HEURISTICS FOR CONCOLIC SOFTWARE TESTING

by

Yavuz Köroğlu

B.S., Computer Engineering, Boğaziçi University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2016

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Assoc. Prof. Alper Şen for guiding me with his experience and skill, and fellow jury members Prof. Can Özturan and Asst. Prof. Barış Tankut Aktemur for their invaluable feedback.

I would like to thank especially Tuğba Bostan, for her unlimited support and effort day and night. I would not complete this thesis without her.

I would like to thank my parents Aykut and Fatma Köroğlu, and my brother Oğuz Köroğlu for their support. I also thank my fellow lab partners, Hasan Ferit Enişer, Mehmet Köse, Görker Alp Malazgirt, Nour Alhilal, Salih Bayar and Gökçehan Kara for discussing new ideas and how to represent them.

The author was supported by TUBITAK MS Fellowship BIDEB-2210-A.

ABSTRACT

HEURISTICS FOR CONCOLIC SOFTWARE TESTING

Software testing is an essential part of the software development process. Concolic testing is an automated unit test generation technique which is a result of decades of study on making the automated testing scalable. However, bottlenecks such as constraint solving still prevents concolic testers to be used in large projects. The constraint solving bottleneck occurs due to the large number of branches on the execution paths of a Unit Under Test (UUT).

In this thesis, we design a novel constraint solving strategy called Incremental Partial Path Constraints (IPPC) on top of a standard concolic tester. Our strategy makes more but smaller queries to the constraint solver, i.e. ignores some path conditions. We implement IPPC on top of a known concolic testing framework, CREST. We show that it is possible to reach the same branch coverage as the standard concolic tester while decreasing the burden on the constraint solver. We support our claims by testing several C programs using different strategies. Experimental results show that our modification improves runtime performance of the standard concolic tester in half of the experiments and results in more than 5x speedup when the UUT has many infeasible paths. Ultimately, IPPC eliminates the need for solving large constraints while automatically generating unit tests.

ÖZET

KONKOLİK YAZILIM TESTİ İÇİN SEZGİSEL YÖNTEMLER

Yazılım testi, yazılım geliştirme sürecinin ayrılmaz bir parçasıdır. Konkolik test üzerinde sistematik testin daha büyük yazılımlara uygulanabilmesi amacıyla on yıllarca süren çalışmaların sonucu geliştirilmiş bir birim test yaratma tekniğidir. Ancak kısıt çözümü gibi darboğazlar konkolik testçilerin büyük projelerde kullanılmasına engel teşkil etmektedir. Kısıt çözümündeki darboğaz Test Altındaki Birim'in icra yollarındaki yüksek sayıda dallanmadan ileri gelmektedir.

Bu tezde, Artımlı Kısmi Yol Kısıtları adını verdiğimiz ve standard konkolik testçi üzerine geliştirdiğimiz özgün bir kısıt çözüm stratejisi önermekteyiz. Önerdiğimiz değişiklik bazı yol koşullarını gözardı ederek kısıt çözücüye çok sayıda ama küçük sorgular göndermektedir. IPPC algoritmasını iyi bilinen bir konkolik test sistemi olan CREST üzerinde geliştirdik. IPPC üzerine olan çalışmamızda kısıt çözücü üzerindeki yükü azaltırken aynı dallanma kapsamasına ulaşılabileceğini göstermekteyiz. İddialarımızı farklı stratejileri çeşitli C programları üzerinde test ederek desteklemekteyiz. Deneysel sonuçlar yaptığımız değişikliğin standard konkolik testçinin işleyiş süre performansını deneylerin yarısında artırdığımızı ve eğer test altındaki birim fazla imkansız yol içeriyorsa 5 kattan daha fazla hızlanma sağladığını göstermektedir. Yaptığımız çalışma otomatik birim test yaratımı için büyük kısıtların çözülmesine olan ihtiyacı ortadan kaldırdığımızı göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES ix
LIST OF SYMBOLS x
LIST OF ACRONYMS/ABBREVIATIONS xi
1. INTRODUCTION 1
1.1. Motivation 1
1.2. Contributions 4
1.3. Outline 5
2. BACKGROUND 6
2.1. Automated Unit Test Generation 6
2.2. Constraint-Based Testing 8
2.2.1. Symbolic Execution 10
2.2.1.1. Example 13
2.2.2. Concolic Testing 16
2.2.2.1. Example 18
3. RELATED WORK 21
3.1. Concolic Testing 21
3.2. Constraint Solving 23
3.3. Incremental Constraint Solving 24
4. INCREMENTAL PARTIAL PATH CONSTRAINTS (IPPC) 27
4.1. Partial Path Constraints 27
4.2. Incremental Partial Path Constraints 28

4.2.1. Example 29
4.3. Algorithm 30
4.4. Correctness and Completeness 31
5. EXPERIMENTS AND RESULTS 33
5.1. Comparison of DFS and IPPC 35
5.2. IPPC Against Other Strategies 36
5.3. Threats to Validity 39
6. CONCLUSIONS AND FUTURE WORK 42
6.1. Conclusion 42
6.2. Future Work 43
REFERENCES 44
APPENDIX A: SCRIPT FOR COVERAGE MEASUREMENT 51
APPENDIX B: SOURCE CODES FOR SMALL BENCHMARKS 52

LIST OF FIGURES

Figure 2.1. Categorization of Automated Unit Test Generation Approaches . . 8

Figure 2.2. Source Code for Greatest Common Divisor (GCD) Algorithm . . . 12

Figure 2.3. Control-Flow Graph (CFG) of GCD 14

Figure 2.4. Execution Tree of GCD with Maximum Depth = 6 14

Figure 2.5. Concolic Tester, DFS with Bounded Maximum Iterations (CREST) 17

Figure 2.6. Instrumented GCD for Concolic Testing 19

Figure 2.7. Execution Trace for $[a = 4, b = 6]$ 20

Figure 4.1. Incremental Partial Path Constraints (IPPC) 31

Figure 5.1. Relationship between Speedup and #Infeasible Constraints 36

Figure A.1. bcov.pl: Branch Coverage Measurement Script. 51

Figure B.1. Source Code of *bsort* Benchmark 52

Figure B.2. Source Code of *sqrt* Benchmark 53

Figure B.3. Source Code of *prime* Benchmark 55

Figure B.4. Source Code of *factor* Benchmark 57

LIST OF TABLES

Table 2.1. Execution Traces of GCD in SSA Form with length ≤ 6 13

Table 2.2. Full Path Constraints for each Execution Trace in Table 2.1 15

Table 5.1. List of Benchmarks 34

Table 5.2. IPPC Speedup over DFS 35

Table 5.3. Experimental Results (The best results are highlighted) 38

Table 5.4. The Best Concolic Testers 39

LIST OF SYMBOLS

F	False
T	True
ϕ	Partial Path Constraint
π	Full Path Constraint

LIST OF ACRONYMS/ABBREVIATIONS

ART	Adaptive Random Testing
bDFS	Bounded Depth-First Search
CBT	Constraint-Based Testing
CFG	Control-Flow Graph
CS	Constraint Solver
DFS	Depth-First Search
FSM	Finite State Machine
GA	Genetic Algorithm
GCD	Greatest Common Divisor
IPPC	Incremental Partial Path Constraint Algorithm
MBT	Model-Based Testing
PPC	Partial Path Constraint Algorithm
PSO	Particle Swarm Optimization
RT	Random Testing
SBT	Search-Based Testing
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
UUT	Unit Under Test

1. INTRODUCTION

In this chapter, we describe the motivating factors behind our work, present a summary of our contributions and give an outline of this thesis.

1.1. Motivation

Testing is a crucial part of software development. A study conducted by National Institute of Standards and Technology (NIST) in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually [1]. The potential of early detection of bugs and avoiding this cost naturally drives many researches to investigate different testing techniques.

Testing can be done in several levels, defined solely by the test target. These levels are unit testing, integration testing, and system testing [2]. Unit testing, also known as component testing, refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors [3]. Unit testing not just discovers localized defects, but also allows programmers to verify later changes, such as refactoring, do not cause existing code to break. Therefore, unit testing is an important aspect of regression testing for a software product. Although unit testing has been the best practice for years, its adoption in industry is poor. While a few companies have successfully instituted unit testing, such as Microsoft (where 79% of developers are dedicated to writing unit tests [4]), the majority of software developed in industry have either few, outdated, or low quality unit tests, due to the high cost of creating and maintaining the test suite [5].

Testing with manually generated inputs is the predominant technique in industry to ensure software quality. This type of testing accounts for up to 80% of the typical

cost of software development, but manual test generation is expensive, error-prone, and rarely exhaustive. Thus, several techniques have been proposed to automatically generate test inputs [6].

Automated unit testing dates back to 1975 [7], along with other following works [8–10]. These approaches exploit a technique known as symbolic execution. Symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch. Symbolic execution is used in conjunction with an automated theorem prover or constraint solver based on constraint logic programming to generate new concrete inputs (test cases) with the aim of maximizing code coverage. The theorem provers of that date were not sufficent to accomplish such a task, so the research on the subject has become popular only after the development of strong constraint solvers such as Z3 [11] and Yices [12].

Although symbolic execution is effective at generating high coverage tests, it still is not scalable enough to apply in industry. Therefore, concolic testing has been proposed [13]. Concolic testing is an abbreviation for *conc*rete and symb*olic* execution. Concrete execution can be defined as executing the Unit Under Test (UUT) with concrete inputs. Concolic testing needs only the program code to execute and it combines symbolic and random testing to overcome shortcomings of both.

Concolic testing can generate effective test cases, which has been evaluated in previous studies [6, 13–15]. These studies were conducted on small programs, most of which were libraries or utility functions. The software under test may not contain the limitations that large industrial software does. Popular concolic testers such as

CREST [6] is unable to test large systems within a few days [5].

The bottleneck of concolic testing comes from three sources. These are path explosion, path divergence, and constraint solving [16]. Most of the research focuses on the path explosion and the path divergence [17–22]. Instead, our aim in this thesis is to develop methods that improve concolic testing by reducing the constraint solving overhead.

At this point, we describe in what way the automated unit test generation techniques interact with constraint solvers. Automated unit test generator considers the Unit Under Test (UUT) as a system with inputs. The input space is divided into equivalence classes where each equivalence class represents a unique execution path in the UUT. Both concolic and symbolic testing aims to test all execution paths via generating one test input for each equivalence class. To be able to generate the test inputs, the tester collects information on the execution paths by analyzing the program. The information of each execution path is represented as a path constraint, a conjunction of conditions on inputs. At this point, a constraint solver is used to generate inputs that satisfy each path constraint. This way, the automated unit test generator guarantees that the generated test inputs take each execution path of the UUT at least once.

Constraint solving is a serious bottleneck [16]. For small to moderate sized programs, we demonstrate that the cost of constraint solving dominates the cost of executing the UUT. We know that constraint solvers are exponential time algorithms that depend on the number of inputs and the number of conditions. For large path constraints (i.e. long execution paths), we demonstrate that there can be thousands of conditions to solve at once.

We propose a solution to the constraint solving bottleneck, called Incremental Partial Path Constraints (IPPC). A standard concolic tester instruments the unit under test (UUT) to collect operations that either affect or depend on symbolic variables

during a concrete execution. This sequence of operations is called an *execution trace*. A concolic tester symbolically reexecutes the execution trace to generate *path conditions* that solely depend on the symbolic variables. Then, the concolic tester negates the last path condition that is not negated before. At the final step, constraint solver is called only once with all path conditions to generate a new input vector. As noted in [23], solving more but smaller constraints against one large constraint (i.e. a constraint that contains large number of path conditions) is more efficient. Towards this goal, we use an heuristic which selects only a part of the path conditions at the input generation step of a standard concolic tester such that we call the constraint solver multiple times, but using only a few path conditions at each invocation. Conjunction of these selected path conditions are called Partial Path Constraint (PPC). To the best of our knowledge, usage of Partial Path Constraints (PPC) is a novel approach in CBT domain. Our experiments show that we can generate inputs that fall into the same equivalence class (i.e. inputs that force the program into generating the same execution trace) using fewer path conditions than a standard concolic tester. Although we pay the cost of more constraint solver calls on average, we show that our modification results in more than 5x speedup when UUT has many infeasible paths and 3.35x speedup on the average.

