressions that can be evaluated into a number and do not contain any expression within itself. The first is Number, which can not be evaluated into a symbolic value. The second is Input, which would always cause an out-of-bounds error. The last possible expression is Identifier.

Thus, if in the normalized AST, an index is a symbolic value, it comes from an Identifier expression. For each element that can be accessed, we duplicate the current symbolic state multiple times. In each state, we update the value of the index variable with one of the possible indices.

Finally, we add all these states to the worklist and backtrack back to the run function, where we pick another state from the worklist that gets explored.

Code snippet 5.3 is an example of a program having a symbolic value as an index.

Code listing 5.3 An example of a symbolic value being an array index

```
1  main() {
2    var arr, i, res;
3    arr = [0, 1, 2];
4    i = input;
5    res = 0;
6    if (i < 3 && i >= 0) {
7       arr[i] = -1;
8       res = arr[i];
9    }
10   return res;
11 }
```

In lines 7 and 8, an array element is accessed, but the index is symbolic. Thus, we do not know what element of the array was accessed. If the value of the parameter createNewStateAtSymbolicArrayAccess is false, all possible memory locations would be merged into an ITE val.

If createNewStateAtSymbolicArrayAccess is false, then a new state is created at line 7 for each possible value of the index i. The value of i is set in each state to be the value of the index.

That means that in our example, three states would be created. They would all be located on line 8. The first would set the variable i to zero. The second would set the variable i to one. The third would set the variable i to two.

Then, all these states are added to the worklist, and the current path is stopped.

5.3.3.3 Evaluation of function calls

The symbolic execution of function calls is implemented in the method `runFunction`, and the following text discusses it in detail. First, recall that thanks to the code normalization discussed in the subsection 5.2.1, every function call is normalized, so it is always the single expression on the right side of an `assign` statement.

To execute a function call, we have to symbolically execute the called function, return to the current function, and continue executing the code from the successor statement of the statement with the function call.

Before the function is called, a new frame must be pushed to the symbolic state, and the evaluated arguments of the function must be copied to the new frame. Also, the current program location is saved. Then, the CFG associated with the called function is found, and the program location of the symbolic state is changed to the entrypoint node of the function CFG.

When we return from the function, the frame is popped from the symbolic state, and the current program location in the state is set to the successor of the saved program location.

Notice that sometimes, when a state is picked from the worklist in the `run` method, it has a program location located in a different function than `main`. This is a problem when we want to return from the function since the call of the `runFunction` method that called the current function is not on the system stack anymore. Thus, we do not know who the callee of the current function was and what was the program location of the function call. For this reason, every state has to remember all the program locations of the chain of function calls that led to the current location of the state. In the implementation, these program locations are stored in the field `callStack` in the `SymbolicState` class.

The following example 5.4 shows the problem.

Code listing 5.4 An example of a program using factorial

```
fac(n) {
    var f;

    if (n == 0) {
      f = 1;
    } else {
      f = n * fac(n - 1);
    }

    return f;
}


main() {
  var a,b;
  b = input;
  a = fac(b);
  output(a);
  return 1 / (a - 2);
20 }
```

The program computes an inversion of a factorial number depending on an input value subtracted by 2. Thus, if the factorial is 2, the program should result in a division by zero error.

The initial state reaches line 17 and calls the `runFunction` method of the symbolic executor to execute the `fac` function. Line 4 is the only place in the program where we split the execution into multiple states. In the first visit to this statement, we add the `else` state, which executes line 7, to the worklist, and we finish the execution of the `then` state, which executes line 5. The function ends its execution and returns the value 1 at line 10. After backtracking to the `runFunction` function, we continue the execution of `main` function on lines 18 and 19 and do not find any error.

In the symbolic executor, we backtrack to the `run` function and pick the only present state from the worklist. Notice that no `runFunction` call is now present on the system stack. The state starts its execution on line 7 and calls the `fac` function again. On line 4, the state is split again. We add the `else` state to the worklist and continue executing the `then` state on line 5. On line 10, we return from the function, but we backtrack all the way to the `run` function since there is no `runFunciton` on the stack.

Therefore, we have to remember the program locations of the function calls. If we did so, we can now pick a location from the top of this call stack. Recall that since the CFG is normalized, all the calls to a function are always the only expression on the right side of an assigned statement. Thus, we update the variable on the left side of this statement with the function call result and call the step function on the successor of this statement in the CFG. In our example, we update the variable `a` and move to line 18.

5.3.4 Statistics

The `Statistics` class contains the information about how many paths were already explored. When a state finishes its execution, the `associatedPathsCount` method of the `SymbolicState` is called, and the number of explored paths is increased by this number.

5.4 Path explosion optimizations

This section discusses the implementation of the techniques for tackling path explosion discussed in chapter 4. This section is structured similarly to that chapter, discussing each technique in a subsection.

The subsection 5.4.1 discusses the implementation of path running, the subsection 5.4.3 discusses the implementation of the state merging, the subsection 5.4.4 discusses the implementation of the loop summarization, and the 5.4.2 discusses the implementation of the path subsumption.

In the following text, we will use the concept of a general symbolic state. A general symbolic state is a state in which all values are symbolic. It is created by copying all values from a standard symbolic state to the general state but turning them into symbolic ones. We can do this because all variables are guaranteed to have the same type during the execution of the program.

5.4.1 Path pruning

The path pruning is implemented as an internal part of the symbolic executor. When an `if` statement or a `while` statement is encountered, and the path condition is updated, the satisfiability is checked. If it is not satisfiable, the path is pruned, and another state gets to be executed. Thus, the implementation is present in the `SymbolicExecutor` class itself.

5.4.2 Path subsumption

The path subsumption technique is implemented in the `PathSubsumption` class. The constructor of the class takes a constraint solver of type `ConstraintSolver` as a parameter. Path subsumption in the symbolic executor is enabled by passing a `PathSubsumption` class as an argument to the `SymbolicExecutor`. The class contains a map called `annotations`, whose keys are `CFGNode`, and values `Expr`. In this map, we store annotations that may potentially prune the current path for every program point.

In the implementation, the maximal number of iterations of the body of the loop for the computation of the annotations before the execution leaves the loop is set to one. If the annotations collected by running one iteration of the body of the loop are not sufficient, the standard symbolic execution is run.

The implementation provides the following methods.

addAnnotation `addAnnotation(node: CfgNode, expr: Expr): Unit`

The method takes a node of type `CFGNode`, and an expression of type `Expr` as parameters.

It puts the node as the key and the expression as the value to the `annotations` map. If an expression is already present, a disjunction of the original expression and the new expression is created.

addAnnotations `addAnnotations(nodes: List[CfgNode], expr: Expr): Unit`

The method takes a list nodes of type `CFGNode`, and an expression of type `Expr` as parameters. . It calls `addAnnotation` for each node.

computeAnnotationFromSuccessors
`computeAnnotationFromSuccessors(node: CfgNode): Unit`

The method takes a node in the CFG and computes the annotation for the node from the successors of the node in the CFG. The computed annotation is added to the `annotations` map. The detailed rules for computing annotations based on annotations of the successor nodes are described in the 4.2 section.

checkSubsumption `checkSubsumption(state: SymbolicState): Boolean`

The method takes a symbolic state of type `SymbolicState`, and returns a boolean that says whether the path condition subsumes the annotation stored for the program location of the symbolic state.

