Conflict-Driven Symbolic Execution

How to Learn to Get Better

by

Celina Gomes do Val

B.Sc., Universidade Federal de Minas Gerais, 2009
M.Sc., Universidade Federal de Minas Gerais, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2014

© Celina Gomes do Val 2014

Abstract

Due to software complexity, manual and automatic testing are not enough to guarantee the correct behavior of software. One alternative to this limitation is known as Symbolic Execution.

Symbolic Execution is a formal verification method that simulates software execution using symbolic values instead of concrete ones. The execution starts with all input variables unconstrained, and assignments that use any input variable are encoded as logical expressions. Whenever a branch is reached, the symbolic execution engine checks which values the branch condition can assume. If more than one valid evaluation is possible, the execution forks, and a new process is created for each possibility.

In cases where the program execution is finite, symbolic execution is complete, and potentially executes every reachable program path. However, the number of paths is exponential in the number of branches in the program, and this approach suffers from a problem know as path explosion.

This thesis presents a novel algorithm that can dynamically reduce the number of paths explored during symbolic execution in order to prove a given set of properties. The algorithm is capable of learning from conflicts detected while symbolically executing a path.

I have named this algorithm Conflict-Driven Symbolic Execution (CDSE), since it was inspired by the conflict-driven clause learning (CDCL) insights introduced by modern boolean satisfiability solvers. The proposed algorithm takes advantage of two features responsible for the success of CDCL solvers: conflict analysis and non-chronological backtracking. In a nutshell, CDSE prunes the search space every time a certain branch is proven infeasible by learning the reason why there is a conflict.

In order to assess the proposed algorithm, this thesis presents a proof-of-

concept CDSE tool named Kite, and compares its performance to the state-of-the-art symbolic execution tool Klee [10]. The results are encouraging, and present practical evidence that conflict-driven symbolic execution can perform better than regular symbolic execution.

Preface

The work presented in this thesis was primarily done by myself, except for the encoding of the search constraints (Section 3.2.1), which I designed in collaboration with Sam Bayless.

None of the text of this thesis was taken from previously published articles.

Table of Contents

Abstract . . . ii

Preface . . . iv

Table of Contents . . . v

List of Tables . . . viii

List of Figures . . . x

Acknowledgments . . . xii

Dedication . . . xiii

1 Introduction . . . 1
- 1.1 Motivating Example . . . 4
- 1.2 Text Organization . . . 6

2 Background . . . 8
- 2.1 Notations and Definitions . . . 8
- 2.2 Symbolic Execution . . . 11
  - 2.2.1 Solving Strategies . . . 14
  - 2.2.2 Query Optimizations . . . 16
- 2.3 SAT Solvers . . . 18
  - 2.3.1 Definitions . . . 19
  - 2.3.2 DPLL SAT Solvers . . . 20
  - 2.3.3 CDCL SAT Solvers . . . 22
- 2.4 CDCL Applications . . . 26

3 Conflict-Driven Symbolic Execution . . . 28
3.1 Overview . . . 29
3.2 Learning Scheme . . . 32
3.2.1 Search Constraints . . . 32
3.2.2 Conflict Analysis . . . 34
3.3 CDSE Algorithm . . . 36
3.3.1 Preprocess . . . 38
3.3.2 Decide . . . 39
3.3.3 Propagate . . . 39
3.3.4 Conflict Analysis . . . 40
3.3.5 Learn and Backtrack . . . 41
3.4 Multiple Properties . . . 41
3.5 Loop Handling . . . 42

4 Kite: A Conflict-Driven Symbolic Execution Tool . . . 45
4.1 Kite's Architecture . . . 45
4.2 Preprocessor . . . 47
4.2.1 CFG Construction . . . 48
4.2.2 fCFG Construction . . . 49
4.3 Instruction Interpreter . . . 50
4.4 Constraint Solver . . . 53
4.5 Decision Engine . . . 55
4.6 Known Limitations . . . 57

5 Results . . . 60
5.1 Testing Environment . . . 60
5.2 Overall Evaluation . . . 62
5.3 Evaluation of the Decision Engines . . . 65

6 Related Work . . . 70
6.1 Interpolation-Based Learning Schemes . . . 72
6.2 Clause Learning Schemes . . . 75

7 Conclusion 79
7.1 Contributions 79
7.2 Future Work 81

Bibliography 84

Appendix

A Extended Results 91

List of Tables

5.1 Table showing the execution time (in seconds) spent by Klee, CBMC and Kite to prove that each test case respects their assertions. Since every tool spent more than $10s$ to solve the 7 instances marked with a '†', they will be classified as hard, in contrast to the other instances, classified as easy, that could be solved in less than $1s$ by at least one tool. 63

5.2 Table showing the execution time (in seconds) spent by Klee, CBMC and Kite to find a bug in each test case. 65

5.3 Total time spent (in seconds) by each tool to solve the benchmarks. Overall, Kite had the best performance, and CBMC had the worst performance. 66

5.4 Table showing the effect of different decision engines on Kite's execution time (in seconds) to verify the safe instances. . . . 68

5.5 Table showing the effect of different decision engines on Kite's execution time (in seconds) to find a property violation in the unsafe instances. In the instance *unsafe_07*, two engines did not finish before the $200s$ timeout. Thus, the total time reported for these engines are not precise. 69

A.1 Renaming of Safe Instances. 91

A.2 Renaming of Unsafe Instances. 92

A.3 The execution time (in seconds) spent by the two versions of Klee to solve the safe instances. Klee_f corresponds to the forked version, while Klee_l corresponds to the latest one. . . . 93

A.4 The execution time (in seconds) spent by the two versions of Klee to solve the unsafe instances. Klee_f corresponds to the forked version, while Klee_l corresponds to the latest one. . . . 94
A.5 Table showing the number of instructions that were symbolically executed by all versions of Klee and Kite to solve the safe instances. In instances `safe_05` and `safe_06`, Kite was capable of proving property's correctness just by applying the code optimization passes, and the slicing algorithm. 95
A.6 Table showing the number of instructions that were symbolically executed by all versions of Klee and Kite to solve the unsafe instances. 96

List of Figures

1.1 The execution tree for Listing 1.1, where P represents the property being proved. 6

2.1 The control flow graph and the execution tree for the program described in Listing 2.1. (a) The CFG represents every basic block in the program, along with all possible transitions between them. (b) The execution tree represents every feasible execution path. It can be obtained by unwinding the CFG, and checking which paths are feasible. 11

3.1 The control flow graph and the execution tree for the program described in Listing 1.1. (a) In the CFG, each branch b and each basic block B has a unique ID. (b) The execution tree represents the 8 paths in the CFG that would lead the program to check the assertion. The execution tree edges represent only conditional branches; thus, one node may represent the execution of more than one basic block. 30

3.2 An example of a satisfying assignment to the fCFG that no longer guarantees the existence of a complete path in a CFG due to the existence of loops. (a) Represents a program CFG with two loops. (b) Represents a satisfying assignment to the corresponding fCFG. In this representation, each CFG element's identification is replaced by the value assigned to the corresponding fCFG variable. Any complete path in the given CFG should include b_3; however, a loop in the CFG allows a node to succeed its predecessor, and rules 4 and 5 can be satisfied without guaranteeing that b_3 must be taken in order to reach B_D. 43

4.1 Kite's main modules, their composition and communication. The preprocessor module provides information to the instruction interpreter and the decision engine. The instruction interpreter symbolically executes the code guided by the decision engine. The interpreter and the constraint solver share information about path condition feasibility. 46

5.1 Number of instructions symbolically executed by Klee and Kite on safe and unsafe instances. As expected, Kite symbolically executed fewer instructions in the majority of the tests, even in unsafe instances where Klee has better performance. 67

6.1 The control flow graph for the program described in Listing 1.1, where basic block B_H represents the assertion failure. . . 72

Acknowledgments

First, I would like to express my gratitude towards my supervisor, Professor Alan J. Hu, for his guidance and his enthusiastic encouragement throughout this work. I am also grateful for all the remarkable advice given by Professor Mark Greenstreet.

I wish to acknowledge financial support from Intel Corporation and from the Semiconductor Research Corporation, which were crucial to this project's development.

Furthermore, I would like to thank all my fellow colleagues in the Integrated Systems Design Laboratory, who always made this journey so delightful. I am particularly grateful for Sam Bayless's valuable insights.

I cannot overstate how much I appreciate the support I received from my friends and family, especially my parents. Despite the distance, I have always been blessed by their love.

Finally, I would like to thank Ian for his patience, his kindness, and his help.

To my parents

Chapter 1

Introduction

Software complexity is increasing, and current testing techniques are not enough to guarantee correct functionality. Bugs, such as the one responsible for NASA's Mars climate orbiter crash [1], can be very costly [37]. In 2002, Tassey [47] estimated that the cost of software bugs in the U.S. economy is $59.5 billion dollars per year.

Moreover, failures in safety-critical software not only have impact in economic terms, but also can endanger lives. For example, a software bug in the Therac-25 radiation therapy machine controller caused at least five patients deaths due to excessive X-rays administration in the 1980s [33].

In order to avoid bugs from reaching the market, verification has become a crucial stage of software development.

Software testing is by far the most used technique for software verification due to its simplicity and scalability. The code is executed with different inputs in order to assure quality and look for bugs. The inputs can be generated randomly, or to try to imitate expected usage scenarios. In addition, coverage directed techniques can be used to generate inputs that aim at increasing test-case coverage, e.g., these techniques can generate inputs that force the program to execute specific lines of code [10].

Nevertheless, testing is not enough to ensure correct behavior. For instance, a test suite for any program with only one input of 32 bits would require 2^{32} different test inputs to be complete. Assuming one program from this set executes in 10 ms, a complete test suite would take more than 80 years to finish.

Complementary to automated testing, static analysis techniques have become part of the software verification process. Any analysis performed without concretely executing the program can be classified as static. For

example, modern compilers use static analysis to detect irregular and unsafe constructions before generating the executable code. These detection algorithms trigger compilation warnings or errors whenever they identify code that appears buggy.

Recently, static analysis has also been used to build automatic formal program verification tools. In contrast to previous techniques, formal program verification proves some correctness properties instead of just bug finding. Two predominant techniques used for the construction of formal program verification tools are Model Checking and Symbolic Execution:

Model Checking was introduced by Clarke and Emerson [12] and by Queille and Sifakis [42], independently. It is a technique for verifying temporal logic properties of a given transition system (including software) by exploring its state space. In its original form, model checking implementations used explicit representations of the state transition graph. However, the number of states grows exponentially with the number of system variables — a problem known as state explosion.

Symbolic state representation was later introduced to ameliorate the state explosion problem [8]. In particular, the employment of model checking spread widely in industry after the introduction of Bounded Model Checking (BMC) [6] and the adoption of satisfiability solvers (SAT solvers). Bounded model checking modifies model checking to encode only a finite sequence of state transitions that reaches certain states of interest as a Boolean formula. SAT solvers are used to determine whether the formula is satisfiable or not, which proves if the states of interest are reachable or not, respectively[1]. The generated boolean formula is linear in the length of the sequence of state transitions, and determining if this formula is satisfiable or not is an NP-Complete problem. In spite of the high worst-case complexity, BMC has scaled to larger problems than other approaches, largely because it benefits directly form ongoing advances in SAT solvers.