1.2. Contributions

In this thesis, we provide the following contributions,

- We propose an input generation strategy based on partial path constraints, called IPPC and incorporate our strategy into a standard concolic testing algorithm.
- We implement our modification using the CREST tool [6], which is a concolic testing framework for C.
- We perform several experiments to demonstrate the effectiveness of IPPC on several benchmarks.

1.3. Outline

We give a background on automated unit test generation and a detailed background on CBT approaches in Chapter 2. We present the related work for concolic testing, constraint solving, and incremental solving strategies in Chapter 3. We describe our strategy in detail with examples, algorithms, and proofs in Chapter 4. We present and discuss our experimental results in Chapter 5. We describe conclude with discussion on validity, future work, and our contributions in total in Chapter 6.

2. BACKGROUND

In this chapter, we describe and categorize various automated unit test generation methods. We categorize all testing methods which depend on a constraint solver as Constraint-Based Testing (CBT). We describe both symbolic execution and concolic testing with algorithms and examples. We explain how constraint solving continues to be a bottleneck through examples.

2.1. Automated Unit Test Generation

A unit is defined as the smallest testable part of a computer program. When subjected to testing, the unit is called Unit Under Test (UUT). In procedural programming, UUT could be an entire module, but more commonly it is an individual function or a procedure. In object-oriented programming, UUT is sometimes an entire interface or a class, but often it is an individual method. Unit testing is done in an isolated and controlled environment. Anything other than the UUT is assumed to work correctly during testing.

Today, automated unit test generation is more common as it is implemented for popular languages such as Java and C. Some programming languages such as Ruby [24] comes with a built-in automated unit test generator whereas there exist unit testing frameworks for other languages such as JUnit for Java [25] and XUnit for C# [26].

Automated unit test generation procedures can be viewed in two categories, black-box and white-box (structural) testing. Black-box approaches treat the UUT as an unknown system with inputs. Black-box approaches do not require the source code of the UUT and are effective at identifying out-of-the-box errors. Black-box approaches can be utilized to perform stress tests and are easy to setup. However, black-box approaches may fail to find many bugs, or take a long time before finding bugs. We

give a general categorization of automated unit test generation approaches in Figure 2.1. We discuss Symbolic Execution and Concolic Testing in Section 2.2.

Executing the UUT with random inputs is called *Random Testing* (RT) or Monkey Testing. Improved black-box testing such as Adaptive Random Testing (ART) is also proposed [16, 27]. ART exploits a distance metric defined between inputs of the UUT. If a group of randomly generated inputs does not hit a bug, ART generates new inputs farther from the correctly executing inputs. ART can be viewed as sampling from a non-uniform distribution of inputs.

Model-Based Testing (MBT) approaches still view the UUT as a black-box entity, but acknowledge that the UUT has an internal state which changes as it processes inputs [16]. Although MBT treats the UUT as a black-box, it requires some model of the UUT be given or extracted such as a Finite State Machine (FSM). MBT generates inputs from the model.

Fuzz Testing is also researched to generate close-to-valid inputs from an input model which tend the find the deepest crashes. One of the early tools which employs Fuzz Testing is *crashme* [28]. It is introduced in 1991 as a tool to test robustness of Unix systems.

White-box unit testing views the UUT as a computer program with different execution paths. The UUT executes one of its execution paths depending on the input. White-box unit testing approaches fall into two main categories, Constraint-Based Testing (CBT) and Search-Based Testing (SBT).

All testers which use a constraint solver to generate test inputs are called Constraint-Based Testing (CBT) approaches. We make a detailed discussion of CBT in Section 2.2.

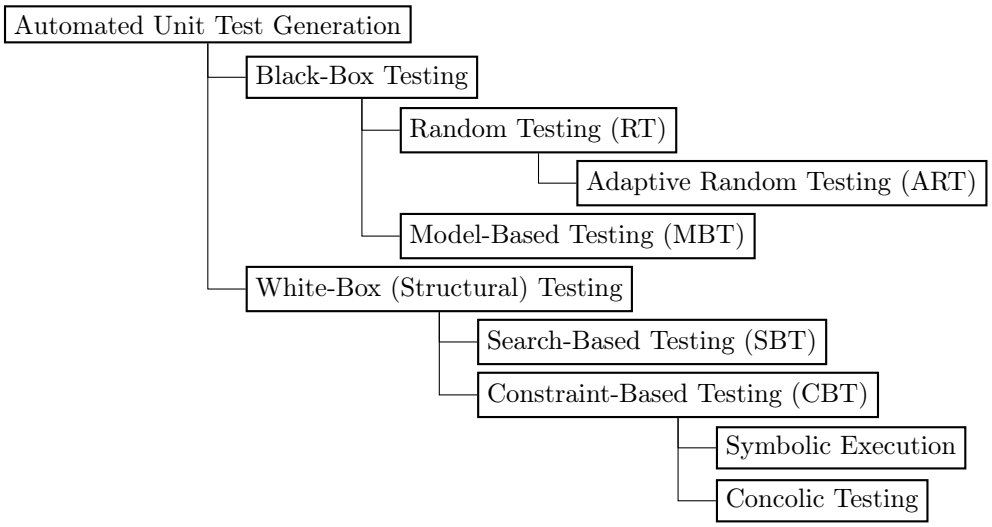

Figure 2.1. Categorization of Automated Unit Test Generation Approaches

SBT uses Genetic Algorithm (GA) to generate new inputs to cover as many execution paths of the UUT as possible [16]. EvoSuite is a common Search-Based Testing tool for Java [29]. SBT is proposed to avoid the usage of constraint solvers, thus completely avoid the constraint solving bottleneck while achieving less coverage. Also, SBT is not good at generating test inputs if the execution path contains conditions like string equalities or regular expressions which require a constraint solver to satisfy [30].

2.2. Constraint-Based Testing

In this section, we give definitions to facilitate the understanding of Constraint-Based Testing (CBT) and describe CBT using these definitions. We present two main CBT techniques, Symbolic Execution and Concolic Testing in Sections 2.2.1 and 2.2.2, respectively.

Definition 2.1. *(Symbolic Variable).* A symbolic variable is a program variable which initially represents all values of its range.

Definition 2.2. *(Execution Trace).* An execution trace of a UUT is a sequence of assignments and conditions on symbolic variables which is a result of one particular execution of the UUT.

Definition 2.3. *(Symbolic Execution Tree).* A symbolic execution tree of the UUT is a binary tree of all execution traces of the UUT. Assignments of the execution traces are denoted on the nodes and conditions are denoted on the edges of the tree.

Definition 2.4. *(Execution Path).* An execution path is a path on the symbolic execution tree which starts from the root of the tree and ends at a leaf node.

Definition 2.5. *(Bounded Symbolic Execution Tree).* A bounded symbolic execution tree of depth k is a subgraph of the symbolic execution tree which contains the root, any node whose distance to the root is not greater than k, and all edges between these nodes.

Definition 2.6. *(Path Condition).* A path condition is a function from the test inputs of the UUT to $\{T, F\}$.

Definition 2.7. *(Path Constraint).* A path constraint is a conjunction of path conditions.

In Constraint-Based Testing (CBT), the UUT is divided into several logical parts and inputs are represented as symbolic variables. A path constraint is extracted for each such part. Then, a constraint solver is used to generate one test input for each feasible path constraint. Coverage of the resulting test suite depends on the partitioning of the UUT. DeMillo et al. [31] coined the term CBT by doing mutation analysis, describing the test cases using algebraic constraints and then solving the constraints using a constraint solver.

Symbolic Execution and Concolic Testing view the UUT as a collection of execution paths and extract constraints for each execution path. Then, they generate one input for each feasible path constraint to achieve path coverage.

2.2.1. Symbolic Execution

Symbolic Execution (or symbolic evaluation) is a very old testing technique that dates back to 1975. Due to the lack of powerful constraint solvers, Symbolic Execution used to be considered neither scalable nor practical until recently.

Definition 2.8. *(Full Path Constraint).* A path constraint (π) is a full path constraint of an execution path e if and only if all test inputs that satisfy π follow e and all test inputs that does not satisfy π does not follow e.

In Symbolic Execution, inputs are denoted as symbolic variables. Initially, a symbolic variable represents every value in the range of the input. Then, we extract the bounded symbolic execution tree of the UUT by inspecting the code line by line. If a statement does not involve any of the symbolic variables, that statement is either a terminate statement or it is ignored. If a statement contains symbolic variables or it is a terminate statement, we create a node in the tree for that statement. If the statement is a terminate statement, the added node is a leaf and has no children. If the statement is a branch condition statement, the added node has two children, one when the branch condition evaluates to T and one for the branch condition evaluates to F. If the statement is an assignment statement, the added node has only one child.

Symbolic execution trees generated during symbolic execution are in Single Static Assignment (SSA) form, in which each variable is assigned exactly once. To be able to achieve this, we split each symbolic variable x into versions x_0, x_1, etc., one subscripted variable for each assignment. The advantage of SSA form is that it makes the generation of symbolic execution trees simple (e.g. in Figure 2.4, the static single assignment on the node L_7 allows us to denote conditions on the subtree of L_7 to be generated without any calculations required).

In Figure 2.2, we present an example UUT written in C. Let both inputs a and b be symbolic variables. Since we use the SSA form, we start with a node where $i = 0, j = 0$ and $a_i = a$ and $b_j = b$. Then, we start the symbolic execution by executing the first line, L_0. L_0 is a branch condition. We transform the condition $a > 0$ to it's SSA form as $a_i > 0$. We create two children and connect $a_i > 0$ to one and the negation $a_i \leq 0$ to the other. Then we symbolically execute each children and continue. L_1 is a terminating statement and therefore is a leaf node on the tree. Then, we symbolically execute L_2, L_3, L_4, L_8, and L_5 as we previously explained in this example. L_6 and L_7 are assignment statements to the symbolic variables, so we increase the corresponding subscript by 1 and pass the assignment to the new symbolic variable. We present the generated bounded symbolic execution tree in Figure 2.4.

Definition 2.9. *(Test Suite).* A tests suite is a collection of test inputs.

From the symbolic execution tree, we extract the execution traces (each execution path corresponds to an execution trace). Then, we convert these execution traces to full path constraints. We use a constraint solver to generate test inputs for each full path constraint. We form the test suite by taking all generated test inputs.

Converting execution traces into full path constraints is a straightforward process. Execution traces are written in SSA form, which means they contain subscripted variables. First, we write any subscripted variable x_i in terms of x_0. Then, we replace any variable x_0 with the original symbolic variable x. At last, we remove all statements which equates the same expression to itself like $x = x$. For example, an execution trace like $[a_0 > 5, a_1 = 2a_0, a_1 \neq 12]$ is first converted into $[a_0 > 5, 2a_0 = 2a_0, 2a_0 \neq 12]$. Then, we remove the subscript as $[a > 5, 2a = 2a, 2a \neq 12]$. At last we remove the second statement and get $[a > 5, 2a \neq 12]$.