Recall that subsumption is a relationship between formulas `A` and `B`, where every interpretation that makes `A` true makes `B` true too. The method combines the path condition and the negation of the annotation into a conjunction. The subsumption relationship is present if this constraint is unsatisfiable.

performInduction

```
performInduction(nodes: List[CfgNode],
                 identifier: Identifier,
                 symbolicState: SymbolicState,
                 executor: SymbolicExecutor,
                 loop: CfgNode): Unit
```

The method performs the induction discussed in the section 4.2. For annotations of every loop, we plug the number zero instead of the `identifier`. Recall that the annotations of one node are stored as a disjunction. We split the disjunction into individual elements.

For example, by splitting the following disjunction of three elements, we get those three elements.

```
a1 or a2 or a3              ->              (a1, a2, a3)
```

We check whether each element is inductive by using a general symbolic state. For each path in the loop, we apply the effect of the path to the symbolic state. We also have one symbolic state that is not updated by any path.

We use the constraint solver to check whether there is some interpretation of the formula so that the element was satisfiable before the path was executed, but its negation became satisfiable after the path was executed.

Thus, we check whether an element is satisfiable using the non-updated general state, but its negation is satisfiable with one of the updated general states.

While executing the paths to create the updated general states, a path that has not yet been explored can contain an error. In that case, we stop the symbolic execution with an error found. This way, we can also detect unreachable errors. During the execution of these paths, we set the `ignoreUncertainError` parameter of method `evaluate` to ensure that only errors independent of the current symbolic state are detected. Otherwise, errors dependent on a path condition could be detected.

Code snippet 5.5 shows it in an example.

Code listing 5.5 An unreachable error when collecting paths

```

main() {
  var a,b,c,i,n;
  n = input;
  a = input;
  b = 0
  c = [1, 2, 3];
  i = 0;
  if (a != 0) {
    a = 1;
  }

13   while (i < n) {
14     b = b / a;
15     if (i == -1) {
16         b = c[3];
17     }
18     i = i + 1;
19   }
20   return b;
21 }
```

On line 13, we collected an annotation after running the body of the loop once, and we wanted to make sure that the annotation was inductive. Thus, we execute all paths in the loop to create the updated general states.

There are statements, such as the one on line 14, that can produce an error based on the current symbolic state. When updating the general symbolic states, we have to ignore these errors. We do it by passing `ignoreUncertainError=true` to the method `evaluate`.

During the execution of the path containing line 16, an error is detected, even though it is unreachable. The symbolic execution stops there.

5.4.2.1 Computation of annotations

The annotations are checked before evaluating a statement by calling the `checkSubsumption` method. If the method returns `true`, the path is pruned. Otherwise, we continue with standard symbolic execution.

The algorithm for computing annotations works as described in the section 4.2. When the `ReturnStmt` statement is encountered, the annotation `true` is added by calling the `addAnnotation` method.

While backtracking from the invocations of the `step` method, the method `computeAnnotationFromSuccessors` is invoked to compute the annotations for the non-return statements.

Recall that in the section 4.2, the technique was presented using programs with explicit error locations. However, microc contains expressions that produce an error only for some values of the variables.

The solution for this is not to propagate annotations from the successor in the CFG if the current statement can contain an error. Code snippet 5.6 shows an example of this.

Code listing 5.6 An example of a possible error that removes annotations

```

main() {
  var x, y, i;
  x = input;             // false
  i = input;             // false
  y = x;                 // false
  if (i == 0) {          // false
     i = 1;              // false
  }
10    i = 1 / i;              // false
11    if (x < y) {            // x >= y
12        x = 1 / 0;
13    }
14    return 0;               // true
15 }
```

The statement on line 10 can possibly cause an error if `i` is zero. Thus, the implementation does not propagate any annotations from the successors of the statement on line 10 to its predecessors, so we do not prune any path that can reach line 10 in the future.

A better strategy would be to either transform the program to have only explicit error locations or to try to modify the annotations so that only those paths that definitely can not produce an error on line 10 are pruned. The annotations could look like this in code snippet 5.7

Code listing 5.7 An example of a possible error that modifies annotations

```
1
2  main() {
3    var x, y, i;
4    x = input;               // true
5    i = input;               // true
6    y = x;                   // true
7    if (i == 0) {            // x >= y
8        i = 1;               // x >= y
9    }
10   i = 1 / i;               // x >= y and i != 0
11   if (x < y) {             // x >= y
12       x = 1 / 0;
13   }
14   return 0;                // true
15 }
```

The treatment of loops works as described in the section 4.2. A variable `_t` is introduced, and annotations preventing it from being smaller than zero are added for each node in the loop. The CFG of the loop is temporarily modified by plugging a statement at the end of the loop that decrements the variable `_t`. In the implementation, we collect the annotations by running only one iteration of the body of the loop. After collecting the annotations, we keep only the inductive ones by calling the `performInduction` method.

5.4.3 State merging

The implementation supports enabling state merging.

A symbolic state created by merging two original symbolic states is implemented in class `MergedSymbolicState`, which inherits from `SymbolicState`. The class takes all parameters that class `SymbolicState` takes and initializes the parent class with them. The last parameter of `MergedSymbolicState` is a pair of two symbolic states of type `SymbolicState` called `innerStates`. These are the states out of which the merged state is formed. Class `MergedSymbolicState` overrides the methods `getIfTrueState`, and getIfFalseState. The implementation is the same as in the parent class, but a MergedSymbolicState instance is returned. Also, the method associatedPathsCount that returns the number of paths associated with a state is overridden and returns a sum of associatedPathsCount of the subStates.

Both to-be-merged states must have the same call stacks and program locations, and the merged state will reuse both. The path condition of the merged state is a disjunction of the path conditions from the to-be-merged states.

5.4.3.1 The algorithm for merging two states

The following algorithm describes how the symbolic stores are merged. It will first be explained theoretically and then explained with an example.

We create a new symbolic store. We go through all frames and each variable in the frames of the original stores. We add a frame to the merged store for each frame in the original store. We compare the values of the variables in both symbolic stores. If the values are the same, we add the variable with its value to the new store. Otherwise, we must create an ITEVal.

Some values contain a pointer to another value in the original symbolic state. We must also copy the values that are pointed to and change the values of the pointer to point to the right location in the merged symbolic state. We also have to ensure that if two pointers point to the same address in the original symbolic state, they also point to the same address in the merged store.

The algorithm 1 moveValues is used to move the values of a variable from the source states to the merged state. The function is called for each variable in each frame. The algorithm also uses the algorithm 2 inside.

The parameter resStore is the symbolic store where we put the merged values. The parameters sourceState1 and sourceState2 are the symbolic stores we are currently merging.

The parameter ptr1 is a pointer to the value that we want to merge in the sourceStore1. Similarly, ptr2 is a pointer to the value we want to merge in the sourceStore2.

In the first call of the function for each variable, we pass the pointer to the value of the variable in the sourceStore1 as ptr1, and the value in the sourceStore2 as ptr2.

The pointerMapping1 parameter is a map where addresses from sourceState1 are mapped to addresses from resStore. Each time we move a value to the resStore, we update this map with the address of the value in the sourceState1 being mapped to the address of the value in the resStore.