[1] There exists different model checking techniques, known as Unbounded Model Checking, than can often prove properties of infinite execution sequences. However, they go beyond the scope of this thesis.

Symbolic Execution was introduced by King [28]. It is a method to simulate software execution using symbolic values instead of concrete ones[2]. While simulating the program with symbolic values, branch statements might have more than one feasible evaluation. When this is the case, the symbolic execution engine must consider every feasible possibility; consequently, it forks. Every time the execution forks, a new constraint derived from the branch condition will be added to each simulation process according to the direction taken. These constraints are accumulated during the execution, and they comprise the path condition. Symbolic execution employs constraint solvers to determine the possible values of a branch statement given a certain path condition. In cases where the program execution is finite, symbolic execution is complete, and potentially executes every reachable program path. Symbolic execution tools may also bound program loops to guarantee the analysis termination, but like BMC the analysis is restricted to paths that respect the loop bound. In both scenarios where the execution is finite, the number of paths is still exponential in the number of branches in the program; consequently, this approach suffers from path explosion.

Both methods represent state transitions in programs as propositional logic formulas, and use constraint solvers to check their satisfiability[3]. Bounded model checking, however, typically builds one single formula to represent the entire program, while symbolic execution builds different formulas for each execution path.

This difference gives two advantages to symbolic execution. First, symbolic execution is usually more efficient for finding deeper bugs, because its

[2]In this thesis, symbolic execution refers to the path-by-path exploration with symbolic values as it was first presented by [28]. Such a terminology is also used by many other authors [5, 25, 48]. In contrast, Symbolic execution has been used as a broader term by [3, 29], to include techniques this thesis classifies as model checking.

[3]Even though this chapter emphasizes the usage of SAT solvers as constraint solvers, other solvers can be used for model checking and symbolic execution, such as Satisfiability Modulo Theories (SMT) solvers. The emphasis on SAT solvers is because most algorithmic advances were first applied to SAT solvers.

formula size doesn't grow as fast as model checking to reach a certain depth in the program. Thus, if lucky, the symbolic execution might find a bug at a depth where the model checking formula is too big and complex to be solved in a reasonable time.

The second advantage is that symbolic execution can be used to partially verify software. Whenever symbolic execution fails to prove whether one or more properties holds in a certain time or memory constraint, it can return which paths were completely executed. Thus, symbolic execution can guarantee these paths's correctness regarding the properties tested. Furthermore, the path condition can be used to generate test inputs or constraints to guide automated tests. In contrast, when a model checker does not reach a conclusion, the model checker doesn't produce any intermediate result.

Symbolic execution is at a disadvantage compared to model checking when it tries to prove that a certain property holds. Not only are symbolic execution decision heuristics not as optimized as SAT solvers's, but, more importantly, SAT solvers can learn from the decisions previously made, whereas most symbolic execution engines can't.

Ideally, any symbolic execution engine would also take advantage of its previous decisions in order to reduce the verification effort. After executing a path, the engine can learn facts regarding the path's fragments. Furthermore, these facts can be used to predict whether other paths will satisfy the properties tested.

In this thesis, I present an efficient technique that can reason about symbolically executed paths and prune the solution search space. This technique is based on the learning scheme used in Conflict-Driven Clause Learning (CDCL) SAT (Satisfiability problem) solvers, which are known for the success and scalability of SAT solvers.

1.1 Motivating Example

Before providing the proposed learning technique, this section presents the intuition behind the learning scheme through an example. I illustrate how to learn facts from symbolically executed paths, and to prune a large num-

```
1   int main () {
2
3       symbolic int x, y;
4
5       if (x < 0)
6           x = -x;
7       if (y < 0)
8           y = -y;
9
10      int result = x - y;
11
12      if (x < y)
13          result = y - x;
14
15      assert (result >= 0);
16
17  }
```

Listing 1.1: An example to illustrate that not all paths are needed to prove a property. In this example, the property to be proven is an assertion statement embedded in the code. (In this example, this assertion always holds)

ber of paths.

In Listing 1.1, the property to be verified is encoded as an assertion statement, which should hold for every possible execution. There are 8 distinct execution paths that reach the assertion, as shown by the execution tree in Figure 1.1. Most symbolic execution tools, like Klee [10] and Java PathFinder [48], would simulate all 8 paths before returning that the given property always holds. However, it is possible to prove that the assertion always holds by executing only 2 paths.

For example, if the first path to be executed is the one where all `if` statements ($x < 0$, $y < 0$ and $x < y$) are TRUE (red path), symbolic execution will reach the conclusion that `result` is always greater or equal to 0. Such a conclusion can be derived from the constraint '$x < y$' and the assignment in line 13 alone; thus, the other constraints and statements are irrelevant to prove the property in this path. Once that fact is learned, every other path that includes the true branch of the '`if` $x < y$' statement doesn't

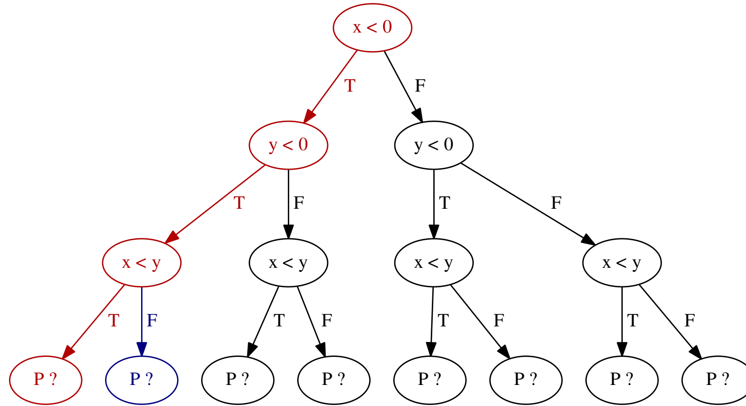


Figure 1.1: The execution tree for Listing 1.1, where P represents the property being proved.

have to be executed, because these paths will include the two constraints that are enough to guarantee the property tested.

Now, suppose the second path to be executed is the one where the first two `if` statements ($x < 0$ and $y < 0$) are still TRUE, but the last one, $x < y$, is FALSE (the blue edge). The `result` definition comes from line 10 and the path condition includes the constraint $\neg(x < y)$, or equivalently $x >= y$. Together, the definition and the constraint imply that `result` is always greater or equal to 0. This new implication is enough to prune every path where the last branch, $x < y$, is FALSE.

Since every path that reaches the assertion will contain either the constraint $x < y$ and $result = y - x$ definition *or* the constraint $x >= y$ and $result = x - y$ definition, those two paths are enough to prove the property $result >= 0$.

1.2 Text Organization

This thesis introduces a technique to systematically learn facts (like the ones in the preceeding example) from each symbolically executed path. These

facts can be used to predict the behavior of non-executed paths. I also present a proof-of-concept tool named Kite, which is an implementation of the learning technique built on the top of the symbolic execution tool Klee [10].

This thesis is organized as follows: Chapter 2 provides background on Symbolic Execution and on the CDCL algorithm. Chapter 3 explains how these two approaches can be combined to build a more efficient software verification tool. Chapter 4 introduces Kite, as well as implementation details and optimizations that make such an approach more scalable. Chapter 5 presents the tests and the results obtained from executing Kite, as well as a comparison with other state-of-the-art verification tools. Chapter 6 presents the recent works that are related to the proposed method. Finally, Chapter 7 discusses this thesis's main contributions, its limitations, and future work.

Chapter 2

Background

This chapter presents the background on two important techniques that will be the basis of this work. The first is symbolic execution, which is a software analysis technique usually used to generate tests, as well as to find run-time errors and assertion failures. The second is Conflict-Driven Conflict Learning SAT solving, which is a powerful constraint solver framework that has been used as an underlying engine for symbolic execution, model checking and many other industrial applications.

Symbolic execution engines and SAT solvers are both search tools. Symbolic execution engines search for an assignment to program variables that will lead the execution along a certain path, while SAT solvers search for an assignment to boolean variables that will lead a formula to evaluate to true.

However, there is one main difference between these search mechanisms that is relevant for this thesis: modern SAT solvers learn from failed search attempts, while symbolic execution tools usually don't — even though, as we will see, they can.

2.1 Notations and Definitions

This section establishes the notation followed throughout this thesis. In order to distinguish SAT instance variables from program variables, this thesis will represent the former with a lower case v, and the latter with Greek letters. Specifically, the program variables will be represented with σ. Additionally, boolean expressions will be represented with β, program properties will be represented with a ρ, and property violations will be represented with a μ.

Additionally, this thesis assumes the program under verification follows

the imperative or procedural programming paradigm, where the program execution follows a sequence of commands — an *execution path* — that change the *program state*. A program state is defined as the collection of program variables and their associated values, together with the control location, which is known as the *program counter*. Each instruction (command or program statement) either modifies program variable values, or the control location, i.e., each instruction is either an assignment or a branch (conditional or unconditional). For simplicity, sequences of consecutive instructions with a single entry and a single exit point, and no branches, are grouped as *basic blocks*.

Without loss of generality[4], this thesis considers programs that contain one method only — all method calls are inlined — and all conditional branches are represented by `if`-statements with the following format:

`if` β `then` [`goto TRUE BB`] `else` [`goto FALSE BB`]

where if the given condition β is true, the next basic block (BB) to be executed is the `TRUE BB`; otherwise, the next one is the `FALSE BB`. The `else` statement is optional, and when omitted, the `FALSE BB` is equivalent to the basic block that follows the if statement.

The flow of control within a program can be captured as a directed graph, known as the *control flow graph* (CFG). The control flow graph represents the sequence of commands and possible transitions between them. Thus, in a program's CFG, the nodes represent basic blocks and the edges indicate possible control flow transitions between basic blocks. Every branch instruction destination associated with the instruction location originates at an edge in the control flow graph. In this thesis, basic block identification starts with a B, and branch identification starts with a b. Consequently, the same notation is used to represent the CFG nodes and edges, B and b respectively.

Consider the small example in Listing 2.1, and its CFG depicted in Figure 2.1(a). The CFG has 3 nodes — representing basic blocks B_A, B_B and B_C

[4]The structured program theorem [7] states that any program can be described using three transition techniques: sequence, selection, and repetition (or loop).

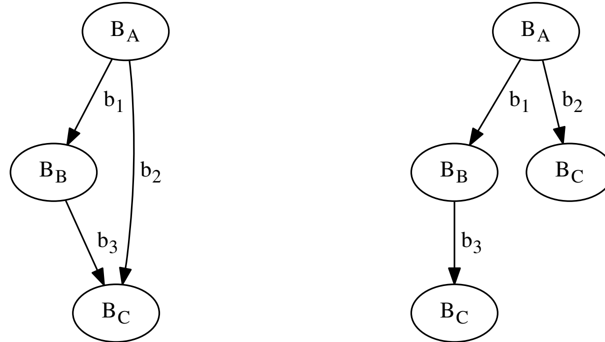
```
1   int main (int argc, char** argv) {
2       symbolic int a, b;
3       int ret = a + b;
4       if (argc > 1)
5           ret = ret / 2;
6       return ret;
7   }
```

Listing 2.1: A small example in C to illustrate how basic blocks are defined. The program has 3 basic blocks, the entry basic block B_A goes from line 1 to 4, while B_B contains only the statement from line 5 and B_C is composed by the return statement in line 6.