We generally bound the depth of the generated execution tree. This is because there can be too many paths in the symbolic execution tree (execution tree where all

```
int gcd(int a, int b) {
    if (a <= 0) {        // L0
        return ERROR;    // L1
    }

    if (b <= 0) {        // L2
        return ERROR;    // L3
    }

    while (a != b) {     // L4
        if (a > b) {     // L5
            a = a - b; // L6
        } else {
            b = b - a; // L7
        }
    }

    return a;            // L8
}
```

Figure 2.2. Source Code for Greatest Common Divisor (GCD) Algorithm

leaves are terminal nodes), which is called the *path explosion* problem.

As a last note, we describe feasibility of a path constraint which is used in the literature [32]. Infeasible path constraints correspond to the impossible executions of the UUT (i.e. there are no such input that follows the corresponding execution path).

Definition 2.10. *(Feasibility of a Path Constraint).* A path constraint is feasible if and only if the path constraint is satisfiable.

2.2.1.1. Example. We present an example UUT in Figure 2.2 written in C. We also present the Control-Flow Graph (CFG) of the UUT in Figure 2.3 to represent all execution paths of the UUT in graph notation. Symbolic Execution can be viewed as unwinding the CFG and generating a tree of possible execution traces using Static Single Assignment (SSA) form. The tool we used, CREST, automatically generates SSA form using the CIL instrumentation framework [33]. Since the while-loop would cause a path explosion, we bound the tree to have a maximum depth of 6. We can generate execution traces from the execution tree. We present the execution traces in Table 2.1. For example, the execution trace $[a > 0, b \leq 0]$ is generated from the path $L_0 \rightarrow L_2 \rightarrow L_3$ from Figure 2.4. Each execution trace starts with $i = 0, j = 0, a_i = a, b_j = b$ and ends with either a `RETURN` statement or a `'-'` denoting that a return statement is not reached until the maximum depth. From the execution traces, we generate path constraints as shown in Table 2.2. We call these path constraints *full path constraints* since they contain all conditions on the execution path. Full path constraints only depend on the input variables and can be given as input to a Constraint Solver (CS). If the full path constraint (π) is feasible, CS(π) returns an input that satisfy the path constraint. In this example, the test suite contains 7 test inputs and covers all execution paths of length 6 or shorter (We can see that each test input in Table 2.2 follows a unique path in the symbolic execution tree in 2.4).

Table 2.1. Execution Traces of GCD in SSA Form with length ≤ 6

#	Execution Trace	Return Value
1	$a_0 \leq 0$	ERROR
2	$a_0 > 0, \quad b_0 \leq 0$	ERROR
3	$a_0 > 0, \quad b_0 > 0, \quad a_0 = b_0$	a_0
4	$a_0 > 0, \quad b_0 > 0, \quad a_0 \neq b_0, \quad a_0 > b_0, \quad a_1 = a_0 - b_0, \quad a_1 = b_0$	a_1
5	$a_0 > 0, \quad b_0 > 0, \quad a_0 \neq b_0, \quad a_0 > b_0, \quad a_1 = a_0 - b_0, \quad a_1 \neq b_0$	-
6	$a_0 > 0, \quad b_0 > 0, \quad a_0 \neq b_0, \quad a_0 \leq b_0, \quad b_1 = b_0 - a_0, \quad a_0 = b_1$	a_0
7	$a_0 > 0, \quad b_0 > 0, \quad a_0 \neq b_0, \quad a_0 \leq b_0, \quad b_1 = b_0 - a_0, \quad a_0 \neq b_1$	-

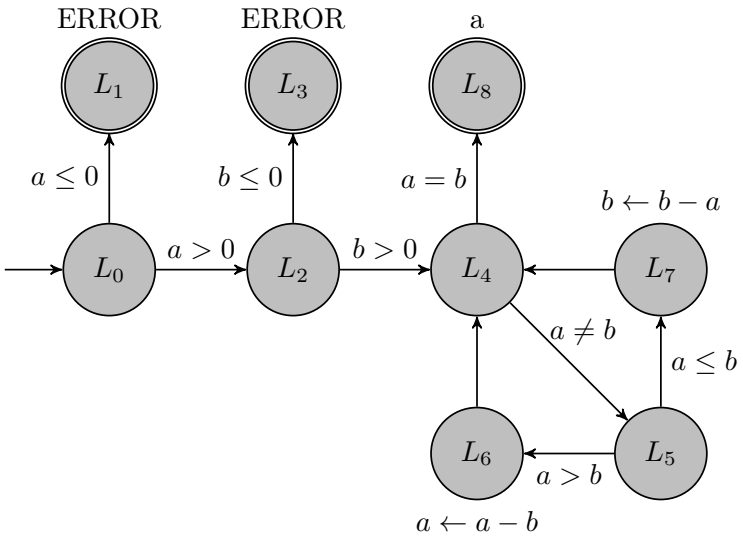

Figure 2.3. Control-Flow Graph (CFG) of GCD

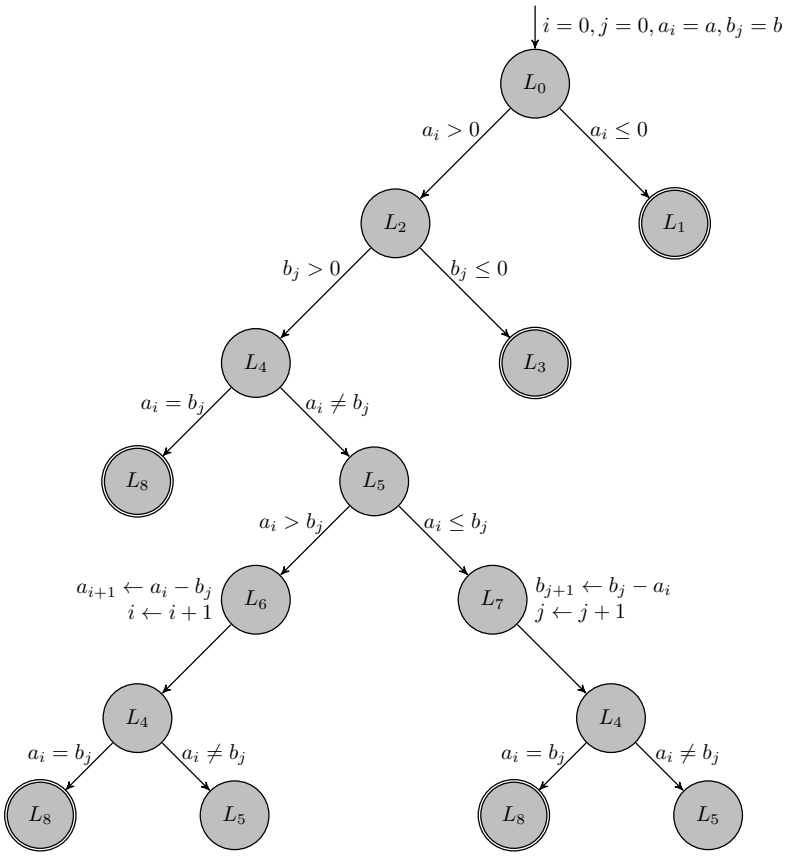

Figure 2.4. Execution Tree of GCD with Maximum Depth = 6

Table 2.2. Full Path Constraints for each Execution Trace in Table 2.1

#	Full Path Constraint (π)	CS(π)
1	$(a \leq 0)$	$a = 0, b = 0$
2	$(a > 0) \wedge (b \leq 0)$	$a = 1, b = 0$
3	$(a > 0) \wedge (b > 0) \wedge (a = b)$	$a = 1, b = 1$
4	$(a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a > b) \wedge (a - b = b)$	$a = 2, b = 1$
5	$(a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a > b) \wedge (a - b \neq b)$	$a = 3, b = 1$
6	$(a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a = b - a)$	$a = 3, b = 6$
7	$(a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a \neq b - a)$	$a = 3, b = 7$

We give the Definitions 2.11 and 2.12 to measure the performance of a constraint-based tester in terms of constraint solver calls.

Definition 2.11. *(Size of a Path Constraint).* The size of a path constraint is defined as the number of path conditions that the path constraint contains. It is denoted as s.

Definition 2.12. *(Constraint Solving Overhead).* Constraint solving overhead of a constraint-based tester over a UUT is defined as

$$\text{overhead} = \frac{1}{N} \sum_{i=1}^{N} s_i$$

where N is the number of CS calls by the tester and s_i is the size of the i^{th} path constraint.

We calculate the *constraint solving overhead* as the average number of conditions in a path constraint used by the CS as denoted in Definition 2.12. In Example 2.2.1.1, symbolic executor made 7 CS calls as shown in Table 2.2 with average path constraint size of $(1+2+3+5+5+5+5)/7 \approx 3.71$. If the bound k is increased, we can see that the average path constraint size will increase as well.

2.2.2. Concolic Testing

Since symbolic execution is bounded, it fails to execute critical parts deep inside the UUT. To be able to test the UUT without extracting the full execution tree, Concolic Testing has been proposed [13]. Concolic Testing is an abbreviation for combination of *conc*rete and symb*olic* testing.

At this point, we present an overview of concolic testing procedure. First, the concolic tester instruments the UUT (i.e. adds statements to specific places of the source code), so the UUT generates its own execution trace whenever it is executed with concrete inputs. This procedure is automated using some instrumentation framework (CREST uses CIL [33]). Then, the UUT is executed with random inputs. The concolic tester captures the execution trace and generates its full path constraint. We order the path conditions inside the path constraint by their order in the execution trace. According to this ordering, the *last path condition* that is not negated before is negated. After the negation we get a new path constraint. We give the new path constraint to the constraint solver to generate new inputs. We continue to make concrete executions with the new inputs and generate new inputs until no path condition remains unnegated (exhaustion) or a bound is hit. We bound the maximum iterations of this procedure since CREST implements the standard concolic testing in this way as well. This algorithm is called DFS after Depth-First Search (given in Algorithm 2.5), since it corresponds to a depth-first search of the symbolic execution tree of the UUT. Although there could be other strategies such as random branch selection, DFS is the most common strategy [6, 13].

CREST's standard concolic testing algorithm (denoted by DFS) is shown in Algorithm 2.5. We initially give $nIterations \leftarrow 0$, $TestInput \leftarrow random\_input$ and $TestSuite \leftarrow \emptyset$. We can see from line 1 that the algorithm is bounded by the maximum number of iterations. At line 2, we check the test input because constraint solver is assumed to return *null* if a test input can not be generated, i.e. given path constraint

```
input  : uut (unit under test), nIterations, maxIterations, testInput, testSuite
output: testInputs
1  if nIterations < maxIterations then
2      if testInput ≠ null then
3          nIterations ←nIterations +1;
4          π ←execute(uut, testInput);
5          add(testSuite, testInput);
6      end
7      for i ←sizeOf(π)−1 to 0 do
8          if !isNegatedBefore(π[i]) then
9              setNegatedBefore(π[i]);
10             π[i] ← ¬π[i];
11             testInput ← constraintSolver({π[0]...π[i]});
12             return crest(uut, nIterations, maxIterations, testInput, testSuite);
13         end
14     end
15 end
16 return testSuite;
```

Figure 2.5. Concolic Tester, DFS with Bounded Maximum Iterations (CREST)

is infeasible. We increment the number of iterations only if the path constraint is feasible. At line 4, we execute UUT with *input* and save the full path constraint of the execution trace as π. Lines 8 and 9 ensure that the algorithm performs a depth first search (DFS). According to the ordering of the execution trace, we always negate the last unvisited condition of a path constraint. If there exists no path condition that is not negated before, we stop because all paths are executed. At line 11, constraintSolver takes a path constraint and returns a satisfying test input. We don't terminate the solver, we use it in its built-in incremental mode. Notice that we exclude the path conditions which come after the negated path condition on the full path constraint. Negated path condition will force the program to a different execution trace. So, path

conditions coming after the negated condition is not a part of the new execution trace and therefore should be removed. If the unnecessary path conditions are not excluded, they may cause false path constraint infeasibilities. For example, if we are going to negate the second path condition of $\pi = (a > 0) \wedge (b > 0) \wedge (a = b)$, we exclude the third condition and get $\pi' = (a > 0) \wedge (b \leq 0)$. If we used $(a > 0) \wedge (b \leq 0) \wedge (a = b)$, this would be infeasible and therefore would not generate any test input although the execution path was feasible (there exists test input i such that i follows the execution path). False infeasibilities force the concolic tester not to generate any test input for the path constraint and therefore cause a decrease in test coverage.