The parameter pointerMapping2 works similarly, but it maps addresses from sourceStore2 to addresses in resStore.

If the value that we want to move to the new state is a pointer, we must follow the chain of pointers in the symbolic store until a non-pointer is reached. Since all variables are the same type at any time, the only situation when the pointer chains have different sizes is if one state has the variable still uninitialized. If the values at the end of the chain of pointers are the same, we add the chain of pointers and the value to the store once. Otherwise, we have to add both chains and create an ITEVal that encapsulates both values at the end of the chains.

Figure 5.3 shows three storages, and figure 5.2 shows three frames. We will show

Algorithm 1 Move Values

```
function MoveValues(resStore, sourceStore1, sourceStore2, ptr1, ptr2, pointerMapping1, pointerMapping2)    ▷ Returns a pair of pointers to new values in resStore
    if $pointerMapping1.contains(ptr1)$ & & $pointerMapping2.contains(ptr2)$ then
        return $(pointerMapping1.get(ptr1), pointerMapping2.get(ptr2))$
    end if
    if $pointerMapping1.contains(ptr1)$ then
        return $(pointerMapping1.get(ptr1), MoveValue(..., ptr2, ...))$
    end if
    if $pointerMapping2.contains(ptr2)$ then
        return $(MoveValue(..., ptr1, ...), pointerMapping2.get(ptr2))$
    end if
    $val1 \leftarrow sourceStore1.getVal(ptr1)$
    $val2 \leftarrow sourceStore2.getVal(ptr2)$
    if $val1$ and $val2$ are both pointers, arrays, records or ite then
        $(ptr1, ptr2) \leftarrow MoveValues(..., val1, val2, ...)$
        if $ptr1 == ptr2$ then
                $r \leftarrow resStore.addNewVal(ptr1)$
                $pointerMapping1.put(ptr1, r)$
                $pointerMapping2.put(ptr2, r)$
                return $(r, r)$
        else if else then
                $r1 \leftarrow resStore.addNewVal(ptr1)$
                $r2 \leftarrow resStore.addNewVal(ptr2)$
                $pointerMapping1.put(ptr1, r1)$
                $pointerMapping2.put(ptr2, r2)$
                return $(r1, r2)$
        end if
    else if $val1$ and $val2$ are both arrays, records or ite then
        Handle similarly to pointers
    else if values are identical then
        $r \leftarrow resStore.addNewVal(val1)$
        $pointerMapping1.put(ptr1, r)$
        $pointerMapping2.put(ptr2, r)$
        return $(r, r)$
    else
        $r1 \leftarrow MoveValue(..., ptr1, ...)$
        $r2 \leftarrow MoveValue(..., ptr2, ...)$
        $ite \leftarrow createITE(r1, r2)$
        return $(ite, ite)$
    end if
end function
```

Algorithm 2 Move Value

```
function MoveValue($resultStore, sourceStore, ptr, pointerMapping$)
    if $pointerMapping.contains(ptr.address)$ then
        return $PointerVal(pointerMapping(ptr))$
    end if
    Match ($sourceStore.getVal(ptr)$):
      if the value is pointer, array, record or ite:
        Recursive call to $MoveValue$ on inner pointers.
        Use the returned pointers to create new value in the $resultStore$.
      else:
        Add the val to the $resultStore$.
    $pointerMapping.put$(ptr, address of the new value)
    return address of the new value
end function
```

how to merge two symbolic stores called `store1` and `store2` into a `merged store`. The `storage1` is a storage associated to `store1`, and `frame1` is a frame associated with `store1`. The same is true for `store2`, `storage2` and `frame2`. The `merged frame` and `merged storage` belong to `merged store`.

The value of type `num` stands for numbers, and the numbers of type `ptr` point to a memory cell in the store.

frame1

Variable	Pointer
x	ptr(0x01)
y	ptr(0x03)

frame2

Variable	Pointer
x	ptr(0x02)
y	ptr(0x04)

merged frame

Variable	Pointer
x	ptr(0x01)
y	ptr(0x06)

Figure 5.2 An example of a merged frame

We create a new symbolic store to move the merged variables there. Since the symbolic stores that will be merged have only one frame, we create one frame in the merged store.

First, we move the values of the variable `x`. We call the function `moveValues` with the values being the pointers stored in the frames.

```
moveValues(merged store, store1, store2, ptr(0x01), ptr(0x02),
pointerMapping1, pointerMapping2)
```

In `storage1` the value at address `0x01` is `ptr(0x00)`, and in the `storage2` the value at address `0x02` is `ptr(0x00)`. Thus, we call the `moveValues` function for these values.

```
moveValues(merged store, store1, store2, ptr(0x00), ptr(0x00),
pointerMapping1, pointerMapping2)
```

storage1

Address	Value
0x00	num(1)
0x01	ptr(0x00)
0x02	num(2)
0x03	ptr(0x01)

storage2

Address	Value
0x00	num(1)
0x01	num(3)
0x02	ptr(0x00)
0x03	ptr(0x01)
0x04	ptr(0x03)

merged storage

Address	Value
0x00	num(1)
0x01	ptr(0x00)
0x02	num(3)
0x03	ptr(0x02)
0x04	ptr(0x01)
0x05	ptr(0x03)
0x06	ite(cond, ptr(0x04), ptr(0x05))

Figure 5.3 An example of a merged storage

The values at address `0x0` are `num(1)` in both source stores. Thus, we add `num(1)` to the `merged storage` at the address `0x00`. The `moveValues` returns `(ptr(0x0), ptr(0x0))` to the previous call `moveValues`.

Since the returned values are the same, the value `ptr(0x00)` is added to the `merged store`, and the value `ptr(0x01)` is returned from the function `moveValues`. The variable `x` is added to the `merged frame` with the value `ptr(0x01)`.

The value `x` is successfully merged and moved to the merged state. Now, we proceed to move the variable `y`. First, we call the `moveValues` with the following arguments.

```
moveValues(merged store, store1, store2, ptr(0x03), ptr(0x04),
pointerMapping1, pointerMapping2)
```

The values on the addresses of the pointers are pointers themselves, so we call `moveValues` with the following arguments.

```
moveValues(merged store, store1, store2, ptr(0x01), ptr(0x03),
pointerMapping1, pointerMapping2)
```

The address `0x01` in the `store1` is already present in `pointerMapping1`. It is mapped to the address `0x00` in the `merged store`. Thus, this value is reused, and the function `moveValue` is called for the other pointer.

```
moveValue(merged store, store2, ptr(0x03), pointerMapping2)
```

In this function, `moveValue` is called again.

```
moveValue(merged store, store2, ptr(0x01), pointerMapping2)
```

The function adds the value `num(3)` to the `merged storage` and returns the address `0x02`. The first call of `moveValue` adds `ptr(0x02)`, and returns its address.

We return to the function `moveValues`, and since the returned pointers are different, we add them both to the `merged store`. The function returns the addresses of these newly added values. Since the returned values differ, we construct an `ITEVal` from them. This value is added to the `merged frame` for the variable `y`.

The implementation supports several state merging strategies. They all share the same algorithm for merging two states described above, but they differ when it is decided that two states should be merged.

All state merging strategies implement the trait `StateMerging` that extends the `SearchStrategy` trait described in the subsection 5.3.2.2.