— and 3 edges — representing branches $b_0 = (B_A, B_B)$, $b_1 = (B_A, B_C)$ and $b_2 = (B_B, B_C)$. The `if`-statement is a conditional branch statement that connects B_A to either B_B or B_C. The transition from B_B to B_C is implicit, because the unconditional branch statements at the end of a basic block can be omitted in some languages like C.

Although every feasible execution path in a program is represented in the CFG, not every path represented in the CFG is necessarily *feasible*. A path is only considered feasible if there exists at least one set of input values that leads the program execution through the path's command sequence. If no such set of values exists, the path is considered *infeasible*. This thesis will use the the term *possible* execution paths for those that may or may not be feasible (i.e., it hasn't been proven yet whether the path is feasible or not).

The set of all feasible execution paths in a program can be represented as a tree, known as the *execution tree* (see example shown in Figure 2.1(b)). Thus, the subset of paths represented in the execution tree can be obtained by unwinding the CFG, and checking their feasibility. Due to loops in a program, and in the CFG, an execution tree can be infinite.



(a) The control flow graph. (b) The execution tree.

Figure 2.1: The control flow graph and the execution tree for the program described in Listing 2.1. (a) The CFG represents every basic block in the program, along with all possible transitions between them. (b) The execution tree represents every feasible execution path. It can be obtained by unwinding the CFG, and checking which paths are feasible.

2.2 Symbolic Execution

In a nutshell, *symbolic execution* is a program analysis method that simulates program execution with symbolic values instead of concrete ones. The symbolic execution engine treats each input variable as a symbolic value; hence, the engine can analyze the program behavior under every combination of input values. The usage of symbolic values provides a safe and sound method to find feasible execution paths, as well as to establish under which conditions they are executed. If the program doesn't have an infinite execution tree, then symbolic execution is also a complete method.

In order to handle symbolic values, the program's execution semantics must be extended. First, variables assignments are extended, and a variable can either have a concrete value or a *symbolic expression*. The symbolic expression is defined as an expression that represents a function over the symbolic variables. The symbolic execution engine produces a symbolic ex-

pression while evaluating a program's data manipulation instruction. The evaluation expands each variable used and translates the instruction operation into an arithmetic operator.

The semantics of `if`-statement instructions is also extended, and the `if`-condition might have more than one possible evaluation in the current execution, i.e., the `if`-condition may be true, false or undefined. In the last case, the engine explores both possibilities separately. To do so, the execution forks into two distinct executions, named *processes*: one that follows the `true`-branch, and another one that follows the `false`-branch. Each process assumes that the `if`-condition is either true or false according to the direction taken.

The `if`-statement evaluation depends on the previous assumptions made by the current process. Therefore, the program state is extended in order to maintain the set of conditions that have been assumed true during a process execution. This set is known as the *path condition*, or just *pc*[5].

The symbolic execution engine evaluates an `if`-condition under the path condition in order to establish which branches are feasible. Given a condition β of an `if`-statement and the current *pc*, the engine checks whether one of the branch conditions β or $\neg\beta$ is true under the constraints in *pc*. In other words, the symbolic execution checks the following expressions:

$$pc \implies \beta \tag{2.1}$$

$$pc \implies \neg\beta \tag{2.2}$$

Assuming *pc* is not strictly false, at most one expression is true. If one expression is valid, the execution follows the respective branch. In this case, it is optional whether the engine adds the branch condition to *pc*. However, if both expressions are satisfiable, both branches are feasible. As mentioned before, the execution forks, and the child processes inherit the program state

[5]It is important to emphasize that *pc* stands for path condition in the symbolic execution context[28, 40, 48], and it differs from another common use of *pc* as program counter

from their parent. The engine adds the branch condition to each process *pc* according to the direction taken.

Given these new semantics, the symbolic execution of a program runs as follows[6]:

1. **Initialization:** The symbolic execution starts with a single process from the program's entry block — as any concrete execution. Each input variable is considered symbolic, and the path condition is initialized as true.

2. **Sequential execution:** Next, the engine executes each instruction sequentially until it finds an `if`-instruction (step 3), or until it finds an exit instruction (step 6).

3. **If-statement execution:** The `if`-condition is evaluated under the path condition. If only one branch is feasible, the engine follows that branch, and continues the sequential execution described in step 2. If both branches are feasible, the execution must fork.

4. **Forking:** A process forks by generating two child processes, which inherit their parent's program state. The execution engine sets each child to follow one specific branch, and adds the corresponding branch condition to *pc*. Both children are added to a *worklist*, and the engine selects a process to follow (step 5).

5. **Process selection:** Conceptually, the engine can pick any process from the worklist to follow. Once the engine chooses a process, the engine restarts the process execution from the instruction pointed by the process program counter (step 2).

6. **Process termination:** Whenever an exit instruction is reached, the engine terminates the current process. The engine uses the terminated process *pc* to generate a concrete test input vector that leads the program execution under the completed path. After terminating a

[6] This step division is not standard, but didactic.

process, the engine checks whether there are any processes left in the worklist. If there are not, the symbolic execution is done. Otherwise, the engine goes back to step 5.

Thus, the symbolic execution engine keeps executing each forked process until one of the following cases:

- The engine executes every forked process until the end. In this case, it completely executed every feasible path in the program.
- The engine picks a process that never ends, i.e., the engine follows an infinite execution path, and it never terminates.
- The execution produces an infinite number of processes; thus, it never terminates.

The set of paths executed during the symbolic execution can be represented as a *symbolic execution tree*. In cases where the tree is finite, it also represents the program execution tree, and it represents every feasible path in the program. Moreover, every node in the symbolic execution tree represents a forking `if`-statement execution, and each leaf represents a process termination. The *pc* associated with any leaf is unique, and each concrete input generated in step 6 satisfies one and only one leaf *pc*. Additionally, there is at least one concrete input for each feasible execution path.

The description given in this session is a high-level overview of the symbolic execution algorithm introduced by King [28] in 1976, and which is still the base of most symbolic execution tools available today such as Klee [10], and Java PathFinder [48]. State-of-the-art symbolic execution tools implement additional optimizations that further improve symbolic execution scalability.

2.2.1 Solving Strategies

Symbolic execution is mainly used today for two purposes: to generate test cases, and as a bug finding tool. In these cases, symbolic execution can be

used as a search algorithm that seeks feasible paths that lead the execution to some points of interest in the program. In other words, symbolic execution traverses the program's CFG, while trying to find a feasible path that reaches a certain node. These nodes can represent program exit points, assertion failures or just uncovered program lines.

King's original description of symbolic execution does not define any systematic search strategy to explore the execution tree automatically [28]. King also introduced an interactive symbolic execution tool, named Effigy [27], where the user is responsible for choosing the process the tool will follow at step 5.

Since symbolic execution's introduction, many search heuristics has been introduced in order to choose which path the engine should follow, and to systematically explore the execution tree. The main search strategies implemented by symbolic execution software are described bellow:

Depth-first: In this strategy, the engine implements the worklist as a stack, always visiting the process that has been most recently created. After forking, the engine picks one of the process children, and pushes the other one to the top of the worklist; thus, it always follows one *complete* execution path, before switching to a different path. Once a process terminates, the engine picks the process generated in the most recent fork.

This strategy is implemented by many tools [10, 22, 48] due to its simplicity. It also allows for easy integration with other approaches that may enumerate the paths to be checked, such as concolic execution [22], and model checking [48]. However, in cases where the execution tree is unbalanced, the search can spend a lot of effort in one small portion of the tree — or worse, get stuck in an infinite path.

Breadth-first: A breadth-first search strategy is used to avoid process starvation [48], and it gives equal opportunities for the process to run. The worklist is processed in FIFO order, always selecting the oldest process.

The main drawback of this approach is that if the point of interest can only be reached deep in the execution tree, then the process may take a very long time to find it.

Uniform Random: In this approach, the engine randomly chooses which process following a uniform probability according to the process depth in the execution tree [9, 10]. The probability of each process to be picked is equal to 2^{-L}, where L represents the fork level on which the process was generated. Thus, this strategy favors processes that are shallow on the execution tree. As with any random search, each execution explores a different part of the execution tree, and results are hard to reproduce.

Coverage-Optimized: Many different greedy approaches have been proposed to select a process that is likely to cover new code [9, 10, 30]. These techniques usually compute a weight for each state based on the minimum distance to an uncovered instruction, and the recently covered new code. Then, the engine chooses the process randomly based on each state weight.

This approach adds some overhead to the process selection step; nevertheless, it is usually one of the most efficient strategies to reach uncovered program statements.

Modern symbolic execution tools usually apply one or more strategies described above. Some tools support a combined strategy, where more than one search strategy is interleaved using a round-robin scheduling technique.

2.2.2 Query Optimizations

In order to evaluate an `if`-statement, the symbolic execution engine employs some sort of constraint solver to check the branch conditions under the path constraint. Conceptually, the symbolic execution engine invokes a constraint solver every time a process execution reaches an `if`-statement, and this can impact on the engine performance. Because query complexity is exponential in the number of variables included in the formula, the number

of queries, and the size of each query, have a major impact on the symbolic execution engine's performance. For this reason many optimizations have been introduced in order to reduce query complexity, or to attempt to solve queries without invoking the constraint solver.

Constant Propagation: This optimization affects the instruction's semantics. When an instruction is executed, a simplification stage follows the expression generation. In this simplification, any arithmetic operation that only involves concrete values is replaced by the operation's result, which is also constant. Thus, instead of propagating formulas that involve only concrete values, this method propagates the results of those operations.

Constant propagation can be applied both statically, as a compiler pass, and dynamically in each individual execution path, according to the values assigned to the program variables. Constant propagation may completely eliminate queries that reference only variables with concrete values. Failing that, this operation may still reduce the size and complexity of queries.

Concrete Execution: In this strategy (also known as *concolic* execution), symbolic execution can be interleaved with concrete executions [22, 31, 44]. The concrete executions help symbolic execution by determining the feasibility of entire execution paths, and well as the feasibility of any subset of each path constraint. Each concrete execution is monitored to record the command sequence followed.

Implied Value Concretization: Some assumptions made along an execution path may imply variable assignments. For example, if a process takes a branch that has the condition $\sigma_i = 0$, the engine can assign 0 to σ_i, as well as replace any occurrence of σ_i in the path condition by the constant 0. After that, it can apply constant propagation to try to simplify the path condition even more.

Constraint Independence: Instead of testing whether a branch condition β is feasible under the complete path condition, the engine can try to

simplify the query by identifying which constraints in pc are independent from β. Two constraints β_i and β_j are classified as independent if they don't share any variables, and if there is no other β_m that is dependent on both β_i and β_j.

This optimization removes constraints that are completely disjoint from the branch condition being checked, and reduces the formula complexity without affecting its satisfiability.

Query Cache: This last optimization is also used to reduce the number of calls to the constraint solver. Query caches store the results of previous queries, to avoid making repeated (expensive) calls to the solver for the same conditions. When combined with constraint independence, queries might represent a sub-path condition; therefore, the cache can eliminate queries from different paths that includes the same sub-path condition.