In Algorithm 2.5, we use functions `isNegatedBefore` and `setNegatedBefore`. We keep a map (preferably a hash map) of path conditions to boolean values where initially all path conditions map to F. When we call `setNegatedBefore`, we set the value of the path condition to T. `isNegatedBefore` returns the boolean value mapped to the given path condition.

2.2.2.1. Example. We consider the UUT from the previous example, given in Figure 2.2. For demonstration purposes, we performed an equivalent manual instrumentation in Figure 2.6 whereas in our implementation uses CIL for this purpose. For the sake of simplicity, we narrow the range of inputs to the set of digits $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Then, we generate random values for a and b like $a = 4$ and $b = 0$. Executing the instrumented UUT will traverse lines $L_0 \rightarrow L_2 \rightarrow L_3$ in the code and therefore generate the execution trace `"i = 0, j = 0, a[i] = a, b[j] = b, a[i] > 0, b[j] <= 0, RETURN ERROR"`. From this execution trace, we get the full path constraint $\pi = (a > 0) \wedge (b \leq 0)$. We negate the last constraint to generate the new path constraint $\pi' = (a > 0) \wedge (b > 0)$. Let CS($\pi'$) return $a = 4$ and $b = 6$. When we execute the UUT with these inputs we get the new execution trace as shown in Figure 2.7 (the execution follows $L_0 \rightarrow L_2 \rightarrow L_4 \rightarrow L_5 \rightarrow L_7 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_8$ in Figure 2.3). We find that the full path constraint of this new execution trace is

```
int gcdInstrumented(int a, int b) {
    fprintf(TRACE, "i = 0, j = 0, a[i] = a, b[j] = b, ");

    if (a <= 0) {       // L0
        fprintf(TRACE, "a[i] <= 0, ");
        fprintf(TRACE, "RETURN ERROR\n");
        return ERROR;   // L1
    }
    fprintf(TRACE, "a[i] > 0, ");

    if (b <= 0) {       // L2
        fprintf(TRACE, "b[j] <= 0, ");
        fprintf(TRACE, "RETURN ERROR\n");
        return ERROR;   // L3
    }
    fprintf(TRACE, "b[j] > 0, ");

    while (a != b) {    // L4
        fprintf(TRACE, "a[i] != b[j], ");

        if (a > b) {    // L5
            fprintf(TRACE, "a[i] > b[j], ");
            a = a - b; // L6
            fprintf(TRACE, "a[i+1] = a[i] - b[j], i = i + 1, ");
        } else {
            fprintf(TRACE, "a[i] <= b[j], ");
            b = b - a; // L7
            fprintf(TRACE, "b[j+1] = b[j] - a[i], j = j + 1, ");
        }
    }
    fprintf(TRACE, "a[i] == b[j], ");
    fprintf(TRACE, "RETURN a[i]\n");
    return a;           // L8
}
```

Figure 2.6. Instrumented GCD for Concolic Testing

$\pi'' = (a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a \neq b-a) \wedge (a > b-a) \wedge (a-[b-a] = b-a)$. Therefore, we can say that the concolic tester has successfully found an input which satisfies a path constraint of size 7 by solving a path constraint of only size 2, which is an improvement over symbolic execution.

```
"i = 0, j = 0, a[i] = a, b[j] = b, a[i] > 0, b[j] > 0, a[i] != b[j],
 a[i] <= b[j], b[j+1] = b[j] - a[i], j = j + 1, a[i] != b[j], a[i] >
  b[j], a[i+1] = a[i] - b[j], i = i + 1, a[i] == b[j], RETURN a[i]"
```

Figure 2.7. Execution Trace for $[a = 4, b = 6]$

After a few iterations of concolic testing, path constraints grow in size. When we continue the example, we negate the last path condition in π'' and are now going to solve $\pi''' = (a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a \neq b-a) \wedge (a > b-a) \wedge (a-[b-a] \neq b-a)$. Solution to this path constraint (e.g. $a = 5$, $b = 6$) is going to lead the concolic tester to solve even larger constraints (e.g. `gcd(5, 6)` will generate an execution trace which follows $L_0 \rightarrow L_2 \rightarrow L_4 \rightarrow L_5 \rightarrow L_7 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow L_4 \rightarrow L_8$). Our experimental results show that to make 1000 iterations, the standard concolic tester solves path constraints of size 161 on average for this UUT as shown in Table 5.3. So, although being better than symbolic execution, constraint solving is still a bottleneck in concolic testing.

Definition 2.13. *(Overapproximation).* A proposition p is an overapproximation of q if and only if

$$q \rightarrow p.$$

Definition 2.14. *(Partial Path Constraint).* A partial path constraint is an overapproximation of the full path constraint. It is denoted as ϕ.

To decrease the constraint solving burden, we aim to use partial path constraints instead of the full path constraints and use simple heuristics for the selection of the partial path constraints. To the best of our knowledge, usage of partial path constraints is a novel approach in CBT domain. We give a detailed description of our approach in Chapter 4.

3. RELATED WORK

In this chapter, we discuss previous work on concolic testing and constraint solving. We also discuss the history of the term incremental solving and how it is used in concolic testing.

3.1. Concolic Testing

There exist a large number of concolic testing tools used in domains such as unit testing for a specific language, model checking, web service testing, and smartphone testing. The most notable related tools and work can be listed as follows:

CUTE [13] is the earliest concolic testing tool (along with DART [23]). It is implemented for testing C programs. It uses source code instrumentation (CIL [33]) to collect symbolic information on a given unit under test (UUT). This approach follows a bounded depth-first strategy (DFS) to exhaust as many computation paths as possible. Even though this work is accepted as the basis of concolic testing, it also includes three major optimizations on constraint solving to boost performance. These optimizations are *Fast Unsatisfiability Check* (checks if the last path condition is the syntactic negation of another path condition in the path constraint.), *Common Sub-Constraint Elimination* (identifies and eliminates common path conditions which occur multiple times in the path constraint) and *Incremental Solving* which we describe in Section 3.3.

Many concolic testers (CUTE [13], LCT [34], jCUTE [35], Jalangi [36]) use bounded depth-first search (*bDFS*) instead of bounding the number of iterations as in IPPC and CREST [13, 37]. *bDFS* can be viewed as an attempt to keep the number of path conditions in a path constraint limited. We believe that our approach makes bounding the number of iterations more preferable to bounding the depth of the

search as in *bDFS* since experiments show that IPPC already reduces the size of path constraints significantly.

KLEE [14] is a widely known pure symbolic execution tool for C where constraint solving is identified as a major time consuming bottleneck. They introduce the idea of caching as an improvement. The main motivation in this idea is that verifying a cached input to see if it satisfies a path constraint is easier than deciding on the input only from the path constraint. We mention a possible improvement based on caching as a future work.

CREST [6] is a testing framework for C which implements different strategies to improve performance of the concolic testing. This work shows that by statically analyzing the control flow graph of a program, it is possible to collect useful information which may guide and improve the concolic tester. CREST implements the basic strategy and optimizations of CUTE as well as other concolic testing strategies. CREST is also publicly available for download unlike CUTE. Therefore, we implemented our strategy on top of CREST and we compared our results against other strategies.

ConCREST [38] extends CREST to generate tests for concurrent C programs. ConCREST solves constraints during testing, like CREST. Therefore, our constraint solving strategy is easily implementable on top of ConCREST to optimize testing of concurrent UUTs. We were not able to test our heuristics on top of ConCREST due to its unavailability.

A recent work on concolic testing introduces interpolation as a technique to subsuming paths that are guaranteed to be bug-free [17]. This approach attacks the path explosion problem. Optimizations on path explosion problem is compatible with our optimization on constraint solving since we do not change any part of concolic testing methodology other than the constraint solving part.

Racageddon [39] extends concolic testing to multithreaded programs for Java, but it specializes on detection of race conditions, it does not generate test suites for multithreaded units.

ACTEVE [40] is a concolic tester which specializes on testing of smartphone apps. This work shows how the methodology of concolic testing can be applied on different domains.

ExpliSAT [41] applies concolic testing to the domain of model checking. In this concolic model checker, the model checker traverses states of the model representing the software being checked, while storing both a concrete state and a symbolic state. The symbolic state is used for checking properties on the software, while the concrete state is used to avoid reaching unreachable state.

3.2. Constraint Solving

All concolic testers require a *constraint solver*. Z3 and Yices are commonly used constraint solvers [11, 12]. Constraint solving is known to be an NP-Complete problem [42]. Solving constraints of different models is an open area of research since constraints may include *different types of variables* and *different operations on variables* such as concatenation on strings and modulo on integers. Details on constraint solver limitations for concolic testing can be found in [5].

Yices is a constraint solver which uses Satisfiability Modulo Theories (SMT). CREST uses Yices as default. In CREST, Yices is used in incremental mode. We discuss this incremental solving optimization in Section 3.3.

For Yices, path conditions are assumed to be in linear form, $k \bowtie \sum_{i=1}^{N} c_i x_i$ where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $k \in \mathrm{R}$, $\forall i \in \{1, ..., N\}, c_i \in \mathrm{R}$ and N: total number of symbolic variables. However, this form does not cover *division* and *modulo* operations. Z3

supports *division* and *modulo* operations. Also, Z3 gives very limited support for non-linear constraints. CREST-z3 is a modified CREST which uses Z3 constraint solver instead of Yices in order to support broader arithmetic [43]. Although we used Yices, we could easily implement IPPC on top CREST-z3.

Path conditions can easily become non-linear (e.g. multiplication of two input variables). CalCS [44] solves non-linear convex path conditions. Functions that satisfy $f(\Theta x - (1-\Theta)y) \leq \Theta f(x) + (1-\Theta)f(y)$ and whose domain set is convex are called *non-linear convex functions*. Yices and Z3 are not able to solve non-linear conditions. In non-linear case, Yices and Z3 try to assign constants to inputs until the condition becomes linear.

Path conditions may involve high-precision rational numbers (e.g. $5 = x - \sqrt{2}$). CORAL [45] uses Particle Swarm Optimization (PSO) to solve constraints that involve high-precision arithmetic. CREST does not support floating-point arithmetic.

Path conditions may involve string operations. HAMPI [46] solves conditions involving *regular expressions* and even more powerful expressions which accept members of a *fixed-size context-free language*. CREST does not support string operations.

In summary, there exist several works that improve the performance of constraint solver power, so it becomes more and more adequate for test input generation. We seek to exploit domain knowledge to decrease the constraint solving burden.

3.3. Incremental Constraint Solving

The idea of incremental constraint solving for concolic testing is as old as concolic testing itself and suggested along with CUTE, one of the first concolic testing tools [13]. However their incremental solving idea should not be confused with the one that we present here. In CUTE, since they negate only one path condition, they keep the

variables which are irrelevant to that path condition fixed. Therefore, they solve for variables which are only relevant to the negated path condition and decrease the burden on the constraint solver [13]. In our approach, CREST still does the incremental solving optimization of CUTE. On top of that, we also produce path contraints with fewer path conditions than a standard concolic tester.

Solving only a subset of path conditions instead of the whole path constraint is an approach recently used in finding integer overflows [47]. The motivation is supported by observing that many of the path conditions of an execution trace are irrelevant w.r.t. integer overflows. In our work, we make a more general assumption that some of the path conditions should be irrelevant to the execution trace itself. Experimental results support our motivational assumption.