The `AgressiveStateMerging` strategy merges the states whenever it can. Recall that the states are mergeable if they share the same program location and call stack. The `HeuristicBasedStateMerging` strategy merges the states only if the future number of constraint solver invocations is expected to be low.

5.4.3.2 Heuristic-based state merging

The heuristic-based state merging algorithm merges states only if the additional pressure on the constraint solver is expected to be low. This is computed by estimating how many times ITE values would be part of the constraints in the future constraint solver invocations. For each program point and each variable, an estimation of future constraint solver calls containing the variable is computed.

We later use this information during symbolic execution to decide whether to merge the two states. For each variable whose values differ in the two symbolic stores, we sum the precomputed values for the variables at the current program point. We compare the sum with the `limit-cost` parameter of the technique. If the sum is lower, we merge the states.

The thesis implements two different algorithms for the precomputation. Since the final experiments were performed on single-function programs, both algorithms are implemented as intraprocedural. Notice that handling guards of unbounded loops can be challenging. The constraint solver is called an unknown number of times, so it is impossible to precompute these numbers precisely. The solution to this problem is a little bit different in both approaches.

recursion-based-approach The recursive-based algorithm was already discussed in 4.3.1.

The solution of the algorithm for the problem of unbounded loops is to introduce a parameter κ that specifies a maximal number of iterations of every loop.

lattice-based-approach A lattice is a data structure that allows the merging of information from different paths in a program. For any two elements in the lattice, there is a least upper bound representing the smallest element that is greater than or equal to both. Similarly, there is the greatest lower bound for any two elements. It represses the largest element, which is less than or equal to both.

The dataflow lattice analysis aims at gathering information about the behavior of the program at every program point. It propagates values through the `CFG` and merges them using lattices until a fixpoint is reached, meaning that running the analysis further does not change the result.

We also differentiate between forward analyses and backward analyses. In the forward analyses, the information of the node is computed from its predecessor, while in the backward analyses, the information is computed from the successors.

In the implementation of the symbolic executor, the lattice-based query count estimation is implemented using backward analyses. The solution of the analyses for the problem of unbounded loops is to introduce a limit for the computed value that can not be exceeded. We can do it because we are interested only in smaller computed values of future constraint solver invocations, and we do not mind that the computation of big values will be imprecise. This allows us to reach a fixpoint.

In the implementation, the maximal computable value for a variable and a program location is 10. We can set the value this low since the experiments performed in 6 showed that heuristic-based state merging performs best with small values of the parameter `limit-cost`.

5.4.4 Loop summarization

The loop summarization technique is implemented in classes `LoopSummarization`, `PDA`, and `Trace`. Class `LoopSummarization` inherits from the `SymbolicExecutor` class.

The method `stepOnLoop` is overridden so that the loop is tried to be summarized. If it is impossible to summarize the loop, the `stepOnLoop` method of the `SymbolicExecutor` class is called.

The loops are summarized lazily when they are first encountered by the symbolic executor. If it is possible to summarize the loop, we create the PDA. Otherwise, we remember that the loop is unsummarizable, and the next time we encounter the loop, we do not try to summarize it.

We analyze the loop and find all inner paths. For this, we need to create general symbolic states. For each path, we capture the effect of its execution by updating the general symbolic state. If there is an unsummarizable statement, the loop is registered as unsummarizable.

We must also ensure all conditions within the loop are IVs, as discussed in 4.4.2.2. A condition is updated predictably if all variables inside it are updated predictably. Thus, we register the loop as unsummarizable if there is one variable in a condition that is not updated by the same value in an iteration of a path belonging to the loop. This is also true for the case when the variable is updated in a nested loop.

5.4.4.1 Path dependency automaton

The collected paths are the vertices for the `PDA` graph.

The function `computePathRelationship` is used to detect whether there is a directed edge from a source vertex to a target vertex.

A vertex is in the `Init` set if the condition of the path associated with it is satisfiable. Thus, all paths with satisfiable path conditions are in the `Init` set.

The vertices with no successors in the graph are put in the `Accept` set.

The PDA is checked to see whether it contains connected cycles. If there are some, the loop is registered as unsummarizable.

5.4.4.2 Getting summary from the PDA

The PDA is traversed from the vertices in the `Init` set through the computed edges until a vertex in the `Accept` set is reached. All such paths within the graph are collected into a trace. A trace consists of a trace condition and of the changes to the variables that the trace summarizes. During each transition through an edge, the edge condition is added to the trace condition, and the changes associated with the edge are added to the changes associated with the trace.

A cycle in the graph is detected by reaching a vertex for the second time within one trace. We try to compute the periods for every vertex in the cycle. A period is a known number of iterations of a path in the cycle before the next path in the cycle gets executed. The z3 solver supports returning a value for a variable and a constraint that satisfies the constraint when the variable holds this value.

If multiple numbers could be a period for one of the paths in the cycle, then we cannot produce a summary. To check whether a period `p` provided by a constraint solver is the only possible solution, we repeat the constraint solver query, but we add an expression `p != solution for the number of iterations` to the query. Thus, if the solver returns that the constraint is satisfiable, there are multiple possible values for the period. Otherwise, there is only one.

5.4.4.3 Summarization of nested loops

An inner loop can be encountered when summarizing an outer loop. To summarize the outer loop, the inner loop must be summarizable. If the summarization of the inner loop finishes well, we split the paths collected for the summarization of the outer loop several times so that there is a new path for every combination of the outer loop path and inner loop trace. The path condition of every such path is a conjunction of the condition of the path from the outer loop and the condition from the trace. The changes of variables stored for the new path are those stored for the outer loop path, updated by the changes associated with the trace. Then, we continue to summarize the next statement after the inner loop.

Chapter 6

Experiments

This chapter discusses the experiments to evaluate the techniques that were implemented. Section 6.1 discusses the inner workings of the microc program generator that had to be developed because of the lack of real-world microc programs. Section 6.2 discusses the metrics used during the experiments. Section 6.3 discusses the settings and results of the experiments. Finally, section 6.4 discusses the limitations of the experiments.

6.1 Microc code generation

The class `ProgramGenerator` generates random microc programs. The generator guarantees that all variables are initiated before use, but other errors can be randomly generated. There can be divisions by a value that can be zero or accesses out of bounds for an array. There is only one function generated for each program. There can also be guaranteed errors when `errorGuaranteed` is set to `true`.

The generator supports the parameters described in chapter 5. Both statements and expressions are randomly generated. The probabilities of generating a particular statement or expression follow a uniform distribution. However, the probabilities of generating a general while loop or a for loop can be specified by a parameter. Recall that a for loop in microc is a while loop with a comparison of two variables in the guard and an incrementation of one of the variables at the end of the loop.

The generator generates only array access expressions indexed by a number to reduce the number of access out-of-bounds errors. Early experiments with the program generator showed that when other expressions, such as an identifier, can be used as array indices, the errors are very common.

It also remembers the initial size of each variable of the type array. It only generates array accesses, so the index is within the initial size. However, if a smaller array is assigned to the variable, an array out-of-bounds error can happen when an array element is accessed. Code snippet 6.1 shows this situation.