Besides the optimizations described in this section, researchers have also focused on the extension of symbolic execution to support different language features, such as: pointers, heap modeling, environment modeling and concurrency. These optimizations are out of the scope of this thesis. This thesis presents an approach that is orthogonal to these advances, and it should benefit from them without any conceptual change.

Existing search techniques that employ some kind of learning to prune the number of paths executed during symbolic execution will be discussed in more detail in Chapter 6. Chapter 6 also contrasts these learning techniques with the one presented in this thesis.

2.3 SAT Solvers

The problem of deciding the satisfiability of propositional formulas, known as *Boolean satisfiability* or just SAT, is one of the most important open problems in complexity theory. SAT was the first problem proven to be NP-complete, and there are no known algorithms for solving with better

than exponential worst case complexity. Despite this, modern SAT solvers are often highly effective in practice. State-of-the-art solvers are capable of solving industrial problems with over a million variables and several million constraints.

While work on SAT goes back to the 1960s, the main advances in this area started in the late-90s with the introduction of *Conflict-Driven Clause Learning* (CDCL) SAT solvers. Since then, SAT solvers have been producing remarkable results as constraint solvers in areas such as software and hardware verification.

2.3.1 Definitions

A *propositional formula* is a logic expression defined over a set of n Boolean *variables*, $\{v_1, v_2, \ldots, v_n\}$, and that contains only three logic operations: *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$). Modern SAT solvers usually assume that the propositional formula are represented in a specific form: *Conjunctive Normal Form* (CNF).

A CNF formula consists of a conjunction of m *clauses*, $\{c_1, c_2, \ldots, c_m\}$, where each clause is a disjunction of one or more *literals*. A literal represents the occurrence of a variable v_i or its complement $\neg v_i$. Formula 2.3 is an example of a CNF formula with 4 clauses and 3 variables:

$$(v_1) \wedge (\neg v_2 \vee \neg v_3) \wedge (v_1 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3) \tag{2.3}$$

From now on, this thesis assumes that any propositional formula φ is represented in CNF, and CNF formulas will simply be called formulas.

Each variable in φ is a Boolean variable, and can either be *assigned* to *true* (1) or *false* (0). Assigning a variable to true (resp. false) is equivalent to assigning all positive occurrences of its literals to true, and all negative occurrences to false (resp. true), and this thesis will make assignments to literals or variables interchangeably. When a variable hasn't been assigned any value, it is a *free variable*. A set A of the variables in φ together with their corresponding assigned value is a *truth assignment*, or just assignment,

to φ. An assignment A is *complete* if it assigns a value to every variable in the given formula ($|A| = n$); if $|A| < n$, then A is a *partial assignment.*

A formula φ evaluated under a complete assignment A is either *satisfied* (true) or *unsatisfied* (false) by A. Additionally, if φ is evaluated under a partial assignment, φ can also be *undefined*, i.e., the evaluation cannot conclude the value of φ by the given assignment.

The satisfiability problem can be defined as the problem of finding whether there exists at least one complete assignment that satisfies φ. For a formula with n variables, there are 3^n possible partial assignments, of which 2^n are complete assignments.

2.3.2 DPLL SAT Solvers

The first efficient search procedure applied to SAT was introduced by Davis, Logemann and Loveland [14], and it was based on the resolution decision procedure presented by Davis and Putnam [15]. This search algorithm known as DPLL is the base of most modern complete SAT solvers, including CDCL solvers.

DPLL solvers take advantage of some useful properties of CNF formulas, which emerge from the fact that each clause in a formula φ can be evaluated separately under an assignment A. If any literal in a clause is assigned to true, the entire clause is assigned to true. If every literal in a clause is assigned to false, the clause is false, and so is φ. Otherwise, a clause is undefined under A. Thus, for a formula to be unsatisfied, it only requires one unsatisfied clause.

Another useful property emerges from clauses that have one literal only, such as the clause (v_1) in formula 2.3. These clauses are known as *unit clauses*, and any unit clause implies that its unique literal must be assigned to true in order for the formula to be evaluated to true.

Besides that, if a literal l occurs in a formula, and its negation doesn't, the literal is classified as a *pure literal.* In this case, this literal can be safely assigned to true. This is due to the fact that a satisfying assignment A either assigns l to true, or A can be modified by assigning l to true, and the

resulting assignment will also satisfy φ.

DPLL traverses the search space in a top-down approach. It starts from an empty assignment, and it iteratively assigns values to free variables, and checks whether the new assignment is a satisfying or an unsatisfying (partial) assignment.

At each iteration, the algorithm tries to find a variable assignment that is implied by the detection of unit clauses (known as *unit propagation*), or of pure literals. If no value is implied, the solver arbitrarily chooses a free variable v_i, and assigns either v_i or $\neg v_i$ to true. An arbitrary assignment is called a *decision*, and splits the search space in two. A DPLL solver maintains the set of decisions previously made as a *decision tree*, and a *decision level* is associated with every arbitrary assignment to denote its depth in the decision tree.

After any variable assignment, the algorithm creates a new formula φ' by modifying φ to:

- Remove the occurrence of the literal assigned to false.
- Remove the clauses with the literal assigned to true.

If φ' is empty, then the current assignment is a satisfying one. If φ' has an empty clause, then the current assignment is an unsatisfying one. Otherwise, the original formula is undefined under the current assignment, and the search continues by picking another variable assignment.

The occurrence of an empty clause is known as a *conflict*, and the algorithm *backtracks.* In DPLL, the search always backtracks to the previous decision level by reverting to the formula φ'. The search then checks whether it has tried both branches of the decision tree. If it hasn't, the search assigns the opposite value to the decision variable. If the search has tried both values, it has to backtrack one more level.

The DPLL search algorithm terminates when it finds a satisfying assignment, or when it backtracks past level 1 — there is no satisfying assignment.

Algorithm 2.1 CDCL framework implemented by modern SAT solvers. Given a CNF formula, CDCL is a search procedure that tries to find a satisfying assignment. The CDCL algorithm traverses the decision tree applying conflict analysis, clause learning, and non-chronological backtracking to reduce the search space.

Input: Set of clauses.
Output: SAT if there exists an assignment that satisfy all the clauses simultaneously, UNSAT otherwise.

```
function CDCL( )
    status ← PREPROCESS( )
    if status ≠ UNKNOWN then
        return status
    loop
        l ← DECIDE( )
        if l ≠ ∅ then
            status ← PROPAGATE(l)
            if status = CONFLICT then
                if level = 0 then
                    return UNSAT
                else
                    level, cause ← ANALYZE_CONFLICT( )
                    LEARN(cause)
                    BACKTRACK(level)
        else
            return SAT
```

2.3.3 CDCL SAT Solvers

The DPLL algorithm was proposed in 1962, and it remained the framework of state-of-the-art complete solvers until the introduction of *Conflict-Driven Clause Learning* (CDCL) SAT solvers. In 1996 [45], Silva and Sakallah presented a new SAT solver named GRASP, based on DPLL solvers. Unlike other DPLL solvers, GRASP was capable of analyzing and learning from conflicts reached during the search. CDCL solvers are named after GRASP's conflict-driven clause learning procedure.

Algorithm 2.1 presents pseudo-code for the CDCL framework. Like

DPLL, CDCL starts from an empty assignment, and the solver implements an iterative search. However, the search is modified to support conflict analysis, clause learning, and non-chronological backtracking.

Preprocess: In the beginning of the algorithm, a CDCL solver may perform some preprocessing to find out whether the formula's satisfiability can be trivially determined, or if any variable value can be implied.

Decide: Like DPLL solvers, if the solver can't deduce any free variable assignment, the solver picks a literal to assign to true. A CDCL solver also records the level of each decision.

Propagate: Once a decision is made, CDCL reduces the formula being assessed by removing any clauses containing the selected literal, and removing from each remaining clause any occurences of the complement of the decision literal. In practice, the solver uses efficient datastructures to manage these operations. During propagation, if the solver identifies a unit clause, its unique literal is assigned to true, and the new assignment is propagated; if it detects an empty clause, then it raises a conflict.

Analyze Conflict: If the search reaches a conflict, then the solver analyzes the set of decisions, and the assignments implied by them. To do so the solver either maintains and analyzes an implication graph, or implements a sequence of selective resolution operations. More details about how this operation is performed can be found in [45, 50].

Learn: Once the solver finds a subset of the assigned literals that together are sufficient to cause a conflict, the solver generates a new clause containing the negation of these literals. By adding this *learned clause* into the clause database, the solver will be prevented from returning to the same search space in the future.

Backtrack: The last step after a conflict is detected is a non-chronological backtracking. The solver has to backtrack to a level that reverts at

least one conflicting assignment. If an earlier decision is responsible for the conflict, the search can backtrack more than one level.

The search ends if it successfully assigned a value to each variable in the formula without raising a conflict, i.e., it found a satisfying assignment. Otherwise, if the formula is unsatisfiable, the solver terminates when a conflict is raised at level 0.

Efficient CDCL solvers also include many other implementation improvements introduced after GRASP. One example is the new unit propagation detection algorithm introduced by Moskewicz et al. [36], which reduced the solving time of hard instances up to two orders of magnitude. Their approach does not investigate every clause at each iteration while searching for new unit clauses. Instead, their approach picks two literals in every clause that have not yet been assigned false, and watches them. After assigning a new value to a variable, only the clauses watched by the corresponding false literal need to be visited (during which either a new watching literal will be found for each clause, or unit propagation will be triggered, or a conflict will be raised). This BCP algorithm is known as the *2-literal watching scheme*. Other improvements include restarts [24], sophisticated decision heuristics [23, 36], and learned clause deletion [23].

In the last 20 years, SAT solver advances have not been restricted to performance improvements. New features have been added to SAT solvers to improve their usability in different applications. The following ones are the most relevant to this thesis:

Unsatisfiability Core : In the process of generating an unsatisfiability proof to a formula φ, many modern SAT solvers can identify a subformula φ', $\varphi' \subseteq \varphi$, that is also unsatisfiable, known as an *unsatisfiability core*. This core can be very helpful in diagnosing the causes of some formula's infeasibility.

In practice, the goal is to generate a small unsatisfiability core, known as *minimal* unsatisfiability core. However, this is a hard problem, harder than proving that a formula is unsatisfiable, and there exists

many different strategies that attempt to efficiently identify small and minimal cores (for more details see [38, 51]).

Incremental Solvers : An incremental SAT solver is a solver capable of reusing its solving state after solving a formula φ_i to successively solve similar formulas. The variable activity used in decision heuristics, deletion / restarts strategies and others can be safely reused independently of the formulas. However, learned clauses can only be reused if no clause of φ_i is deleted. A solver can keep the learned clauses to solve φ_j after solving φ_i if:

$$\varphi_i \subseteq \varphi_j$$

Because φ_j contains every clause in φ_i, this technique can be used to solve problems incrementally. This process is much faster than solving each formula independently.

Assumption Mechanism : Eén and Sörensson [19, 20] observed that it is safe to keep the clauses learned even if some clauses are deleted. However, this only applies if all the removed clauses are satisfied at the top level (such as unit clauses). Given this observation, the authors introduced a modification in the preprocessing stage that allows a solver to check a formula satisfiability under *initial assumptions* (or just assumptions). An assumption is a unit clause that represents a variable assignment that should be propagated before a formula is solved. This assumption mechanism allows the solver to simulate the effect of removing clauses from a formula.