Our approach to decrease constraint solver burden is applied in different domains such as Model Checking. IC3 [48] uses the incremental approach to prove reachability of states, instead of a monolithic approach where one formulates the required specification as a large boolean expression and gives it to a constraint solver. In the incremental approach, the aim is to make thousands of small queries instead of a few large queries, which has been shown to result in an increase in performance, thereby an increase in scalability. This incremental approach is very similar to the Counter-Example Guided Abstraction Refinement (CEGAR) methodology which is commonly used in model checking [49].

Random branch selection (RND) is an approach introduced in CREST [6]. In random branch selection one negates each path condition of a path constraint with probability 0.5. A variant of this strategy negates only one path condition at random and calls the constraint solver with that path condition.

Control flow directed search (CFG) is a search strategy which guides the search using the static structure of the UUT. CFG assigns weights to edges of the control

flow graph of the UUT and calculates distances to each unvisited branch. Then, CFG solves a path constraint which leads to the unvisited branch with the least distance to the current execution trace [6]. We compare our approach with both CFG and RND.

A recent study on CREST proposes a new search strategy, called DYNASTY, which uses the control flow graph of the UUT to guide the search as in CFG technique [50]. They modify CREST to increase branch coverage with fewer iterations. Their approach is based on avoiding infeasible paths, whereas our approach decreases the penalty of hitting an infeasible path by using partial path constraints.

4. INCREMENTAL PARTIAL PATH CONSTRAINTS (IPPC)

In this chapter, we explain our approach on decreasing the constraint solving bottleneck in detail. We first describe the motivation behind using overapproximations of full path constraints which we call partial path constraints in Section 4.1. However, we argue that using partial path constraints drop completeness and we propose our solution, Incremental Partial Path Constraints (IPPC) in Section 4.2. To the best of our knowledge, usage of Partial Path Constraints (PPC) is a novel approach in CBT domain. We provide the algorithm and the correctness proof as well as a motivating example to explain how IPPC improves the standard concolic testing.

4.1. Partial Path Constraints

In this section, we describe our motivation behind using partial path constraints instead of the full path constraint itself using three examples.

(i) In the example at Section 2.2.1.1, we demonstrated how full path constraints are used to generate test inputs. Then, we described the concolic testing as an improvement. In Section 2.2.2.1, the path constraint $\pi'' = (a > 0) \wedge (b > 0)$ was used to generate a test input. To generate the same input, symbolic executor would use the full path constraint $\pi''' = (a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a \neq b - a) \wedge (a > b - a) \wedge (a - [b - a] \neq b - a)$. Notice that $\pi''' \rightarrow \pi''$, in other words π'' is an *overapproximation* of π'''. This shows that overapproximations can be used instead of full path constraints to generate test inputs.

(ii) In π''' mentioned above, notice the two path conditions $(a \neq b - a)$ and $(a > b - a)$, let us call these conditions p and q, respectively. We can see that the relation $q \rightarrow p$ holds. From the satisfiability point of view, $p = (a \neq b - a)$ is redundant

(unnecessary). If a path condition p is an overapproximation of another path condition q in the full path constraint, then we can safely remove p. This further motivates us not to use the full path constraint itself.

(iii) Consider $\pi = (a > 0) \wedge (b > 0) \wedge (a = b)$ and $\phi = (a = b)$. Notice that ϕ is an overapproximation of π. Assume that constraint solver returns one of the satisfying inputs randomly. Then, for $a, b \in \mathbb{N}$, with 0.25 probability, CS(ϕ) returns an input that satisfies π. For $\phi' = (b > 0) \wedge (a = b)$, CS($\phi'$) returns an input that satisfies π with 0.5 probability. Therefore, even if there is no redundancy, usage of partial path constraints allows us to find correct inputs with smaller queries to the constraint solver.

4.2. Incremental Partial Path Constraints

In this section, we propose the Incremental Partial Path Constraints (IPPC) approach which utilizes partial path constraints. We explain why IPPC should be used as a partial path constraints methodology. We design the IPPC to replace the constraint solver call in a constraint-based tester.

The symbolic execution and concolic testing algorithms are correct and complete algorithms assuming the underlying constraint solver is correct and complete. A test input returned by a correct algorithm always drives the execution into the execution path of the full path constraint whereas a complete algorithm terminates as we discuss in Section 4.4. When partial path contraints are used instead of full path constraints, we drop completeness. This is because the constraint solver can not guarantee that the inputs generated via the partial path constraint will satisfy the full path constraint. When a generated input does not satisfy the full path constraint, *path divergence* problem occurs. *Path divergence* is described as the case that the generated input does not force the UUT into the execution path of the full path constraint. The input follows the execution path of the full path constraint until a branch condition which we call the *cause of divergence* and takes the wrong side of the branch. Therefore, we

may not execute some execution paths in the UUT.

To avoid dropping completeness, we propose Incremental Partial Path Constraints (IPPC) methodology. In this methodology, to generate satisfying inputs for π, we first start with selecting ϕ as the last path condition in π. If ϕ is infeasible, then π is also infeasible (see Theorem 4.1), and we conclude that the execution path is *infeasible*. Otherwise, CS(ϕ) generates a test input i which satisfies ϕ. If i also satisfies π, we can use i as our test input. If i does not satisfy π, there must exist a cause of divergence (c_d) in π where i does not satisfy c_d. Then we update ϕ using the rule $\phi \leftarrow \phi \wedge c_d$ and repeat the procedure until we prove infeasibility of the execution path or we reach a fixed point where $\pi \leftrightarrow \phi$ and at that point, satisfying inputs of ϕ must also satisfy π.

The selection of the initial partial path constraint in IPPC is important. We designed IPPC on top of CREST, which uses concolic testing. Therefore, we used an heuristic which lets the last path condition in the full path constraint be the initial partial path constraint. This heuristic is based on the fact that in concolic testing, we change only the last path condition. CREST utilizes Yices constraint solver in incremental mode, in which the old input is used whenever it does not appear in the path constraint. So, intuitively, we expect to make a small change on the inputs we have to make the last condition satisfied and hope that the previous path conditions are still satisfied after the change.

4.2.1. Example

Consider $\pi = (a > 0) \wedge (b > 0) \wedge (a \neq b) \wedge (a \leq b) \wedge (a \neq b - a) \wedge (a > b - a) \wedge (a - [b - a] = b - a)$ from the example in Section 2.2.2.1.

Let $\phi^1 = (a - [b - a] = b - a)$. Yices generates $a = -2, b = -3$ for CS(ϕ^1). However, this input does not satisfy π due to the cause of divergence $c_d^1 = (a > 0)$. Then we generate $\phi^2 = (a - [b - a] = b - a) \wedge (a > 0)$. Now Yices generates $a = 2, b = 3$.

The generated input satisfies π therefore we stop.

In this example, we found the correct test input by using a path constraint of size 2 instead of 7. However, we pay the price of making 2 constraint solver calls instead of 1. As noted in [23], solving more but smaller constraints against one large constraint is desirable, so we implemented IPPC on top of CREST. For the GCD example with 1000 iterations, IPPC makes 11K CS calls in total while the DFS makes 5.6K CS calls. However, IPPC's CS calls are very small, path constraints have an average size of 1.9, whereas DFS's CS calls have a path constraint size of 161 on average. IPPC finishes testing in 7.5 seconds whereas DFS finishes testing in 22.6 seconds.

4.3. Algorithm

IPPC is exactly the same as Algorithm 2.5 except we replace line 11 with $input \leftarrow$ IPPC $(UUT, \{\pi[0]...\pi[i]\}, \{\pi[i]\})$. We describe IPPC in Algorithm 4.1. We keep the constraint solver running in incremental mode.

Line 1 of Algorithm 4.1 invokes the constraint solver with ϕ. If the constraint solver is unable to generate a test input, we conclude that the full path constraint is also *infeasible* and return *null*. We prove the infeasibility of a full path constraint given the infeasibility of a partial path constraint in Theorem 4.1. The for loop starting at line 5 checks if the test input satisfies the full path constraint. If *input* satisfies π, we can return *input*. If not, we *learn* the unsatisfied path condition by adding it to ϕ and generate a new test input. This process may be continued until $\phi = \pi$ in the worst case where either an input can be generated or π is infeasible given that the constraint solver is sound (if exists, returns a correct test input) and complete (terminates and is correct for all cases). This worst case condition is rare or non existent at least in our experiments.

```
input  : uut (unit under test), π (full path constraint), φ (partial path
         constraint)
output: testInput
1  testInput ← constraintSolver(φ);
2  if testInput = null then
3  |  return null;
4  end
5  for i ← 0 to sizeOf(π) − 1 do
6  |  if !sat(testInput, π[i]) then
7  |  |  append(φ, π[i]);
8  |  |  return IPPC (uut, π, φ);
9  |  end
10 end
11 return testInput;
```

Figure 4.1. Incremental Partial Path Constraints (IPPC)

4.4. Correctness and Completeness

To be able to produce readable proofs, we assume all boolean operations on a path constraint are performed on the conjunction of all conditions of the path constraint.

Theorem 4.1. *If a partial path constraint ϕ is unsatisfiable, then its full path constraint π is also unsatisfiable.*

Proof. Since ϕ is unsatisfiable, $\neg\phi$ is valid. Since ϕ is an overapproximation of π, $\pi \rightarrow \phi$ by definition. Therefore by modus tollens, $\neg\pi$ is valid. In other words, π is unsatisfiable. □

We now show the correctness of IPPC. We change only one line in Algorithm 2.5. Therefore, to prove the correctness of IPPC, we only need to ensure that for all path constraints, IPPC generates a test input that has the same property as the test input generated by the constraint solver. If such a test input exists, a constraint solver always generates a test input that satisfies the full path constraint.

Theorem 4.2. *(a) IPPC generates* null *whenever the constraint solver in Algorithm 4.1 at Line 11 generates* null*. (b) Otherwise IPPC always generates a test input that satisfies the full path constraint.*

Proof. Proof of (a) is trivial due to lines 1-4 of Algorithm 4.1. We can see from lines 5-10, that Algorithm 4.1 would not stop until the test input satisfies the full path constraint. Therefore proof of (b) is trivial as well. □

Theorem 4.3. *IPPC is correct and eventually terminates, i.e. IPPC is complete.*

Proof. IPPC is correct by Theorem 4.2. The second part of the proof is as follows. At any time, partial path constraint ϕ is either (a) unsatisfiable, or a test input i is generated. If a test input i is generated, either (b) i satisfies π, or (c) we add a new path condition of full path constraint to ϕ. If (a), IPPC returns *null* and terminates. If (b), IPPC returns i and terminates. If (c) happens $N - 1$ times, where N denotes the number of path conditions in π, $\phi \leftrightarrow \pi$ must hold. In other words, we build up the partial path constraint up to the full path constraint. In that case, the constraint solver must either generate *null* or generate a satisfying test input, therefore the algorithm terminates. □

5. EXPERIMENTS AND RESULTS

We implemented IPPC on top of CREST concolic testing framework. Then, we compared our strategy against the strategies implemented in CREST. Our implementation of IPPC and experimental results are available online [51].

We carried out the experiments on a virtual Linux guest with 1024MB memory and one CPU hosted by a MacBook Pro with an Intel Core i7 2.9 GHz GPU and 8GB Memory. We collected the following information for each experiment:

(i) Time elapsed: Time spent to test the UUT.
(ii) Total CS Calls: Total number of constraint solver calls made by the concolic tester.
(iii) Avg Const. Size: Average size of path constraints solved by the constraint solver.
(iv) Branch Coverage: Measurement of total branch coverage of UUT.

Concolic testing involves a degree of randomness due to the randomness of initial inputs. Therefore we performed 10 executions of each experiment and took average values of each measure.