Code listing 6.1 An example of a random generated program with an array access out-of-bounds error

```
main() {
  var var1;
  var1 = [0, 1, 2, 3, 4, 5] // The initial size is 6
  // So the maximal index of array access
  // for var1 that we generate is 5
  ...
  var1 = [0, 1, 2, 3];
  var1[5]; //error
  return 0;
}
```

The array `var1` is initialized with 6 elements, so every time we want to retrieve an element from `var1`, we use an index between zero and five. An error is possible after an array with a smaller size is assigned to `var1`.

6.2 Metrics

The techniques will be compared based on several metrics:

Path coverage: we count the number of paths in the program that were explored. We do not count the paths during which an error was found. Thus, if an error is found in the first path that is executed, the outputted coverage by the symbolic executor is zero.

Time to finish: we measure the time from when the symbolic executor starts to when it stops. The symbolic executor has a timeout that defaults to 30 seconds.

Error found: we measure the number of errors detected. The symbolic executor is designed to stop after finding one error, so the number is either zero or one.

6.3 Experiments

The experiments were split into three parts. The first part served to find the best parameters for the parametrizable algorithms. The experiment is discussed in 6.3.1 subsection.

The implemented symbolic executor is parametrizable in the following ways:

There are six search strategies. Each one can be combined with one of three merge strategies, or no merge strategy can be used. The path subsumption and loop summarization can be enabled or disabled. That is 96 different configurations in total. Thus, in part two of the experiment 6.3.2, a small experiment is performed using all configurations to pick a smaller subset on which a bigger experiment is going to be performed.

In path three of the experiments 6.3.3, we evaluate how the techniques perform on programs generated under different settings of the program generator.

6.3.1 Parameters for merge strategies

An experiment was performed to evaluate different settings for `lattice-based` and `recursive` state merging strategies. We were interested in the total achieved path coverage in a set of programs. Both strategies have the parameter `limit-cost`, which defines the maximal amount of future constraint solver invocations on constraints with ITE values that we consider to be bearable enough to still merge the states.

The strategy `recursive` also has the parameter `kappa` discussed in 4.3.1. That specifies the number of times a body of a loop is executed for all loops in the program.

We generated three different sets of programs based on their expected size. Table 6.1 shows the values of the parameters of the program generator used for the generation.

Table 6.1 Definitions of program sets for the first experiment

program set	maxBlockDepth	maxTopLvlStmtsCount	maxStmtsWithinABlock
small programs	3	10	5
medium programs	4	10	10
large programs	5	20	10

The other parameters shared the same values for every set. The parameter `loopGenProb` had the value `0.15`. The parameter `forLoopGenProb` had the value `0.10`. The parameter `errorGuaranteed` was `false`. The parameter `generateDivisions` was also `false`.

First, we did an experiment using just the `precomputeVariableCosts` command of the implementation to compare how fast the algorithms for precomputation of the variable costs discussed in 5.4.3.2 are. We set the timeout for 30 seconds. In 6.2, we may see the number of programs for each set whose precomputation exceeded the timeout.

Table 6.2 Number of timeouts for precomputation of query count analyses

merge strategy	recursive	lattice-based
small programs	0	0
medium programs	10	5
large programs	39	33

We can see that while the precomputation for small programs does not usually last very long and is never timeouted, the large programs generally cause a timeout. We can see that there were more timeouts using the strategy `recursive`. In table 6.3, we can see the total time spent doing the precomputation.

Table 6.3 Total time (ms) spend precomputing variable costs for different program sets

merge strategy	recursive	lattice-based
small programs	7109	12765
medium programs	454663	377573
large programs	1210021	1121559

The trend is similar to when we looked at the results for timeouts. We can see that the lengths of computations rise rapidly as the programs get bigger. We also see that the strategy `recursive` took more time than the strategy `lattice-based`.

We can look at how good the path coverage is when we use different values of the parameter `limit-cost`. First, we executed the programs with merging strategy `lattice-based` and the search strategy `BFS`. The timeout was 30 seconds. Recall that the precomputation of the query count estimation is not considered within the timeout. We tried several different values for the parameter `limit-cost`. Table 6.4 shows the sum of achieved path coverage values for different values of `limit-cost`.

Table 6.4 Path coverage for different values of the parameter `limit-cost`

Limit cost	**Path coverage**
1	14562
2	15135
3	16200
5	17442
10	15165

We can see that while there is no big difference between the results, the best results were achieved when `limit cost` was five. Parameters 3 and 5 seem to be a little bit more favored than the others.

We repeated the same for the strategy `recursive`. Recall that the strategy also uses a parameter `kappa`. The results can be seen in table 6.5.

Table 6.5 Path coverage for different values of the parameter limit

Limit cost/κ	1	2	3	5	10
1	16514	17170	13545	17662	18562
2	18078	14470	15228	19518	20819
3	18272	14651	12685	16718	17460
5	17492	14788	13393	17705	19960
10	17300	13270	15267	18176	18149

We can again see that the differences between individual results are rather small. The best results we achieved for a combination of `limit-cost` being 2 and `kappa` being 10.

Lastly, another experiment comparing different values of `limit-cost` was performed on the `large` program set. Since the precomputation times for the `recursive` strategy were rather large, the experiment was performed only for the `lattice-based` strategy. Table 6.6 shows the results.

Table 6.6 Path coverage for different values of the parameter limit

Limit cost	**Path coverage**
1	24136
5	20576
10	20363
20	20101

In the experiment, the value `limit-cost` archives the best results when it is 1. The differences between other values are very small. Thus, it seems that very low values for `limit-cost` are preferred for larger programs.

6.3.2 Comparing search and merge strategies

An experiment was performed to compare different search strategies and state-merging strategies. The test was split into two parts. In the first part, smaller programs were generated with the following settings.

- `number of test programs`: 50
- `forLoopGenerationProbability`: 0.15
- `generalWhileLoopGenerationProbability`: 0.10
- `maxBlockDepth`: 3
- `maxTopLevelStatementsCount`: 10
- `maxNumberOfStatementsWithinABlock`: 5
- `guaranteedError`: false
- `generateDivisions`: true

The programs were then run with a timeout of 30 seconds, `kappa` of `10` and `limit-cost` of `2`.

In table 6.7, we can see the computed path coverage for different types of search strategies and merge strategies.

Table 6.7 Path coverage for different search and merge strategies on smaller programs

search strategy/merge strategy	none	recursive	lattice-based	aggressive
dfs	35463	26495	22923	26114
bfs	34809	25134	22499	25906
random	34187	25914	22761	24877
tree	34860	25391	22416	19328
coverage	29553	25053	24277	23167
klee	30945	24558	23491	23943

We can see that the merge strategy performing the best is `none`. The other merge strategies afford roughly the same performance. The likely reason is that the costs to merge the states were higher than the optimization gained by exploring multiple parts simultaneously. The merging was especially slow for states that were already merged from other states and contained a lot of ITE expressions.

The results for search strategies are very similar, and there is no clear winner or loser.

In table 6.8, we can see how many errors were detected during the experiment.

We can see that no error was detected at all. This probably means that the testing programs had almost no reachable error. This might be possible because of our settings where `errorGuaranteed=false` and `generateDivisions=true` combined with smaller programs being generated.