For example, a user can invoke an incremental solver successively to check φ_i and φ_j, even if $\varphi_i \nsubseteq \varphi_j$. For that, he / she adds a new variable v_k to both formulas, and adds the literal v_k to every clause that is in φ_i but not in φ_j. Then, the user invokes the solver to check whether the modified φ_i' under the assumption $\neg v_k$ is satisfiable. During preprocessing, the solver propagates the assumption, which removes every occurrence of the literal v_k, and the φ_i' is reduced to φ_i.

After solving φ_i, the user can add the clauses that are exclusive of φ_j, and invoke the same solver; however, the user passes v_k as an assumption. The solver propagates this new assumption, which trivially satisfy all clauses that contain v_k. Thus, the solver can remove them from search.

The assumption mechanism also provides a powerful and lightweight mechanism to detect the unsatisfiability reason, or even the unsatisfiability core. Because each assumption is treated as a unit clause, the solver will still represent it in the implication graph. Once the solver proves that a formula is unsatisfiable, the solver can analyze the conflict, and check if the final conflict is a consequence of any of the assumptions given. If there is no assumption involved, the solver can conclude that the formula is unsatisfiable. On the other hand, if there are one or more assumptions involved in the final conflict, the solver concludes that the formula is unsatisfiable under the assumptions given. Additionally, the solver provides a subset of the assumptions that is sufficient to make the formula unsatisfiable [18].

2.4 CDCL Applications

The unquestionable efficiency of CDCL SAT solvers has inspired many researchers to apply the same reasoning behind CDCL to solve problems in different domains. For example, Satisfiability Modulo Theories (SMT) solvers employ SAT solvers to enumerate abstract solutions, and a theory solver, which is able to handle atomic constraints in some decidable first-order theory to test and refine each solution. The efficiency of modern SMT solvers comes from the integration of the CDCL SAT solver and the theory solver. The SAT solver consults the theory solver after every decision, and the theory solver may return a new deduced assignment, as well as a conflict clause — if a conflict is detected. This tight integration, known as *lazy SMT*[7], is responsible for a great advance in SMT performance, and it has a direct

[7] The name was given in contrast to *eager* SMT, which encodes the SMT formula into an equivalently satisfiable Boolean formula before invoking a SAT solver.

impact in symbolic execution, since most symbolic executions employ some kind of SMT solver as their constraint solver [43].

Another example is the *Conflict Driven Fixed Point Learning* (CDFL) technique [17], which instantiates a CDCL architecture over abstract domains. This technique was successfully applied for bounds analysis, and it was mathematically generalized to lattice-based abstractions [16].

In the software verification context, the insights of modern SMT solvers, and consequently CDCL solvers, has inspired the introduction of a program analysis named *Satisfiability Modulo Path Programs* [25]. Chapter 6 describes this analysis technique at length, and compares it to the method introduced in this thesis.

Chapter 3

Conflict-Driven Symbolic Execution

Symbolic execution has been highly optimized to generate lightweight queries to the constraint solver, which allows tools to successfully explore many execution paths even in large programs. Nevertheless, symbolic execution tools still try to exhaustively execute every feasible path in order to prove/disprove certain properties. This strategy resembles DLL SAT solvers; thus, symbolic execution might benefit from CDCL techniques, in the same way that SAT solvers did.

Therefore, this chapter presents a novel symbolic execution algorithm that avoids executing every execution path, named Conflict-Driven Symbolic Execution (CDSE). In a nutshell, CDSE prunes the search space every time a certain branch is proven infeasible, by learning the reason why there is a conflict.

The initial CDSE description considers programs that have no loops and only one property. Section 3.4 describes how CDSE can handle multiple properties, while Section 3.5 presents different CDSE solutions to verify programs with loops.

In addition, this thesis assumes that a program property ρ is encoded as a boolean expression β_ρ. The property ρ is embedded in the source code as an `assert` statement:

$$\texttt{assert}(\beta_\rho)$$

which is semantically equivalent to the following `if`-statement:

$$\texttt{if}\ \neg\beta_\rho\ \texttt{then}\ abort()$$

The branch that represents the transition to the *abort*() statement, i.e. a property violation, will be represented with a μ

3.1 Overview

Like any symbolic execution algorithm, CDSE is a search algorithm. In CDSE, each program branch statement and a possible direction represents a decision. From now on, this pair will be called a 'branch', inspired by the program's control flow graph (CFG) representation. Thus, CDSE explores the program's search space by choosing which branches should be executed in sequence.

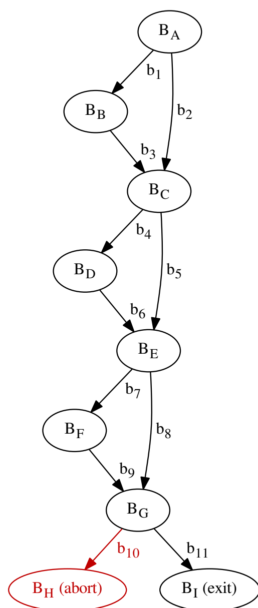
In order to reduce the search space, the CDSE search engine prunes paths by employing the same insights from CDCL solvers: if a certain subset of decisions raises a conflict, this conflict can be used to backtrack non-chronologically, and to preempt the occurrence of similar conflicts.

The engine uses the control flow graph (CFG) and facts learned from previous attempts to guide its future decisions. The engine iteratively chooses which branch to take, and then evaluates this decision's consequences before making a new decision.

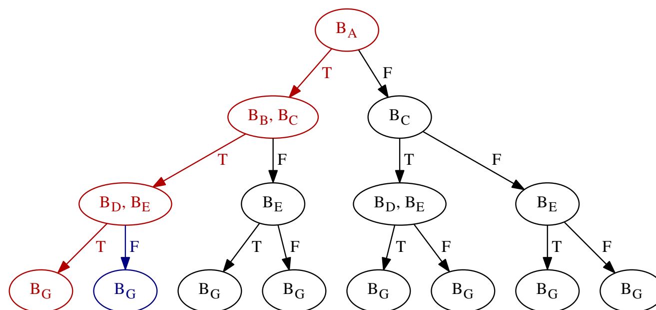
For example, going back to Listing 1.1 at page 5, conflict-driven symbolic execution exploits the same intuition presented in Section 1.1 to prove that the given assertion always holds by executing only 2 out of 8 execution paths. In addition, it extracts information from the program's CFG to optimize the learning process, as well as the decision procedure.

In order to prove that the assertion always holds, the CDSE engine searches for a feasible path between the entry node and the assertion failure `BB` (the TRUE BB of the translated `if`-statement). First, the engine extracts from the program's CFG (shown in Figure 3.1(a)) which branches could never be included in any path that connects the entry node (B_A) and the assertion `TRUE BB` (B_H). As a result, the engine learns branch b_{11} would never be included; therefore, it should never be taken.

After that, the engine symbolically executes the program by alternately choosing which branch to follow, and checking whether it is feasible or not.



(a) Program's control flow graph



(b) Program's execution tree — modified copy from Figure 1.1 representing executed basic blocks.

Figure 3.1: The control flow graph and the execution tree for the program described in Listing 1.1. (a) In the CFG, each branch b and each basic block B has a unique ID. (b) The execution tree represents the 8 paths in the CFG that would lead the program to check the assertion. The execution tree edges represent only conditional branches; thus, one node may represent the execution of more than one basic block.

Following the description in Section 1.1, in order for the CDSE engine to execute the path marked in red in Figure 3.1(b), the engine must choose to follow the branches b_1, b_3, b_4, b_6, b_7, b_9, and b_{10}. When the engine reaches b_{10}, and checks if its condition is feasible, the engine finds a conflict.

The branch b_{10} condition is not feasible because of the constraint '$x < y$' and the result definition from assignment '$result = y - x$' (inside B_F). The engine adds the constraint '$x < y$' to the path condition as a consequence of the decision to follow b_7. Additionally, the `result` value is defined in B_F because the engine takes b_9. Consequently, the decision of taking branches b_7, b_9 and b_{10} led the symbolic execution into a conflict, and the engine learns that it should never take them together again. This learning prunes 4 possible execution paths (including the red path) from its search.

In order to leave the conflicting state, the engine has to backtrack and remove at least one of the conflicting decisions. The engine can keep decisions that led it to take b_1, b_3, b_4, b_6, but it cannot take either b_7 or b_9. Taking b_7 implies that execution will follow b_9, and b_9 execution must follow b_7's. Moreover, the engine knows that only paths that include b_{10} are relevant to the proof — since all paths either include b_{10} or b_{11}, and previous analysis excluded b_{11}'s.

If the engine decides to keep the same decisions until b_6, and to take b_8 and b_{10}, the execution follows the path with the blue edge. Once again, the engine reaches a conflict when it checks whether b_{10} is feasible. This conflict emerges because of branches b_8 and b_{10}. As a consequence to the decision of taking b_8, the engine adds the constraint '$x > y$' to the path condition, and the definition of variable `result` ($result = x - y$) from B_E reaches B_G. This constraint and this definition together conflict with the branch b_{10} condition $\neg(result >= 0)$. Therefore, the conflict detected is an invariant to every path that includes b_8 and b_{10}.

CDSE concludes the assertion always holds after applying what the engine learned from both: the two conflicts it reached and the program's CFG.

3.2 Learning Scheme

In order to reduce the search space, the CDSE engine must be able to learn from its bad decisions, as well as to make new decisions based on the facts learned from previous iterations. Thus, the CDSE learning scheme provides a mechanism to extract the decisions that led the search to a conflict, and to create a new constraint that blocks them. Moreover, this learning scheme also provides a mechanism to restrict the engine's choices according to the constraints previously learned. Consequently, every time a conflict is reached, the engine learns a new constraint and the available choices become more restricted.

3.2.1 Search Constraints

Every decision made by the CDSE engine respects the constraints imposed by the program's CFG and the facts learned along the search. In other words, the engine should pick only branches that compose a CFG path *and* that haven't been blocked yet.

The engine represents these constraints as a set of clauses (CNF). The CNF representation allows the engine to use a SAT solver to find decisions that do not violate the search constraints. From now on, this CNF will be called *fCFG*.

An fCFG is a formula used to represent the constraints that guide the CDSE search, including those extracted from the program's CFG. Each variable in the fCFG formula has a one-to-one correspondence to an element in the CFG — either a branch or a basic block. Therefore, the engine can easily map elements of one set to the other.

The engine uses a SAT solver to solve the fCFG and consequently to enumerate CFG paths that would lead the program execution to a property violation, given that no conflict is detected by the engine. Thus, the fCFG is encoded to be satisfiable if and only if there is a path in the CFG between the program's entry node and a property failure μ. Consequently, the reachability problem becomes a satisfiability problem, where:

1. The variable corresponding to the entry node must be true.
2. Any variable corresponding to an exit node (excluding the abort call from the property encoding) must be false.
3. A branch variable v_b can be true if and only if the origin and target variables are also true, when they exist.
4. A basic block variable v_B can be true if and only if at least one incoming edge variable is true.
5. A basic block variable v_B can be true if and only if at least one outgoing edge variable is true.
6. The variable corresponding to the property violation μ must be true.