We implemented five small benchmarks, *gcd*, *bsort*, *sqrt*, *prime*, and *factor* to conduct our initial experiments. *gcd* implements the binary greatest common divisor algorithm. We downloaded and modified the bubble sort algorithm *bsort*, which sorts a given array of integers [52]. *sqrt* takes the floor of the square root of a given integer. *prime* decides whether a given integer is prime or not. *factor* is an integer factorization algorithm. *replace* and *grep* are benchmarks that come with CREST framework and used in several research studies on concolic testing [6, 50, 53]. *ptokens* is the printtokens benchmark available at Software Infrastructure Repository (SIR) [54]. *ptokens* tokenizes the given string according to a grammar. For reasons described in Section

Table 5.1. List of Benchmarks

UUT	**KLOC**	**#vars**
gcd	0.05	2
bsort	0.05	30
sqrt	0.06	1
prime	0.1	1
factor	0.2	1
replace	0.5	20
ptokens	0.6	40
grep	15	10

5.3, our implementation of *gcd*, *prime*, and *factor* do not contain any bitwise masking or modulo operations. Instead, we decide divisibility via only subtraction and comparison operations. Also, *sqrt* does not use any floating point operations since CREST has no symbolic equivalent of floating point variables.

In our experiments, we used programs in various sizes. None of the given programs are too large in size so they can be tested in reasonable time without getting into scalability issues. In our experimental set, we argue that the structure of UUT is related to the performance of the testers that we use rather than the sizes of the programs. We observed that small programs such as *factor* can have very long runtimes (120 sec). We argue that IPPC will perform well not when the program is small or large, but when the program contains many infeasible paths. Although being small, *prime* and *factor* contain many infeasible paths. Although being large, *grep* and *replace* did not contain many infeasible paths.

We collected branch coverage of the generated testsuites via *gcov* utility. We share our branch coverage collection script in Appendix A.

5.1. Comparison of DFS and IPPC

In this section, we compare the standard concolic testing (DFS) implemented in CREST and the IPPC which we implemented on top of CREST.

Table 5.2. IPPC Speedup over DFS

UUT	Avg Const. Size Ratio (DFS / IPPC)	Speedup ($t_{\text{DFS}}/t_{\text{IPPC}}$)
replace	4.4	0.6x
bsort	20.8	0.79x
sqrt	21.3	1.25x
grep	31.8	0.83x
ptokens	48.4	1.7x
gcd	97.5	2.77x
prime	115.6	9.1x
factor	137.3	9.8x

We observe from Table 5.2 that there exists a correlation between Avg Const. Size of DFS / Avg Const. Size of IPPC and speedup of IPPC over DFS. When the gap between the constraint sizes of DFS and IPPC increases, the speedup of IPPC over DFS increases with the slight exception of *grep*.

In Table 5.1, we define number of iterations (# Itr) as the number of test inputs generated by the concolic tester. If there are no unsatisfiable constraints during testing, we make exactly one constraint call for each test input, therefore the ratio of *# Itr / # CS Calls* is 1. Figure 5.1 shows that whenever the number of constraint solver calls (Total CS Calls) of DFS is close to the maximum number of iterations (#Itr), DFS works faster. If DFS makes many more calls than the number of iterations, IPPC works faster. Normally, DFS is expected to call constraint solver exactly once for each iteration. DFS makes more than one constraint solver call for an iteration only if the generated path constraint for that iteration is infeasible. In that case, DFS changes the

path constraint and calls constraint solver repeatedly until a feasible path is found. We believe *factor* has the largest number of infeasibilities since DFS makes many constraint solver calls for few iterations. Figure 5.1 shows that the speedup of IPPC over DFS is above 5x when DFS generates four or more infeasible path constraints for each feasible path constraint (i.e. DFS makes more than five constraint solver calls for each iteration, #Itr/Total CS Calls of DFS < 0.2). Therefore, IPPC finds infeasibilities faster for all benchmarks.

Since both DFS and IPPC implement the same Depth-First Search strategy, they result in same branch coverage in all cases according to Table 5.3. IPPC modification results in a performance improvement in terms of time on 5 of the 8 benchmarks. Therefore, we conclude that IPPC clearly dominates its predecessor, DFS.

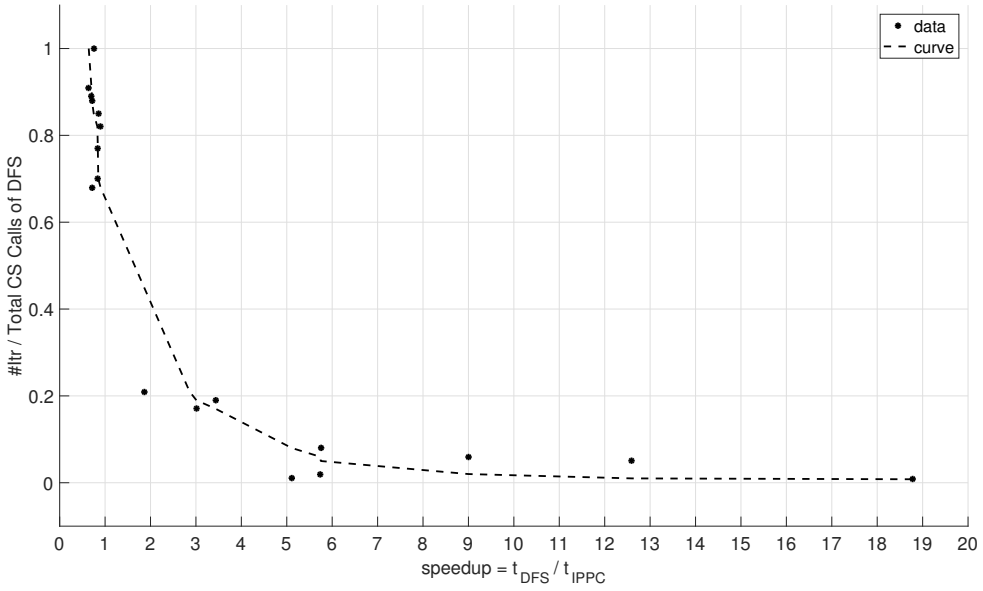

Figure 5.1. Relationship between Speedup and #Infeasible Constraints

5.2. IPPC Against Other Strategies

In this section, we discuss all the results given in Table 5.3. We've gathered these results by comparing IPPC to each strategy implemented in CREST. CREST

implements the standard concolic testing algorithm (DFS) and two different improvements, denoted by CFG and RND. We compared IPPC with these three techniques. Techniques we used in experiments are as follows.

(i) DFS: Standard concolic testing algorithm.
(ii) CFG: Control flow directed testing algorithm.
(iii) RND: Random branch testing heuristic on top of the standard concolic testing algorithm.
(iv) IPPC: Our Incremental Partial Path Constraint algorithm.

Brief explanations of CFG and RND algorithms are given in Section 3.3.

CFG method was not applicable to *grep* due to a bug in CREST, therefore we used *N/A* to denote the results we could not observe in Table 5.3. We used 1000, 5000, and 10000 as the maximum number of iterations (*#Itr*) in experiments. We decreased *#Itr* for *prime*, *factor*, *bsort*, and *grep*, since those units have few distinct execution traces. When there are few distinct execution traces and *#Itr* is too high, DFS and IPPC are able to stop before completing *#Itr* iterations, since they check if all execution traces are explored or not. However, CFG and RND have no such stopping condition and iterate *#Itr* times. So, if we kept *#Itr* high for *prime*, *factor*, *bsort*, and *grep*, it would unfairly result in bad runtimes for CFG and RND.

We show the best algorithms for each benchmark in Table 5.4 where *All Equal* means all techniques are equally well. The column denoted as *by Runtime First* compares techniques by runtime first, if techniques have similar runtimes, then compares their branch coverage. The last column compares techniques by coverage first. DFS does not perform well in both columns. In terms of runtimes RND is the best and IPPC is the second. In terms of coverage CFG and IPPC are the best. Hence, IPPC performs well in both categories.

Table 5.3. Experimental Results (The best results are highlighted)

UUT	**#Itr**	**Total CS Calls**				**Avg Const. Size**				**Branch Coverage (%)**				**Time Elapsed (sec)**			
		DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC
gcd	1000	5681	866	1105	11004	161.0	4.1	2.9	1.9	100	100	100	100	22.6	5.7	**3.9**	7.5
	5000	25801	4364	5653	64844	188.6	5.7	2.8	1.9	100	100	100	100	119.5	24.8	**21.6**	34.7
	10000	47367	8761	11287	120678	206.3	5.9	2.8	1.9	100	100	100	100	234.5	48.7	**45.7**	125.5
bsort	100	117	53	104	1346	145.7	148.5	88.9	10.3	100	100	100	100	1.1	**0.8**	0.9	1.3
	500	711	284	537	10306	308.9	165.3	115.5	12.7	100	100	100	100	7.6	5.0	**4.8**	9.2
	1000	1462	571	1063	25023	342.5	165.0	132.6	14.3	100	100	100	100	15.5	**8.7**	9.5	21.9
sqrt	1000	1010	980	1064	1006	17.2	2.3	1.8	1.0	94.4	94.4	94.4	94.4	4.3	4.5	4.3	**3.9**
	5000	5001	4894	5372	5006	24.0	2.3	1.8	1.0	94.4	94.4	94.4	94.4	30.1	23.1	22.2	**20.3**
	10000	10010	9820	10732	10014	22.7	2.3	1.8	1.0	94.4	94.4	94.4	94.4	45.9	42.0	42.2	**39.1**
prime	100	1275	92	111	132	103.3	7.5	4.8	1.2	**92.5**	85	62.5	**92.5**	2.3	0.5	0.5	**0.4**
	175	3946	165	212	253	186.6	8.0	4.0	1.2	**92.5**	90	70	**92.5**	10.1	1.0	1.1	**0.8**
	250	4166	236	287	1688	189.6	7.2	4.1	1.8	**92.5**	92.5	82.5	**92.5**	13.5	1.6	1.6	**1.5**
factor	50	2678	80	110	7693	222.1	2.9	4.0	1.9	**94.2**	**94.2**	76.9	**94.2**	8.6	**0.2**	**0.2**	1.5
	75	5734	111	145	25255	249.7	3.4	4.7	1.9	**94.2**	**94.2**	83.2	**94.2**	23.5	**0.3**	**0.3**	4.6
	100	11157	148	243	34218	310.9	2.8	16.0	1.9	**94.2**	**94.2**	89.4	**94.2**	120.3	0.6	**0.5**	6.4
replace	1000	1129	1024	1024	6847	16.6	25.9	31.7	3.8	85.2	**88.7**	85.2	85.2	**4.1**	4.5	**4.1**	5.8
	5000	5623	5193	5167	43695	21.5	28.4	32.4	4.8	88.7	**90.8**	88.7	88.7	**22.0**	23.8	22.4	31.4
	10000	10936	10270	10301	103590	24.5	28.0	33.0	5.6	88.7	**90.8**	90.8	88.7	**43.1**	45.5	44.6	66.9
ptokens	1000	1181	1355	1319	38415	591.0	136.8	102.2	29.8	**85.1**	79.8	79.8	**85.1**	30.0	9.4	**8.6**	18.5
	5000	6394	6976	6739	253829	1949.9	148.7	103.2	37.5	**85.1**	84.4	84.9	**85.1**	212.4	48.4	**47.7**	106.1
	10000	13394	13838	13510	432262	2241.6	163.7	99.7	30.5	**91.5**	89.6	88.9	**91.5**	321.7	106.8	**91.2**	210.8
grep	100	100	N/A	121	436	59.0	N/A	362.0	3.2	16.7	N/A	16.7	16.7	**0.3**	N/A	0.4	0.4
	220	269	N/A	272	1671	178.4	N/A	332.2	4.7	16.7	N/A	16.7	16.7	**0.9**	N/A	**0.9**	1.0
	340	439	N/A	451	3671	324.8	N/A	350.1	8.3	16.7	N/A	16.7	16.7	1.6	N/A	**1.5**	1.9

In all experiments, the largest path constraint produced by IPPC had a length of 157, whereas the largest path constraints of DFS, CFG, and RND had lengths of 2922, 1603, and 1391, respectively. We conclude that we eliminated the need for solving large path constraints to generate test inputs while keeping the runtimes fast and coverage high using IPPC.