Table 6.8 Errors detected for different search and merge strategies for smaller programs

search strategy/merge strategy	none	recursive	lattice-based	aggressive
dfs	0	0	0	0
bfs	0	0	0	0
random	0	0	0	0
tree	0	0	0	0
coverage	0	0	0	0
klee	0	0	0	0

Table 6.9 Total time in ms for different search and merge strategies for smaller programs

search/merge strategy	none	recursive	lattice-based	aggressive
dfs	901 880 ms	908 698 ms	902 027 ms	901 872 ms
bfs	925 980 ms	901 641 ms	901 806 ms	904 023 ms
random	901 721 ms	901 954 ms	902 166 ms	901 939 ms
tree	901 939 ms	901 380 ms	901 694 ms	901 953 ms
coverage	902 209 ms	901 658 ms	901 544 ms	902 193 ms
klee	901 924 ms	902 081 ms	901 926 ms	902 308 ms

Table 6.9 shows the total time for the analyses of all programs for every combination of a search strategy and merge strategy.

We can see that execution times are similar. In table 6.10, we can see how many runs were stopped because the timeout was exceeded. Recall that the timeout is 30 seconds.

Table 6.10 Runs timeouted for different search and merge strategies for smaller programs

search strategy/merge strategy	none	recursive	lattice-based	aggressive
dfs	30	30	30	30
bfs	30	30	30	30
random	30	30	30	30
tree	30	30	30	30
coverage	30	30	30	30
klee	30	30	30	30

For all the different combinations, 20 programs finished before the timeout of 30 seconds and had to be stopped. Since the programs are small, this probably means there are 20 programs in the dataset with a finite number of reachable paths and 30 programs with an infinite number of reachable paths.

In the second part of the experiment, larger programs were generated.

- `number of test programs`: 20
- `forLoopGenerationProbability`: 0.15
- `generalWhileLoopGenerationProbability`: 0.10
- `maxBlockDepth`: 5

- `maxTopLevelStatementsCount`: 20
- `maxNumberOfStatementsWithinABlock`: 10
- `guaranteedError`: false
- `generateDivisions`: true

The programs were then run with a timeout of 30 seconds, `kappa` of 10, and `limit-cost` of 2.

The merge strategy `recursive` was not included in this test because of the experiment and the results of the first part of the experiment, where the strategy `recursive` and `lattice-based` achieved similar performance.

Table 6.11 shows the total path coverage achieved on the dataset for different search and merge strategy combinations.

Table 6.11 Path coverage for different search and merge strategies on larger programs

search strategy/merge strategy	none	lattice-based	aggressive
dfs	42342	25439	22246
bfs	42245	24987	22078
random	42789	25185	23270
tree	44839	24840	18697
coverage	44981	26891	24489
klee	45272	26456	25275

Similarly to the experiment with smaller programs 6.7, the `none` merging strategy is the best performing. The difference is even bigger than for smaller programs. This can be explained by the fact that more merging can be present in larger programs than in smaller programs. However, contrary to the test with smaller programs, the `lattice-based` strategy achieves visibly better performance than the `aggressive` strategy. This may be because larger programs contained situations when merging was very disadvantageous, and heuristic state merging was able to spot that.

Table 6.12 shows how many errors were detected for each combination of the merge strategy and search strategy.

Table 6.12 Errors detected for different search and merge strategies on larger programs

search strategy/merge strategy	none	lattice-based	aggressive
dfs	6	6	6
bfs	6	6	6
random	6	6	4
tree	7	6	5
coverage	5	6	4
klee	7	6	6

We can see that the numbers range from 4 to 7, and the higher numbers are generally achieved by combinations that also achieve larger path coverage. An outlier is the combination of search strategy `coverage` and merge strategy `none`. It detected fewer errors than the combination of search strategy `coverage` and merge strategy `lattice-based`, which achieved smaller path coverage.

Table 6.13 shows the total time to analyze the dataset for each combination of merge and search strategies.

Table 6.13 Total time in ms for different search and merge strategies for larger programs

search strategy/merge strategy	none	lattice-based	aggressive
dfs	1 175 992 ms	1 286 597 ms	1 163 204 ms
bfs	1 176 129 ms	1 221 521 ms	1 162 780 ms
random	1 167 890 ms	1 195 775 ms	1 218 451 ms
tree	1 154 474 ms	1 225 384 ms	1 183 792 ms
coverage	1 189 595 ms	1 167 928 ms	1 207 693 ms
klee	1 152 564 ms	1 168 480 ms	1 164 299 ms

The numbers are quite similar, but the combinations that found more errors spent less time, which is expected.

Table 6.14 shows for how many programs the execution timeouted.

Table 6.14 Runs timeouted for different search and merge strategies for larger programs

search strategy/merge strategy	none	lattice-based	aggressive
dfs	38	38	38
bfs	38	38	38
random	38	38	40
tree	37	38	39
coverage	39	38	40
klee	37	38	38

The numbers are interesting when we sum them with the number of detected errors for each combination. This can be seen in table 6.15.

Table 6.15 (Runs timeouted + Error detected) for different search and merge strategies for larger programs

search strategy/merge strategy	none	lattice-based	aggressive
dfs	44	44	44
bfs	44	44	44
tree	44	44	44
coverage	44	44	44
klee	44	44	44

The sum is the same for each combination of strategies. Thus, there are probably 6 correct programs with a finite number of reachable paths. For the other programs, we either detected an error or the computation was timeouted.

6.3.3 Final experiments

The last experiment was performed to compare how the techniques deal with different types of programs. The program sets are listed in table 6.16.

Table 6.16 Definitions of program sets

program set	forProb	whileProb	maxBlockDepth	errorGuarateed	generateDivs
15-10-3-f-f	0.15	0.1	3	false	false
15-10-5-f-f	0.15	0.1	5	false	false
50-10-3-f-f	0.50	0.1	3	false	false
50-10-5-f-f	0.50	0.1	5	false	false
15-50-3-f-f	0.15	0.5	3	false	false
15-50-5-f-f	0.15	0.5	5	false	false
15-10-3-t-t	0.15	0.1	3	true	true
15-10-5-t-t	0.15	0.1	5	true	true
50-10-3-t-t	0.50	0.1	3	true	true
50-10-5-t-t	0.50	0.1	5	true	true
15-50-3-t-t	0.15	0.5	3	true	true
15-50-5-t-t	0.15	0.5	5	true	true
15-10-3-f-t	0.15	0.1	3	false	true
15-10-5-f-t	0.15	0.1	5	false	true
50-10-3-f-t	0.50	0.1	3	false	true
50-10-5-f-t	0.50	0.1	5	false	true
15-50-3-f-t	0.15	0.5	3	false	true
15-50-5-f-t	0.15	0.5	5	false	true

The first six program sets were generated with `errorGuaranteed=false` and `generateDivisions=false`. Thus, these programs have a small probability of containing an error. Contrary to that, the next six program sets are guaranteed to have an error because `errorGuaranteed=true`. The error can be, however, generated unreachable, so it can be impossible to find it. The final six program sets were generated with `generateDivisions=true`, making the probability of the program containing an error significantly higher than for the first six program sets.

There are also program sets containing a lot of while loops like `15-50-3-t-t` or `15-50-5-f-f`, program sets containing a lot of for loops like `50-10-3-t-t`. The sets also differ in the parameter `maxBlockDepth`, which controls how deep the nested statements can be.