Note that the constraints described above do not restrict satisfying assignments to represent only one path. However, they guarantee that a satisfying assignment represents at least one non-blocked path that connects the entry node to the property violation in the CFG. Moreover, if a variable v_b is assigned to true in a satisfying assignment, these constraints guarantee that one of its path includes the branch b.

Initially, the fCFG includes only constraints derived from the CFG. These constraints are enough to block the decisions that could lead the search to an ill-defined path in the CFG, or to an irrelevant path to the property proof. The initial fCFG is satisfiable unless every assertion statement is trivially marked as dead code during CFG construction.

During the search, the engine tries to execute a CFG path that respects the current assignment to the fCFG. If the path is feasible, than it finds an assertion failure. On the other hand, if the path is infeasible, the engine performs a conflict analysis to establish a set of branches that led to the conflict. This conflict analysis, which is described in the next section, returns a new constraint, represented as a clause, that blocks any path that contains every branch in the given set. The engine adds this clause to the fCFG, and invokes the SAT solver to find a new assignment that respects the increased set of constraints.

3.2.2 Conflict Analysis

In order to determine a reason of a conflict, the symbolic execution of a path is slightly modified. The changes allow the CDSE engine to represent the conflict reason as a clause, and to learn from it by adding the clause to the fCFG.

In CDSE, the engine chooses which branches to take in sequence, and it executes a path following these decisions. The engine executes a program statement as a consequence of the past decisions. Therefore, the CDSE expression generation reflects not only the instruction semantics, but also the decisions that led the instruction execution.

In order to represent this dependency, the expression encoding includes *instruction guards*, from now on represented with δ. An instruction guard is a boolean symbolic variable that express the condition that the search must meet in order to execute the guarded instructions. In CDSE, the instruction guard is true if and only if the immediate preceding branch is taken.

The engine encodes an instruction guard as a boolean symbolic variable δ_i that follows the same identification number i as the last branch taken b_i. After taking branch b_i, and before taking any other branch, the engine guards the instructions with the same variable δ_i. In CDSE, the engine may guard an assignment or boolean conditions.

Given an assignment

$$\sigma \ := \ \langle expr \rangle$$

where σ is a program variable, and $\langle expr \rangle$ is the expression to be assigned by regular symbolic execution. The engine guards such an assignment by extending the expression to be assigned to a `select` expression. A `select` expression represents a ternary operation that contains a boolean condition, and two possible assignments. The evaluation of the boolean condition determines which assignment is chosen.

In this new expression, the variable σ gets the $\langle expr \rangle$ if the guard δ_i is true. Otherwise, σ can get any value, i.e. its value is nondeterministic, which is represented as a $\langle nondet \rangle$ symbol. The original assignment is then

replaced by the following guarded assignment:

$$\sigma \;:=\; \delta_i \;?\; \langle expr \rangle \;:\; \langle nondet \rangle$$

Consequently, the variable σ will be defined as $\langle expr \rangle$ if the program execution reaches the assignment instruction — if it takes the preceding branch b_i; otherwise, σ can have any value.

The engine can use data flow analysis to optimize the number of assignments to guard. CDSE does not need to guard a variable assignment if its definition dominates every possible use.

Besides assignments, CDSE also guards constraints generated from conditional branches. Before executing a conditional branch command, the engine chooses a branch b_i that it will follow. Then, the engine creates a new expression with the original branch condition $\langle cond \rangle$ and the instruction guard δ_i. In this new expression, δ_i implies that $\langle cond \rangle$ must be true. In other words, if the branch b_i is taken, the branch condition must be true. Thus, the following constraint is added to the path condition:

$$\delta_i \implies \langle cond \rangle$$

Given the new encoding, the engine invokes the constraint solver passing all guards to the current path as assumptions, i.e., the solver has to solve the given formula assuming every guard is true. Consequently, CDSE still passes an equally satisfiable problem to the constraint solver. Additionally, if the path is infeasible, CDSE leverages the constraint solver conflict reasoning to provide which guards — and which branches — are responsible for a conflict.

Once the constraint solver returns a conflict clause, CDSE has to translate the guards involved in the conflict returned to their respective branches. The translation follows a lightweight mapping mechanism, since the guards identification number is the same as its relative branch. Finally, CDSE generates a conflict clause based on the branches that led to the current conflict, and adds such a clause to the learned clause database.

Thus, CDSE explores the program structure in three different contexts.

The first one is the program CFG, which represents the set of all possible transitions in the program. CDSE uses the CFG to build an initial knowledge about the program. Second, CDSE represents the CFG as a formula, fCFG, where the algorithm can easily add new facts (learned clauses). The fCFG represents each CFG element as a boolean variable. The last context is the symbolic execution encoding itself. CDSE introduces the concept of instruction guards, which are symbolic variables that represent branches in this context. CDSE employs these guards in order to perform a conflict analysis. Therefore, each branch b_i in the program CFG has a corresponding fCFG variable v_i, and a corresponding guard δ_i.

3.3 CDSE Algorithm

The conflict-driven symbolic execution algorithm systematically explores the search space by deciding which branches to take given the knowledge it built from previous decisions. CDSE starts from the entry node in a program, and symbolically executes the program until it reaches a branch statement. Then, CDSE chooses which branch should be taken next, and checks whether such decision is feasible or not. CDSE iteratively executes instructions between the chosen branches, until it reaches the violation branch b_μ, or the branch chosen is infeasible. If the b_μ branch is feasible, a bug was found and a concrete counter example (CEX) is extracted. On the other hand, if a branch is infeasible, a reason for the conflict is determined. The CDSE engine, then, learns which set of branches should not be taken in order to avoid the same conflict, and backtrack.

Algorithm 3.1 presents a pseudo-code for the algorithm described above. Like CDCL solvers, the algorithm can be divided into the following main steps:

Preprocess : In lines 1-7, CDSE employs different static analysis techniques to optimize the code and to extract relevant information for the next steps.

Decide : At every iteration, symbolic execution reaches a branch state-

Algorithm 3.1 Pseudo-code for the conflict-driven symbolic execution proposed in this thesis.

Input: Program P, Bound B and Assertion A
Output: TRUE if assertion can't be violated, a CEX otherwise

```
 1: P' ← ANALYZE_AND_OPTIMIZE(P)
 2: cfg, fCFG ← BUILD_CNF(P', B, A) // Build CFG and translate to CNF
 3: paths ← SOLVE(fCFG)
 4: if paths = ∅ then
 5:     return TRUE
 6: pcond ← ∅ // Path conditions
 7: dstack ← ∅ // Decision stack
 8: loop
 9:     branch ← DECIDE_NEXT_BRANCH(cfg, paths, dstack)
10:     PUSH(dstack, branch)
11:     pcond ← pcond & GET_COND(P', branch)
12:     if IS_FEASIBLE(pcond) = TRUE then
13:         if next = ¬A then
14:             return COMPUTE_CEX(P', pcond)
15:         else
16:             SYM_EXEC_STEP(branch) // Current branch → next branch
17:     else
18:         lclause ← ANALYZE_CONFLICT( )
19:         ADD_CLAUSE(fCFG, lclause)
20:         paths, pcond, dstack ← BACKTRACK(fCFG, pcond, dstack)
21:         if paths = ∅ then
22:             return TRUE
```

ment, and decides which direction to take. Lines 9-10 represent the decision procedure.

Propagate : Once a decision has been made, the algorithm propagates it. From line 11 to 16, the path condition is extended and checked. If feasible, the algorithm checks whether it has found a bug. If a bug was found, the algorithm returns a counter example. Otherwise, CDSE executes the instructions from the last branch until the next branch.

Conflict Analysis : If a conflict was reached, the algorithm establishes

which decisions led to this conflict (Line 18).

Learn and Backtrack: In lines 19-22, the algorithm learns to avoid the conflict detected, and backtracks to some previous branch where this conflict is no longer implied. If there is no path left, the algorithm returns TRUE.

3.3.1 Preprocess

This step precedes the symbolic execution, and the goal of this step is to extract relevant information for the following steps, as well as simplify the program instructions. During Preprocess, the engine can apply many static analysis techniques included in modern compilers to optimize the program, such as constant propagation, loop unrolling, dead code elimination, etc. Additionally, the engine gathers two piece of static information that are required in further steps: the control flow graph and the use-def chain.

The control flow graph can be easily retrieved by visiting each basic block in the program and connecting it to its immediate successors — its branch targets. Because decisions are based on branches, the CFG contains an entry edge that points to the entry node. As mentioned in section 3.2, the engine also represents the CFG as a CNF (the fCFG), which facilitates the decide and learn steps. Thus, the engine also initializes the fCFG, and uses a SAT solver to find an initial assignment during preprocessing.

The second piece of information needed is the set of all definitions of a program variable (namely, variable assignments) that can reach a certain usage, known as the use-def chain. Modern compilers already compute this data flow information while transforming the program into static single assignment (SSA) form. In SSA form, whenever a variable value can come from different sources, due to join points in the CFG, phi-functions (or phi-nodes) are added to the beginning of basic blocks. Semantically, phi-nodes are conditional assignments, where a variable is assigned to its latest value in the preceding block. The phi-nodes carry information about where variable values may come from, and the CDSE engine employs this knowledge to define which assignments have to be guarded.

3.3.2 Decide

The search starts after the preprocess step. At every iteration, the first step is to decide which branch to take next. Each decision is based on the CFG, and the current assignment to the fCFG variables.

First, the decision procedure identifies which choices are available by using the CFG to find the existing basic block transitions that originated from the last basic block executed. The procedure then refines the set of choices available by using the current assignment to the fCFG. The decision procedure can only pick a branch b_i if the fCFG variable v_i has been assigned to TRUE. If after the refinement there is more than one choice available, the decision procedure could apply different strategies to try to guess which one is the best choice, or choose randomly.

After a decision is made, the fCFG variable picked is pushed onto a stack, named the *decision stack*. The new size of this stack determines the level associated to this last decision.

3.3.3 Propagate

After the decision procedure picks a branch b_i, CDSE propagates every change implied by this choice. This includes checking if the branch b_i is feasible, and symbolically executing the instructions that follow it. Symbolic execution will continue until it reaches the next conditional branch statement.

First, the engine has to check the branch condition under the current path constraints. For that, the engine derives the branch condition from the current branch statement and the direction to be taken. The engine guards the constraint with the δ_i that is related to the branch b_i, and extends the path condition by adding this guarded constraint.

Given this new path condition and the current decisions, the engine invokes the constraint solver to check whether the path condition is satisfiable or not. If the new path condition is infeasible, CDSE found a conflict, and the engine follows the conflict analysis step. Otherwise, the propagation continues.

CDSE propagation continues by checking whether it has reached a bug. In which case, it retrieves a valid counter example from the constraint solver, and returns it to the user.

If propagation does not reach any conflict or any bug, the engine symbolically executes the instructions that follows b_i, and it uses δ_i to guard instructions whenever it is necessary. If during the symbolic execution, the engine reaches an unconditional branch b_j, the engine adds b_j to the decision stack as an implied decision, and starts using δ_j instead.

3.3.4 Conflict Analysis

The conflict analysis' goal is to establish a subset of decisions that led the search into a conflicting state, where the path being explored is no longer feasible. CDSE exploits the assumption mechanism adopted in many CDCL-based constraint solvers to implement a lightweight and efficient conflict analysis.