We get exactly the same coverage results for both IPPC and DFS. We expect this since both algorithms perform a depth-first search. We argue that the coverage results being similar indicates that IPPC correctly does DFS on the UUT while improving the performance.

IPPC considers constraints that are 10x smaller than other techniques. We believe it is because that IPPC finds *infeasibilities* early, since most infeasibilities arise from a combination of few conditions in the constraint.

Table 5.4. The Best Concolic Testers

UUT	**by Runtime First**	**by Coverage First**
gcd	RND	RND
bsort	CFG	CFG
sqrt	IPPC	IPPC&DFS
prime	IPPC	IPPC&DFS
factor	RND	CFG
replace	DFS	CFG
ptokens	RND	IPPC
grep	All Equal	All Equal

5.3. Threats to Validity

We assume that the constraint solver is sound, i.e. the constraint solver can find an input vector whenever there exists an input vector which satisfies the path

constraint. In general, this assumption is not valid. We know that the constraint solver used in CREST, Yices [5, 12], does not find solutions for nonlinear path conditions (e.g. $x_2 x_1 < 12$). We also know that CREST may not be able to solve conditions involving modulo operation and bitwise masking [5]. All path conditions generated in our experiments are linear, in other words they can be written as $k \bowtie \sum_{i=1}^{N} c_i x_i$ where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $k \in \mathrm{R}$, $\forall i \in \{1, ..., N\}, c_i \in \mathrm{R}$ and N: total number of symbolic variables). All path conditions in our experiments are also free of modulo and bit masking operations. Hence, we safely assume that the constraint solver is sound in our environment. We also do not use any floating point arithmetic.

The UUT can have intermediary variables calculated from symbolic variables and therefore can have path conditions on those intermediary variables. All path conditions that CREST returns are on the initial symbolic variables. Therefore, even if all branch conditions in the code may seem trivial, CREST may fail to generate correct inputs if variables are nonlinear (e.g. multiplication of two symbolic variables).

We assume the UUT to be *sequential* and *deterministic*, i.e. if an input vector i produces an execution trace e, i will always produce e for this UUT. However, for example a process which depends on random numbers could violate this assumption. We carefully chose the experiments so that we never violate these basic assumptions. We assume the UUT is *terminating*, since if UUT halts, so does the tester as well. The test cases we chose and the test cases in previous work are all terminating. We report runtimes that we acquired from a virtual environment. We got similar results on an host machine as well.

It is possible that IPPC may learn partial path constraints up to a point that they become full path constraints. So in the worst case, a standard concolic tester is more efficient than IPPC. However, our experimental results show that we require only a small portion of the full path constraint to generate input vectors belonging to the same equivalence class, i.e. input vectors which generate same execution traces when

given to UUT.

We assume that CREST is correct and complete. Currently, CREST gives and exception and terminates if we try to test `vim` utility which comes with CREST itself.

6. CONCLUSIONS AND FUTURE WORK

In this section, first we discuss the validity of our research in Section 5.3. Then we give a summary of our study in Section 6.1 and discuss possible future work for our research in Section 6.2.

6.1. Conclusion

In this thesis, we propose an improvement to the constraint solving strategy of standard Constraint-Based Testing (CBT) methods, which we call Incremental Partial Path Constraints (IPPC). We describe how the previous CBT approaches handle constraints and how our approach aims to improve the current strategies through algorithms and examples. We proved the correctness and completeness of our approach and implemented it on top of a common concolic testing framework known as CREST. We conducted experiments on 8 benchmarks to evaluate our approach.

Our experiments show that we eliminate the need for solving large constraints to generate unit tests. We compared our design with other concolic testing algorithms in experiments. We observed that when there are many infeasible path constraints in the Unit Under Test (UUT), IPPC has more than 5x speedup over a standard concolic tester. We significantly reduce the number of path conditions required to generate test inputs and show that IPPC dominates other techniques in two of eight cases, has the best coverage levels in three of eight cases and dominates its predecessor DFS in five of eight cases. We show that it is possible to reach high branch coverage while decreasing the burden on the constraint solver. We also show that IPPC's performance improvement becomes significant when there are many infeasibilities in the UUT. We see the main disadvantage of IPPC is that it makes too many constraint solver calls. Therefore, IPPC may take a longer time to test if the UUT is simple and does not have many infeasibilities.

6.2. Future Work

We believe that our work motivates further research on constraint solving strategies involving partial path constraints. In this section, we describe important avenues to improve IPPC.

The performance of IPPC on nonlinear path constraints (i.e. path constraints that contain at least one nonlinear path condition) or on concurrent software is an important question. We believe IPPC will decrease the extra overhead of more complex constraints by finding infeasibilities faster.

IPPC is a modification that can be implemented on top of any Constraint-Based Testing (CBT) strategy. This allows us to implement our modification on top of other CBT tools.

It is possible to further improve IPPC by input caching, i.e. trying the previously generated inputs to avoid calling constraint solver. In common CBT approaches, we always use the full path constraints, therefore a previous input had no chance of satisfying a new full path constraint. However, previous test inputs may satisfy new partial path constraints. Since, solving constraints introduces more overhead, verifying with the previous inputs is an attractive approach.

Without increasing the constraint solver cost, we can initially start with a larger partial path constraint. A simple idea could be to generate a partial path constraint where no pair of path conditions have common symbolic variables. In this case, the constraint solver solves each path condition independently, avoiding exponential costs.

REFERENCES

1. "The Economic Impacts of Inadequate Infrastructure for Software Testing", 2002, http://www.nist.gov/director/planning/upload/report02-3.pdf, accessed at July 2016.

2. Abran, A., P. Bourque, R. Dupuis and J. W. Moore (Editors), *Guide to the Software Engineering Body of Knowledge - SWEBOK*, IEEE Press, Piscataway, NJ, USA, 2001.

3. Binder, R. V., *Testing Object-oriented Systems: Models, Patterns, and Tools*, Addison-Wesley Longman Publishing Co., Inc., 1999.

4. Venolia, G. D., R. DeLine and T. LaToza, *Software Development at Microsoft Observed*, Tech. Rep. MSR-TR-2005-140, Microsoft Research, October 2005, http://research.microsoft.com/apps/pubs/default.aspx?id=70227, accessed at July 2016.

5. Qu, X. and B. Robinson, "A Case Study of Concolic Testing Tools and Their Limitations", *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, 2011.

6. Burnim, J. and K. Sen, "Heuristics for Scalable Dynamic Test Generation", *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, 2008.

7. Boyer, R. S., B. Elspas and K. N. Levitt, "SELECT, a Formal System for Testing and Debugging Programs by Symbolic Execution", *Proceedings of the International Conference on Reliable Software*, 1975.

8. King, J. C., "Symbolic Execution and Program Testing", *Commun. ACM*, Vol. 19, No. 7, pp. 385–394, 1976.

9. Osterweil, L. J. and L. D. Fosdick, "Program Testing Techniques Using Simulated Execution", *Proceedings of the 4th Symposium on Simulation of Computer Systems*, ANSS '76, 1976.

10. Howden, W. E., "Symbolic Testing and the DISSECT Symbolic Evaluation System", *IEEE Trans. Softw. Eng.*, Vol. 3, No. 4, pp. 266–278, 1977.

11. De Moura, L. and N. Bjørner, "Z3: An Efficient SMT Solver", *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

12. Dutertre, B., "Yices 2.2", A. Biere and R. Bloem (Editors), *Computer Aided Verification*, Vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer International Publishing, 2014.

13. Sen, K., D. Marinov and G. Agha, "CUTE: A Concolic Unit Testing Engine for C", *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 5, pp. 263–272, 2005.

14. Cadar, C., D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs", *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

15. Seo, H. and S. Kim, "How We Get There: A Context-guided Search Strategy in Concolic Testing", *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

16. Anand, S., E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold and P. Mcminn, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation", *J. Syst. Softw.*, Vol. 86, No. 8, pp. 1978–2001, Aug. 2013.

17. Jaffar, J., V. Murali and J. A. Navas, "Boosting Concolic Testing via Interpolation", *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

18. Godefroid, P., "Compositional Dynamic Test Generation", *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, 2007.

19. Anand, S., P. Godefroid and N. Tillmann, "Demand-driven Compositional Symbolic Execution", *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

20. Cui, Z., W. Le, M. L. Soffa, L. Wang and X. Li, *MAGIC: Path-guided concolic testing*, Tech. rep., University of Virginia, Department of Computer Science, 2011.

21. Boonstoppel, P., C. Cadar and D. Engler, "RWset: Attacking path explosion in constraint-based test generation", *Proceedings of the International Conference on Tools And Algorithms for the Constructions and Analysis of Systems*, TACAS'08, 2008.

22. Krishnamoorthy, S., M. S. Hsiao and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs", *Proceedings of the 2010 19th IEEE Asian Test Symposium*, ATS '10, 2010.

23. Godefroid, P., N. Klarlund and K. Sen, "DART: Directed Automated Random

Testing", *SIGPLAN Not.*, Vol. 40, No. 6, pp. 213–223, 2005.

24. "MiniTest", `http://ruby-doc.org/stdlib-2.0.0/libdoc/minitest/rdoc/`, accessed at July 2016.

25. "JUnit", `https://sourceforge.net/projects/junit/`, accessed at July 2016.

26. "XUnit", `http://xunit.github.io`, accessed at July 2016.

27. Chen, T. Y., F.-C. Kuo, R. G. Merkel and T. H. Tse, "Adaptive Random Testing: The ART of Test Case Diversity", *J. Syst. Softw.*, Vol. 83, No. 1, pp. 60–66, Jan. 2010.

28. "Crashme", `http://crashme.codeplex.com`, accessed at July 2016.

29. Fraser, G. and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software", *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

30. Salvesen, K., J. P. Galeotti, F. Gross, G. Fraser and A. Zeller, "Using Dynamic Symbolic Execution to Generate Inputs in Search-based GUI Testing", *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, SBST '15, 2015.

31. DeMillo, R. A. and A. J. Offutt, "Constraint-Based Automatic Test Data Generation", *IEEE Trans. Softw. Eng.*, Vol. 17, No. 9, pp. 900–910, 1991.

32. Delahaye, M., B. Botella and A. Gotlieb, "Explanation-Based Generalization of Infeasible Path", *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, 2010.

33. Necula, G. C., S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs", *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, 2002.

34. Kähkönen, K., R. Kindermann, K. Heljanko and I. Niemelä, "Experimental Comparison of Concolic and Random Testing for Java Card Applets", *Proceedings of the 17th International SPIN Conference on Model Checking Software*, SPIN'10, 2010.

35. Sen, K. and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools", *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, 2006.

36. Sen, K., S. Kalasapur, T. Brutch and S. Gibbs, "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript", *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

37. Sen, K., *Scalable Automated Methods for Dynamic Program Analysis*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 2006, available at `http://srl.cs.berkeley.edu/~ksen/doku.php`.

38. Farzan, A., A. Holzer, N. Razavi and H. Veith, "Con2Colic Testing", *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

39. Eslamimehr, M. and J. Palsberg, "Race Directed Scheduling of Concurrent Programs", *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014.