We executed the program using the techniques implemented for tackling path explosion. Some parameters of the symbolic executor were the same for each technique. The timeout was 30 seconds, and the `search-strategy` was `Tree`. Before the experiment, a small optimization was added to the state merging code, annotating the merged states that took too long to merge. These states will never be merged again. Thus, the state merging in this experiment was slightly faster than in the previous ones. The experiment was also performed with only the basic symbolic executor without any additional techniques. Table 6.17 shows the results.

We can see that no errors were found in the first six sets. In the next six sets, errors

Table 6.17 Final test with no path explosion techniques enabled

settings	path coverage	time (ms)	timeouted	errors detected
15-10-3-f-f	37806	901 826 ms	30	0
15-10-5-f-f	40581	872 229 ms	29	0
50-10-3-f-f	47607	1 141 582 ms	38	0
50-10-5-f-f	50149	1 141 485 ms	38	0
15-50-3-f-f	28069	724 232 ms	24	0
15-50-5-f-f	28011	874 873 ms	29	0
15-10-3-t-t	3041	62 081 ms	2	48
15-10-5-t-t	7868	151 910 ms	5	45
50-10-3-t-t	5657	152 193 ms	5	45
50-10-5-t-t	3419	62 452 ms	2	48
15-50-3-t-t	825	46 485 ms	1	49
15-50-5-t-t	4492	123 366 ms	4	46
15-10-3-f-t	28667	784 102 ms	26	11
15-10-5-f-t	33562	933 317 ms	31	1
50-10-3-f-t	43290	1 085 621 ms	36	6
50-10-5-f-t	43131	966 059 ms	32	9
15-50-3-f-t	19869	741 374 ms	24	7
15-50-5-f-t	28916	994 137 ms	33	0

were found in almost every program. Notice that all programs in these sets were stopped because of the timeout or an error was found. In the last six sets, the errors were found in several programs.

Another observation we can make is that programs generated with a big probability of general while loops have a lot of timeouts. This can probably be attributed to many loops whose guard is always true being generated, and thus, the execution gets stuck in an unbounded loop forever.

More errors were found in the programs with `maxBlockDepth=3`. Since the same later happens with other techniques, it is probably just a coincidence that more errors were generated with `maxBlockDepth=3` than with `maxBlockDepth=5`.

Table 6.18 shows the results when state merging `lattice-based` was enabled. The `limit-cost` was set to one.

We can see that the results when state merging is enabled are very similar to the results with no state merging. As the previous experiment indicated, the path coverage is lower than if state merging is not used. An interesting result can be seen for the program set `15-50-3-t-t`. This was the only time any technique found bugs in all programs in a set. However, it is unsurprising because the basic symbolic executor with no state merging found errors in 49 out of 50 sets.

Table 6.19 shows the results when subsumption is enabled. The state merging is disabled in this test.

The subsumption ended with significantly fewer timeouts and shorter execution times than the previous methods. The fewer timeouts are likely a result of subsumption being able to prove some programs correct, and the shorter execution times are a consequence of

Table 6.18 Final test with state merging enabled

settings	path coverage	time (ms)	timeouted	errors found
15-10-3-f-f	33542	901 403 ms	30	0
15-10-5-f-f	35889	871 625 ms	29	0
50-10-3-f-f	36455	1 141 582 ms	38	0
50-10-5-f-f	40295	1 141 203 ms	38	0
15-50-3-f-f	24357	723 835 ms	24	0
15-50-5-f-f	22908	874 392 ms	29	0
15-10-3-t-t	2414	62 152 ms	2	48
15-10-5-t-t	7342	151 616 ms	5	45
50-10-3-t-t	5696	151 771 ms	5	45
50-10-5-t-t	2844	61 835 ms	2	48
15-50-3-t-t	505	35 563 ms	0	50
15-50-5-t-t	4100	122 836 ms	4	46
15-10-3-f-t	24676	783 858 ms	26	11
15-10-5-f-t	28680	932 748 ms	31	2
50-10-3-f-t	35522	1 085 621 ms	36	6
50-10-5-f-t	36124	964 590 ms	32	9
15-50-3-f-t	16948	731 073 ms	24	7
15-50-5-f-t	27241	993 267 ms	33	0

that. This is further supported by subsumption having essentially the same performance as the previous techniques for program sets guaranteed to have an error.

The subsumption performed very well for programs in the first six sets. This makes sense because these programs contain a small number of expressions that can generate errors because there are no divisions. Contrary to the basic symbolic execution and state merging, errors were found already in these first sets. Interestingly, four errors were found in the set `15-50-5-f-f` where previously no error was found.

However, there are other test sets where subsumption was less successful in finding errors compared to the basic symbolic executor or state merging, such as `15-50-3-t-t`. The reason why the technique is sometimes more successful and sometimes less successful than the other methods might be that, given how it is implemented, it usually explores paths in a different order than if subsumption is disabled. When subsumption is disabled and a loop is encountered, we first explore the path that leaves the loop and add the path that stays in the loop to the worklist. Contrary to that, subsumption explores the path that executes the body of the loop right away to collect the annotations.

When the summarization results of the first six sets and the last six sets are compared, we can see that the last six sets have slightly more timeouts, even though there are also more detected errors. This is probably caused by the generated divisions that are making the annotations much less efficient.

Thus, whether the subsumption is enabled might significantly affect which parts of the program are explored faster. However, the biggest advantage that subsumption brings is analyzing programs with a small number of statements that can possibly cause an error quickly.

Table 6.19 Final test with subsumption enabled

settings	path coverage	time (ms)	timeouted	errors detected
15-10-3-f-f	15272	55 836 ms5	18	0
15-10-5-f-f	19099	603 792 ms	19	2
50-10-3-f-f	24668	947 529 ms	31	0
50-10-5-f-f	23842	929 905 ms	30	1
15-50-3-f-f	12478	455 024 ms	13	0
15-50-5-f-f	4322	429 401 ms	11	4
15-10-3-t-t	817	64 676 ms	2	47
15-10-5-t-t	6968	152 370 ms	5	45
50-10-3-t-t	1554	44 621 ms	1	45
50-10-5-t-t	2104	67 741 ms	2	48
15-50-3-t-t	1875	109 909 ms	3	47
15-50-5-t-t	2188	162 591 ms	5	44
15-10-3-f-t	17037	587 409 ms	18	11
15-10-5-f-t	19369	715 169 ms	22	2
50-10-3-f-t	26851	785 561 ms	25	6
50-10-5-f-t	29095	918 933 ms	29	7
15-50-3-f-t	9566	422 012 ms	12	7
15-50-5-f-t	8098	670 180 ms	19	3

Table 6.20 shows the results for loop summarization.

The number of errors detected using summarization is similar to previous techniques. However, we can see quite low values for the column `time`. When the summarization results of the first six sets and the last six sets are compared, we can see that there are more timeouts in the last six sets, even though there are also many more detected errors. This is probably because the generated divisions make more loops unsummarizable.

Similarly to the previous techniques, there were significantly fewer timeouts for programs with for loops than for programs with general while loops.

6.4 Discussion

Experiments showed how efficient different search strategies, state merging strategies, and other techniques were when analyzing randomly generated microc programs. While the results for search strategies were very similar, state merging was significantly slower than if no state merging was enabled. Profiling indicated that merging states that were already merged from several states takes a lot of time. This may be one of the reasons why state merging harmed efficiency.