As explained in Section 3.2.2, the CDSE engine encodes the instruction guards as assumptions to be respected by the constraint solver. Therefore, if the constraint solver finds that the current path condition is unsatisfiable, the solver returns a reason for the conflict as a conflict clause based on the assumptions given.

Because the conflict clause returned by the constraint solver is based on guards and not on the fCFG variables, the engine has to translate this clause as a function of the fCFG variables. In order to differentiate the conflict clauses returned by the constraint solver from the ones returned by the CDSE conflict analysis stage, from now on, this thesis will use *execution conflict clause* for the former and *CDSE conflict clause* for the latter. Hence, an execution conflict clause is based on instruction guards, while a CDSE conflict clause is based on fCFG variables.

As mentioned in Sections 3.2 and 3.3.1, the CDSE engine uses the same identification number i to identify a branch b_i and its corresponding fCFG variable v_i and guard δ_i. Thus, the engine uses this identification number to map guards to fCFG variables, and to perform the translation from an

execution conflict clause to a CDSE conflict clause.

As a result, the conflict analysis procedure returns a clause with the negation of every fCFG variable involved in the conflict. For example, if branches g_i and g_j are responsible for the conflict, the conflict analysis will return the CDSE conflict clause $(\neg v_i \vee \neg v_j)$. This clause summarizes which decisions can be taken to avoid the same conflict.

3.3.5 Learn and Backtrack

In this last step, the CDSE engine applies the information returned by the conflict analysis to undo previous decisions, and to preempt the occurrence of similar conflicts. First, the engine expands the fCFG by adding the CDSE conflict clause retrieved from the conflict analysis. This clause represents a new constraint that excludes the corresponding decision subset responsible for the last conflict. Therefore, future satisfying assignments to the fCFG can't assign true to every fCFG variable responsible for the conflict.

Once the conflict is learned, the engine needs to backtrack its previous decisions in order to leave the conflicting state. CDSE backtracks by solving the expanded fCFG, and looking for new paths to follow. If the fCFG is no longer satisfiable, every path has been blocked, and CDSE has proven the target property always holds.

In contrast, if the SAT solver finds a satisfying assignment to the fCFG, the engine checks which variables in the decision stack are no longer assigned to true, and backtracks to a level where all these decisions are removed. The engine also removes from the path condition the constraints added due to the backtracked decisions, and the algorithm goes back to the decision step, starting a new iteration.

3.4 Multiple Properties

Conflict-driven symbolic execution can simultaneously verify multiple properties in the target program. Adding support for multiple properties requires just a few changes to the preceding algorithm description. First, the unit

clause v_μ, defined in Section 3.2 item 6, is no longer added to the fCFG. Instead, the fCFG translation adds a property clause to the fCFG. The property clause is defined as the union of every variable relative to a property failure μ_i, or:

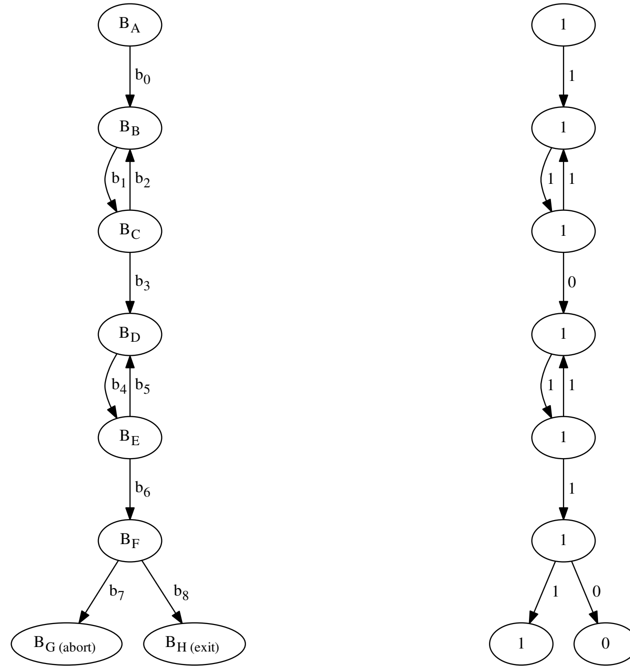
$$\bigvee_i v_{\mu_i}$$

This change is enough if the goal of CDSE is to prove whether the program respects all properties given, i.e., whether it can find any property violation. However, if the goal is to prove/disprove every property given, Multiple Property CDSE includes a second change. Whenever the symbolic execution reaches the violation of a property ρ_i, the engine saves the counter example found and learns to never look for the same violation. For that, the engine adds to the fCFG the unit clause $(\neg v_{\mu_i})$, which blocks every complete path that could violate ρ_i again.

Even though this unit clause blocks every assignment to the fCFG that assigns v_{μ_i} to true, this will not restrict the search for a violation of a non-blocked property. Note that there is no path in the CFG that can include two property violations. A satisfying assignment to the fCFG that includes more than one property violation must also include independent paths that connect each one of them to the entry node. Thus, for any satisfying assignment to the fCFG that assigns v_{μ_i} and v_{μ_j} to true, there is a second satisfying assignment that does not include v_{μ_i}, but that still includes the same paths to μ_j. Adding a unit clause $(\neg v_{\mu_i})$ to the fCFG will block the first assignment, but not the second.

3.5 Loop Handling

So far, the CDSE description relies on the absence of loops in the program control flow graph. In the presence of loops, an assignment to the fCFG no longer guarantees the existence of a path that connects the entry node to one of the target nodes (See Figure 3.2 for an example). Therefore, CDSE must handle the loops in the CFG before converting it to a CNF.



(a) A CFG with loops. (b) A satisfying assignment to the fCFG.

Figure 3.2: An example of a satisfying assignment to the fCFG that no longer guarantees the existence of a complete path in a CFG due to the existence of loops. (a) Represents a program CFG with two loops. (b) Represents a satisfying assignment to the corresponding fCFG. In this representation, each CFG element's identification is replaced by the value assigned to the corresponding fCFG variable.
Any complete path in the given CFG should include b_3; however, a loop in the CFG allows a node to succeed its predecessor, and rules 4 and 5 can be satisfied without guaranteeing that b_3 must be taken in order to reach B_D.

For example, the CDSE engine can apply source code modifications to the program target P, unrolling all loops until a given bound n. In this case, the symbolic execution is bounded, and it can prove property correctness only for execution paths that respect the loop bound, and no other conclusion could be taken about the program's behavior past the bound.

However, source code loop unrolling is expensive, since it generates n copies of the same instruction sequence, and it can be easily avoided in CDSE. Because the CDSE engine takes advantage of the CFG to guide its search, the engine can apply more sophisticated approaches to handle loops, such as:

Static CFG Unrolling: Instead of unrolling the program's source code, the engine can unroll loops only in the CFG.

Dynamic CFG Unrolling: The engine can also unroll the CFG dynamically. The algorithm starts by unrolling every loop twice, and assuming that a loop second iteration should not be taken. Then, CDSE proceeds with the search until it either finds a counter example, or blocks every path available. In the latter, the engine checks if any assumption added causes the current fCFG to be unsatisfiable. If that is the case, the engine removes the assumption, unrolls the loops that were involved in the conflict, and continues the search.

Chapter 4

Kite: A Conflict-Driven Symbolic Execution Tool

This chapter presents my proof-of-concept conflict-driven symbolic execution (CDSE) tool named Kite[8], which was used to evaluate the performance of this new algorithm. This chapter describes Kite's implementation details, including which changes were necessary to employ many of the symbolic execution optimizations described in Section 2.2.2.

4.1 Kite's Architecture

Kite was developed to verify embedded assertions in sequential programs. Kite verifies programs described in the LLVM assembly language, known as LLVM IR, which can be obtained by parsing C files using the LLVM C front-end [41].

Kite was mainly constructed on top of a state-of-the-art symbolic execution tool named Klee [10, 39]. Kite's implementation can be divided into four main modules:

Preprocessor: The preprocessor is responsible for reading the input file, as well as analyzing, and optimizing the program before the symbolic execution starts.

Instruction Interpreter: This module interprets the program instructions following the symbolic execution semantics. The interpreter implements the entire CDSE propagation step.

[8] Kite is an open-source project and its source code can be retrieved from: `www.cs.ubc.ca/labs/isd/Projects/Kite`

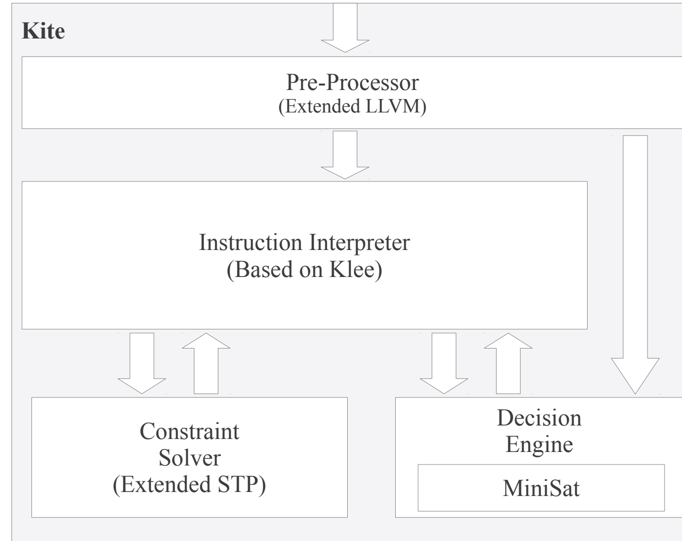


Figure 4.1: Kite's main modules, their composition and communication. The preprocessor module provides information to the instruction interpreter and the decision engine. The instruction interpreter symbolically executes the code guided by the decision engine. The interpreter and the constraint solver share information about path condition feasibility.

Decision Engine: The decision engine chooses which branches to follow, as well as the backtrack level, whenever a conflict is detected. Thus, it implements the decision step, as well as learning and backtracking steps.

Constraint Solver: The constraint solver is responsible for analyzing the path constraints, and deciding whether they are valid or not. In cases where the constraints are not valid, it returns an execution conflict clause. Moreover, the constraint solver also extracts an assignment to the program input variables when the execution reaches an assertion failure.

Figure 4.1 depicts how these four modules are organized inside Kite, as well as the communication between them. The preprocessor is the first mod-

ule to act in Kite's execution, and it provides information to the instruction interpreter and the decision engine. The symbolic execution is then coordinated by the instruction interpreter. This module consults the decision engine every time a decision is required, or after any conflict is detected. Besides that, the interpreter invokes the constraint solver to check whether any modification in the path condition is feasible.

Kite inherited three of its modules from Klee: the preprocessor, the constraint solver and the instruction interpreter. Consequently, Kite also inherited three tools used in Klee: LLVM [32], STP [21], and CryptoMiniSat2 [46]. LLVM is the base of the preprocessor, while STP and CryptoMiniSat2 comprise the constraint solver.

Kite introduces some extensions to the preprocessor and the constraint solver in order to support the CDSE preprocessing requirements and the execution conflict detection. These two modifications are described in Sections 4.2 and 4.4, respectively. Kite's instruction interpreter is the result of many modifications applied to Klee's symbolic execution engine to implement a conflict-driven approach. These changes are described in Section 4.3.

The decision engine is a new module exclusive to Kite. The decision engine includes an underlying SAT solver in order to solve the search constraints. The underlying SAT solver is an extended version of MiniSat [20], which was modified in order to implement different decision heuristics as described in Section 3.2.1.