40. Anand, S., M. Naik, M. J. Harrold and H. Yang, "Automated Concolic Testing of Smartphone Apps", *Proceedings of the ACM SIGSOFT 20th International Sympo-*

sium on the Foundations of Software Engineering, FSE '12, 2012.

41. Barner, S., C. Eisner, Z. Glazberg, D. Kroening and I. Rabinovitz, "ExpliSAT: Guiding SAT-based Software Verification with Explicit States", *Proceedings of the 2Nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, HVC'06, 2007.

42. Tsang, E. P. K., *Foundations of constraint satisfaction.*, Computation in cognitive science, Academic Press, 1993.

43. "CREST-z3", `https://github.com/heechul/crest-z3`, accessed at July 2016.

44. Nuzzo, P., A. Puggelli, S. A. Seshia and A. Sangiovanni-Vincentelli, "CalCS: SMT Solving for Non-linear Convex Constraints", *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, 2010.

45. Souza, M., M. Borges, M. d'Amorim and C. S. Păsăreanu, "CORAL: Solving Complex Constraints for Symbolic Pathfinder", *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, 2011.

46. Kiezun, A., V. Ganesh, P. J. Guo, P. Hooimeijer and M. D. Ernst, "HAMPI: A Solver for String Constraints", *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, 2009.

47. Sidiroglou-Douskos, S., E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim and M. C. Rinard, "Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement", *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pp. 473–486, 2015.

48. Bradley, A. R., "SAT-Based Model Checking without Unrolling", *Verification,*

Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, pp. 70–87, 2011.

49. Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-guided Abstraction Refinement for Symbolic Model Checking", *J. ACM*, Vol. 50, No. 5, pp. 752–794, 2003.

50. Dong, Y., M. Lin, K. Yu, Y. Zhou and Y. Chen, "Achieving High Branch Coverage with Fewer Paths", *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, COMPSACW '11, 2011.

51. "CREST-PPC", `http://bitbucket.org/yavuzkoroglu/crest-ppc`, accessed at July 2016.

52. "BubbleSort", `http://www.programmingsimplified.com/c/source-code/`, accessed at July 2016.

53. Seo, H. and S. Kim, "How We Get There: A Context-guided Search Strategy in Concolic Testing", *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

54. Do, H., S. G. Elbaum and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.", *Empirical Software Engineering: An International Journal*, Vol. 10, No. 4, pp. 405–435, 2005.

APPENDIX A: SCRIPT FOR COVERAGE MEASUREMENT

In this chapter, we present our script for coverage measurement. We first modified CREST to write the test inputs into a file. Then, we add the line `#include "coverage.h"` to the UUT, which redefines `CREST` macros to read values from the provided test suite. In the end, we execute the script in Figure A.1 to execute the UUT using *gcov* utility and extract branch coverage information.

```
#!/usr/bin/perl

open TestSuite, "<$ARGV[0]";

`gcc -Wall -fprofile-arcs -ftest-coverage
-g $ARGV[1]_cov.c -o $ARGV[1]_cov.out`;

$count = 0;
while (<TestSuite>) {
    open TempInputs, ">temp_inputs.txt";

    $count++;
    print "$_\n";
    print TempInputs "$_\n";

    close TempInputs;

    `$ARGV[1]_cov.out $_`;
}

print "By using $count test cases;\n";
print `gcov -b $ARGV[1]_cov.c | grep Taken`;

close TestSuite;
```

Figure A.1. bcov.pl: Branch Coverage Measurement Script.

APPENDIX B: SOURCE CODES FOR SMALL BENCHMARKS

In this chapter, we present source codes of the benchmarks we used in out experiments. We omitted large benchmarks such as *replace* and *grep*. In each benchmark, notice that the symbolic variables are specified using `CREST` macros. We provide our full experimental set with additional benchmarks and instructions to execute online [51]

```
/* Bubble sort code */
#include <stdio.h>
#include <crest.h>

int main()
{
  unsigned char n = 30;
  unsigned char array[30], c, d, swap;

  CREST_unsigned_char(array[0]);
  CREST_unsigned_char(array[1]);
  CREST_unsigned_char(array[2]);
  CREST_unsigned_char(array[3]);
  CREST_unsigned_char(array[4]);
  CREST_unsigned_char(array[5]);
  CREST_unsigned_char(array[6]);
  CREST_unsigned_char(array[7]);
  CREST_unsigned_char(array[8]);
  CREST_unsigned_char(array[9]);
  CREST_unsigned_char(array[10]);
  CREST_unsigned_char(array[11]);
  CREST_unsigned_char(array[12]);
  CREST_unsigned_char(array[13]);
  CREST_unsigned_char(array[14]);
  CREST_unsigned_char(array[15]);
  CREST_unsigned_char(array[16]);
  CREST_unsigned_char(array[17]);
  CREST_unsigned_char(array[18]);
  CREST_unsigned_char(array[19]);
  CREST_unsigned_char(array[20]);
```

Figure B.1. Source Code of *bsort* Benchmark

```
  CREST_unsigned_char(array[21]);
  CREST_unsigned_char(array[22]);
  CREST_unsigned_char(array[23]);
  CREST_unsigned_char(array[24]);
  CREST_unsigned_char(array[25]);
  CREST_unsigned_char(array[26]);
  CREST_unsigned_char(array[27]);
  CREST_unsigned_char(array[28]);
  CREST_unsigned_char(array[29]);

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use < */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
     printf("%d ", array[c]);
  printf("\n");
  return 0;
}
```

Figure B.1. Source Code of *bsort* Benchmark (cont.)

```
#include <stdio.h>
#include <crest.h>

#define FALSE (0)
#define TRUE  (!(0))
```

Figure B.2. Source Code of *sqrt* Benchmark

```
typedef unsigned long long N;
typedef int bool;

N logfloor(N x) {
  int i;
  for (i = 0; x != 0; i++, x >>= 1);

  return i - 1;
}

int sqrtTest(N n, N nSq) {
     if (n * n > nSq)
       return 1;
     else if ((n+1) * (n+1) <= nSq)
       return -1;
     else
       return 0;
}

N sqrtfloor(N x) {
  if (x == 0 || x == 1)
     return x;

  N k = logfloor(x);
  N l, h;
  if (k & 1) {
     l = 1 << ((k - 1) >> 1);
     h = l << 1;
  } else {
    h = 1 << ((k >> 1) - 1);
    l = h << 1;
    h *= 3;
  }

  if (!sqrtTest(l, x)) {
     return l;
  }
  N m;
  for (m = (l + h) >> 1;;
       m = (l + h) >> 1)
```

Figure B.2. Source Code of *sqrt* Benchmark (cont.)

```
  {
    switch (sqrtTest(m, x)) {
      case 1:
        h = m;
        break;
      case 0:
        return m;
      case -1:
        l = m;
    }
  }
}

int main() {
  unsigned int x;
  CREST_unsigned_int(x);

  printf("floor(sqrt(%u)) = %u", x,
    sqrtfloor(x));

  return 0;
}
```

Figure B.2. Source Code of *sqrt* Benchmark (cont.)

```
#include <crest.h>
#include <stdio.h>

#define FALSE (0)
#define TRUE  (!(0))

typedef unsigned long long N;
typedef int bool;

bool divides(N x, N y) {
  while (y > 0) {
    if (x > y) {
      return FALSE;
    } else {
      y -= x;
    }
  }
```

Figure B.3. Source Code of *prime* Benchmark

```
  return TRUE;
}

N logfloor(N x) {
  int i;
  for (i = 0; x != 0; i++, x >>= 1);

  return i - 1;
}

int sqrtTest(N n, N nSq) {
     if (n * n > nSq)
       return 1;
     else if ((n+1) * (n+1) <= nSq)
       return -1;
     else
       return 0;
}

N sqrtfloor(N x) {
  if (x == 0 || x == 1)
     return x;

  N k = logfloor(x);
  N l, h;
  if (k & 1) {
     l = 1 << ((k - 1) >> 1);
     h = l << 1;
  } else {
    h = 1 << ((k >> 1) - 1);
    l = h << 1;
    h *= 3;
  }

  if (!sqrtTest(l, x)) {
     return l;
  }
  N m;
  for (m = (l + h) >> 1;;
       m = (l + h) >> 1)
  {
    switch (sqrtTest(m, x)) {
```

Figure B.3. Source Code of *prime* Benchmark (cont.)

```
      case 1:
        h = m;
        break;
      case 0:
        return m;
      case -1:
        l = m;
    }
  }
}

bool isPrime(N x) {
  if (x == 2 || x == 3)
    return TRUE;
  else if (x == 1 || x == 0
       || divides(2, x) || divides(3, x))
    return FALSE;

  N sqrt_x = sqrtfloor(x);
  N i;
  for (i = 5; i <= sqrt_x; i+=6) {
     if (divides(i, x) || divides(i + 2, x))
       return FALSE;
  }
  return TRUE;
}

int main() {
  unsigned char x;
  CREST_unsigned_char(x);
  printf("isPrime(%u) = %d\n", x, isPrime(x));
  return 0;
}
```

Figure B.3. Source Code of *prime* Benchmark (cont.)

```
#include <stdio.h>
#include <crest.h>

#define FALSE (0)
#define TRUE  (!(0))
```

Figure B.4. Source Code of *factor* Benchmark

```
typedef unsigned long long N;
typedef int bool;

int divides(N x, N y) {
  while (y > 0) {
    if (x > y) {
      return 0;
    } else {
      y = y - x;
    }
  }

  return 1;
}

N divide(N x, N y) {
  N result = 0;
  while (x >= y) {
    x = x - y;
    result++;
  }
  return result;
}

N logfloor(N x) {
  int i;
  for (i = 0; x != 0; i++, x >>= 1);

  return i - 1;
}

int sqrtTest(N n, N nSq) {
     if (n * n > nSq)
       return 1;
     else if ((n+1) * (n+1) <= nSq)
       return -1;
     else
       return 0;
}

N sqrtfloor(N x) {
```

Figure B.4. Source Code of *factor* Benchmark (cont.)

```
N sqrtfloor(N x) {
  if (x == 0 || x == 1)
     return x;

  N k = logfloor(x);
  N l, h;
  if (k & 1) {
     l = 1 << ((k - 1) >> 1);
     h = l << 1;
  } else {
    h = 1 << ((k >> 1) - 1);
    l = h << 1;
    h *= 3;
  }

  if (!sqrtTest(l, x)) {
     return l;
  }
  N m;
  for (m = (l + h) >> 1;;
       m = (l + h) >> 1)
  {
    switch (sqrtTest(m, x)) {
      case 1:
        h = m;
        break;
      case 0:
        return m;
      case -1:
        l = m;
    }
  }
}

bool isPrime(N x) {
  if (x == 2 || x == 3)
    return TRUE;
  else if (x == 1 || x == 0
       || divides(2, x) || divides(3, x))
    return FALSE;

  N sqrt_x = sqrtfloor(x);
  N i;
```

Figure B.4. Source Code of *factor* Benchmark (cont.)

```
  N i;
  for (i = 5; i <= sqrt_x; i+=6) {
     if (divides(i, x) || divides(i + 2, x))
       return FALSE;
  }
  return TRUE;
}

int main() {
  N x;
  CREST_unsigned_char(x);

  if (x > 250)
    return 1;

  if (isPrime(x)) {
    return 1;
  }
  N factor, sqrt_x;
  sqrt_x = sqrtfloor(x);
  for (factor = 2; factor <= sqrt_x; factor++) {
    if (isPrime(factor)) {
      while (divides(factor, x)) {
        printf("%llu ", factor);
        x = divide(x, factor);
      }
    }
  }
  if (x == 1)
    printf("\n");
  else
    printf("%llu\n", x);

  return 0;
}
```

Figure B.4. Source Code of *factor* Benchmark (cont.)