The experiments also showed that techniques like subsumption and summarization can often prove the correctness of a program significantly faster than standard symbolic execution. However, the number of detected errors compared to state merging and the basic test with no additional techniques was similar.

All the experiments had a timeout of 30 seconds. We can see that the time needed for the execution in seconds was often very close to $timeouts * 30$, meaning that the

Table 6.20 Final test with summarization enabled

settings	path coverage	time (ms)	timeouted	errors detected
15-10-3-f-f	3099	251 633 ms	8	0
15-10-5-f-f	9422	498 189 ms	16	1
50-10-3-f-f	9468	640 904 ms	21	0
50-10-5-f-f	8529	502 850 ms	16	0
15-50-3-f-f	2507	203 684 ms	6	0
15-50-5-f-f	5362	305 749 ms	9	0
15-10-3-t-t	456	35 600 ms	1	49
15-10-5-t-t	212	33 980 ms	1	45
50-10-3-t-t	760	36 435 ms	1	45
50-10-5-t-t	1895	65 962 ms	2	48
15-50-3-t-t	1810	106 600 ms	3	47
15-50-5-t-t	2121	104 310 ms	3	44
15-10-3-f-t	6503	434 515 ms	14	10
15-10-5-f-t	7960	497 141 ms	16	3
50-10-3-f-t	10525	584 797 ms	19	5
50-10-5-f-t	4786	293 972 ms	9	7
15-50-3-f-t	5393	318 674 ms	10	5
15-50-5-f-t	4689	329 352 ms	10	0

programs that were proved correct or had an error found finished very quickly. Hence, there would probably not be a bigger number of detected errors or programs proved correct if the timeout was increased.

All the tested techniques achieved quite similar performance for both `maxBlockDepth=3` and `maxBlockDepth=5`. However, this may be a consequence of randomly generated larger programs having more bugs in them. Thus, it was not proven or disproven that deeper loops with more blocks in them are harder to analyze,

There are several limitations to the presented experiments. The generated programs did not resemble programs written by humans. To reduce the final number of errors, the programs consisted of fewer expressions that could contain an error.

Moreover, the programs probably shared many of the same features because the probabilities of generating statements other than loops and the probabilities of generating different expressions were the same for all programs.

The number of timeouted runs was similar for programs with a lesser amount of loops and for programs with a lot of for loops. The reason why programs with a lot of general while loops had significantly more timeouts are loop bodies that are impossible to leave. This makes the general while loops quite unrepresentative of the human-written programs.

Chapter 7

Conclusion

The thesis explains the symbolic execution technique with examples and provides an overview of the microc language, and describes its features.

The thesis presents several techniques for tackling path explosion in symbolic execution, which are important for analyzing programs containing a large number of nested loops.

A symbolic executor supporting the presented techniques and a generator of random programs based on several parameters were developed. Both implementations are provided in the thesis.

The experiments were performed using different types of generated programs and different settings of the symbolic executor. Path subsumption and loop summarization techniques were both able to prove some programs correct quickly, thus providing better overall time than the standard symbolic execution. However, they found a similar amount of bugs as standard symbolic execution. State merging performed similarly to the standard symbolic executor, while also being slower.

All the techniques but also the basic symbolic executor performed reasonably well on programs with more loops or with deeper more nested loops.

7.1 Future Work

Extending the algorithms Subsumption can be improved to support collecting annotations from more than one run of the body of the loop. The mechanism for treating statements that contain a possible error may be greatly improved to compute stronger annotations, as discussed in 5.4.2.

The dynamic state merging discussed in 4.3.2 can be implemented.

Comparison of the interactions between the techniques In the provided implementation, we implemented state merging, subsumption, and summarization. However, only one of the techniques can be used at once. If enabling multiple techniques at once would be supported, we could do an experiment to determine how well the techniques work together.

Extending the program generator The program generator can be improved. The solution of the current implementation for generating large programs that are not full of errors is to greatly limit the number of generated expressions that can contain an error. This may be a great advantage to subsumption in particular because then it may prove very fast that there are no potential errors in the program.

Thus, it would be good to develop a program generator that is able to return larger programs with a small number of errors that contain many expressions that can generate an error.

Bibliography

1. MØLLER, Anders; SCHWARTZBACH, Michael I. *Static Program Analysis*. Department of Computer Science, Aarhus University, 2020. Available also from: `https://cs.au.dk/~amoeller/spa/`.

2. KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2nd ed. Prentice Hall, 1988.

3. SCOWEN, Roger S. Extended BNF — A generic base standard. In: [online]. 1998. Available also from: `https://www.semanticscholar.org/paper/Extended-BNF-%E2%80%94-A-generic-base-standard-Scowen/ed89e6f749768cc4fc585e6ef406afeace436a19?p2df`.

4. TRTÍK, Marek. *Symbolic Execution and Program Loops*. 2013. Available also from: `https://theses.cz/id/p8lj1h/`. PhD thesis. Masaryk University, Faculty of Informatics Brno. SUPERVISOR: prof. RNDr. Antonín Kučera, Ph.D.

5. BALDONI, Roberto; COPPA, Emilio; D'ELIA, Daniele Cono; DEMETRESCU, Camil; FINOCCHI, Irene. A Survey of Symbolic Execution Techniques. *CoRR*. 2016, vol. abs/1610.00502. Available also from: `http://dblp.uni-trier.de/db/journals/corr/corr1610.html#BaldoniCDDF16`.

6. CADAR, Cristian; DUNBAR, Daniel; ENGLER, Dawson R. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI'08)*. San Diego, California: USENIX Association, 2008, pp. 209–224.

7. AVGERINOS, Thanassis; CHA, Sang Kil; HAO, Brent Lim Tze; BRUMLEY, David. AEG: Automatic Exploit Generation. In: *Network and Distributed System Security Symposium*. 2011.

8. GODEFROID, Patrice; KLARLUND, Nils; SEN, Koushik. DART: directed automated random testing. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. PLDI '05. ISBN 1595930566. Available from DOI: `10.1145/1065010.1065036`.

9. MCMILLAN, Kenneth L. Lazy Annotation for Program Testing and Verification. In: TOUILI, Tayssir; COOK, Byron; JACKSON, Paul (eds.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 104–118. ISBN 978-3-642-14295-6.

10. KUZNETSOV, Volodymyr; KINDER, Johannes; BUCUR, Stefan; CANDEA, George. Efficient State Merging in Symbolic Execution. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, China: Association for Computing Machinery, 2012, pp. 193–204. PLDI '12. ISBN 9781450312059. Available from DOI: `10.1145/2254064.2254088`.

11. XIE, Xiaofei; CHEN, Bihuan; LIU, Yang; LE, Wei; LI, Xiaohong. Proteus: computing disjunctive loop summary via path dependency analysis. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016. Available also from: `https://api.semanticscholar.org/CorpusID:18032345`.

12. LIGHTBEND, INC. *Scala 2*. 2024. Available also from: `https://www.scala-lang.org/`. Programming language.

13. MOURA, Leonardo de; BJØRNER, Nikolaj. Z3: An Efficient SMT Solver. In: RAMAKRISHNAN, C. R.; REHOF, Jakob (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN 978-3-540-78800-3.