4.2 Preprocessor

The preprocessor is responsible for reading, analyzing, and optimizing the program under verification. As previously mentioned, Kite's input is a program described using the LLVM IR, and the preprocessor module is mainly composed of the LLVM compiler itself.

The LLVM (Low Level Virtual Machine) compiler was introduced in 2004 by Lattner and Adve [32]. The authors designed LLVM to be a flexible and easy-to-extend compiler framework. Since then, LLVM has grown, and it incorporates many program analysis procedures and optimization passes.

Furthermore, LLVM also behaves as a virtual machine capable of interpreting its own assembly language, the LLVM IR.

The LLVM provides optimization passes that reduce the number and cost of instructions in the final executable. These passes can also be used to optimize the IR for symbolic execution. Many operations that are hard for computer processors are also hard for constraint solvers. For example, memory access and division operations usually demand many machine cycles. These operations also cause big overheads during symbolic execution. Memory access is hard to model, while the translation of a division operation produces symbolic expressions that are hard to solve.

Therefore, Kite includes many LLVM's optimization and analysis passes. Kite uses these passes to transform the code into SSA (Single Static Assignment), to reduce memory access, to apply constant propagation, to simplify arithmetic expressions, to canonicalize loop induction variables, etc.

Additionally, Kite introduces two new passes to retrieve the information necessary to build the program's control flow graph (CFG) required by the CDSE: Loop Analysis and CFG Analysis. The first pass identifies every loop in the program, as well as its structure (back edge, entry block and exit blocks). The second pass is a function analysis pass, which is used to collect the program's structure, i.e., this pass collects every basic block, as well as the transitions between them (branches and function calls).

4.2.1 CFG Construction

The CDSE learning scheme relies on the knowledge of the program's control flow graph. Since LLVM contains only separate representations of CFGs per program function, Kite's preprocessor implements a method to build a unique CFG for the entire program.

The CFG construction can be divided into two stages. In the first one, the preprocessor collects the required information about the program's structure using the Loop Analysis and the CFG Analysis passes. In the second stage, the preprocessor builds the CFG using the information collected. It encodes every basic block as a node, and every branch as an edge. The

preprocessor also decodes every function call in a process similar to function inlining. For that, the preprocessor divides the callee basic block into two nodes, which represents the instructions executed before and after the call. Then, the preprocessor duplicates the called function CFG and connect it to the two new basic blocks. The preprocessor also unrolls every loop and every recursive function until a certain depth. In both cases, the iteration past the depth is represented with an exit node, which is handled in the fCFG construction.

Additionally, the preprocessor employs an optimization to reduce the CFG to nodes that are relevant to the search. This optimization follows the intuition behind *program slicing* [49]. A program slice is a subset of instructions that can directly or indirectly have influence over a program component. Since the assertion statements are the symbolic execution targets during the program verification, basic blocks that cannot precede these statements are removed from the CFG representation. This optimization consists of a simple backward traversal that originates from every assertion statement, followed by the removal of every node that the traversal didn't visit. Exit nodes are connected to the end of the edges that connect a visited node to a non-visited one.

4.2.2 fCFG Construction

The preprocessor is also responsible for generating the fCFG representation, which is used by the decision engine to guide the CDSE search. Once the preprocessor generates the CFG, the preprocessor translates this CFG into a conjunction normal form (CNF) formula. The preprocessor traverses the CFG, and applies the rules described in Section 3.2.1 (page 32) in order to perform such a translation.

Every exit node is represented as one single variable v_{exit}, and a unit clause $\neg v_{exit}$ is added to the CFG. This excludes from the search every path that reaches the program's return statement, as well as every path sliced out of the CFG.

4.3 Instruction Interpreter

The instruction interpreter is responsible for symbolically executing the program following the decisions made by the decision engine. The instruction interpreter was developed on top of the symbolic execution tool Klee [10].

Klee was mainly developed to generate test cases automatically, but Klee is also capable of searching for runtime errors, including assertion failures. Klee generates a test case every time it reaches the end of an execution path. This input can be used to lead the program's execution along a certain path, and it allows the user to debug errors by using regular debugging tools.

Kite maintains Klee's test case generation, but it only generates a test case when an assertion failure is detected. Furthermore, Kite's interpreter takes advantages of many of Klee's features to implement the CDSE algorithm.

Execution Backtrack: The instruction interpreter starts the execution at the program's entry point, with a single process. Every time the interpreter reaches an `if`-statement, the current process forks after checking if the next branch is feasible. However, the interpreter maintains one child process as a copy of the current program state before it takes any branch. The interpreter saves this copy in a stack, which represents the program states at each decision level.

Thus, the interpreter can easily backtrack to any level by popping all program states until the backtrack level, and by setting the current program state to be the state in this level. Because any program state stored by the interpreter is a copy of another state before a branch is taken, the execution always restarts from an `if`-statement, and this triggers another decision step.

Notice that the interpreter discards all program states that are popped during the backtrack step. However, due to the nonchronological backtrack strategy implemented in Kite, some of these states might still be explored in future iterations. Consequently, Kite is penalized for discarding them, and it might have to re-execute some redundant se-

quence of instructions to restore a discarded state. This penalty could be removed with further optimizations, which are presented in Chapter 7.2.

Instruction Guards: Klee represents every symbolic expression as an abstract syntax tree (AST), which links together the operations to be performed in-order. For that, Klee represents each operation, symbolic variable, and concrete value as a specific class that extends an expression base class. In order to represent instruction guards, Kite includes two new expression classes, one to represent guarded expressions, and one to represent guarded constants. These new types are named *δ-expression* and *δ-constant* respectively.

A δ-expression simulates a `select` operation where a guard δ is the boolean condition, the true side is the guarded expression and the false side is a non-deterministic value. This allows the interpreter to replace an expression by its guarded version in any AST.

The δ-constant is a specialization of the δ-expression and the constant class, which represents concrete values. Thus, a δ-constant stores a concrete value and a list of guards. This type of expression allows the symbolic execution to eliminate queries, to propagate constants, and simplify expressions when the guarded expression is a constant.

The implementation of δ-constants is enough to guarantee constant propagation during expression creation. Because this class extends and re-implements every method of the constant class, it can be used anywhere a concrete value is used. Additionally, a δ-constant always propagates its guards.

For example, given an assignment:

$$\sigma_i = \sigma_j + \sigma_k$$

where both used variables σ_j and σ_k have guarded concrete values. If σ_j is assigned to 10 under guard δ_a, and σ_k is assigned to 5 under

guard δ_b, the interpreter will assign σ_i to 15 under both guards δ_a and δ_b.

The constant propagation triggered by the implied value optimization is described in the following item.

Query Optimizations: Klee implements query optimizations that have great impact on symbolic execution scalability, and can reduce the runtime by more than an order of magnitude [10]. Therefore, Kite also implements these optimizations to show that most of them are orthogonal to the CDSE approach. Kite's interpreter extends the query optimization methods in order to comply with the guards' usage.

The *implied value concretization*, for example, has to propagate the guards involved in the concretization. If the execution adds a constraint such as $\sigma_i = 0$, this optimization replaces every occurrence of σ_i by the value 0. However, before replacing the occurrences of σ_i, this method first checks if the added constraint is guarded. If it is, the method collects all guards, and it creates a δ-constant representing the concrete value 0, which is used instead of the concrete value 0.

The *query cache* adaptation is more complicated. The cache implemented in Klee stores all queries its interpreter sends to the constraint solver together with the computed results. The interpreter consults the cache before sending a new query to the constraint solver. In case of a cache hit, the interpreter can use the cached result instead of invoking the solver.

However, the instruction guards used in Kite can impact in the chance of a cache hit, because similar queries that are originated from different paths could have different guards. Therefore, Kite's cache does not keep any information about guards. Kite implements a lightweight guard removal method, which the cache invokes to generate a copy of a query without its guards. The cache uses this extraction mechanism before looking a query up, or before adding a new entry.

Since the guards are used only during conflict analysis, where a query

is unsatisfiable, Kite's cache only stores satisfiable queries. This restriction allows the cache to safely ignore all guards in a query. In cases where the query is unsatisfiable, CDSE requires information about the conflicting guards, which the cache doesn't provide. This forces the interpreter to call the constraint solver in order to obtain this information.

Although Kite could extend the cache in order to keep the guards involved in a conflict previously found, that could increase the complexity of the cache lookup mechanism, and potentially decrease the cache efficiency. Moreover, the CDSE learning scheme already eliminates many redundant unsatisfiable queries.

The *constraint independence* optimization did not require any change. This optimization determines constraint dependence by analyzing the set of symbolic variables that are included in the leaves of each expression AST. Since the guards never represent a leaf, and this method doesn't change any expression, the guards can be safely ignored.

Therefore, the implemented data structure allows Kite to safely apply all the query optimizations included in Klee. Additionally, it allows Kite to learn facts whenever the interpreter reaches a branch with a concrete condition without calling the constraint solver. If a branch condition evaluates to false, its guards represent the conflict to be learned. If a branch condition evaluates to true, the interpreter knows the non-taken branch is infeasible under the same path; consequently, Kite can add a new learned clause to the fCFG even though the execution did not reach any conflict. Kite derives the learned clause from the condition guards, excluding the guard that represents the last branch taken, and the guard relative to the non-taken branch.

4.4 Constraint Solver

Kite employs the constraint solver to prove whether a certain branch can be taken or not. The instruction interpreter passes a formula to the constraint

solver that defines the branch condition and the current path condition. Besides that, the interpreter also provides a set of assumptions that must be considered during the proof.

Then, the constraint solver determines whether the formula is satisfiable or not. If the formula is satisfiable, the constraint solver produces variable values that satisfy the constraints. The instruction interpreter uses these values if the branch taken leads to a bug; thus, these values represent a valid counter example to an assertion. In cases where the formula is proven unsatisfiable, Kite's constraint solver returns a set of assumptions that led the proof to an unsatisfiable result.

Kite inherits the pair STP / CryptoMiniSat from Klee as the constraint solver employed during symbolic execution to check the feasibility of a branch condition. STP is a constraint solver developed to efficiently solve constraints generated by program analysis, bug finding and test generation tools [21]. STP implements different algorithms based on the abstraction-refinement paradigm to reduce the size of the target formula, and then it employs a SAT solver to check the formula satisfiability. CryptoMiniSat is one of the SAT solvers supported by STP, and it is the one chosen in Klee's implementation. CryptoMiniSat extends MiniSat to support the XOR operation that is common in cryptography [46].

STP receive as an input a quantifier-free formula in the theory of bit-vectors and arrays. STP is capable of determining whether a given formula is satisfiable or not, as well as producing an assignment in the first case. Nevertheless, STP does not support solver assumptions; consequently, STP is not capable of generating a final conflict clause, as required by the CDSE algorithm.

Therefore, Kite's implementation required some modifications to STP's interface and internal processing to support assumptions. Kite's constraint solver extends the STP interface, and it includes methods to create non-deterministic variables, to add assumptions to the proof, and a method to retrieve the conflicting assumptions, in case of an unsatisfiable query.

4.5 Decision Engine

The decision engine is the module responsible for choosing which branch the instruction interpreter should follow. This module includes a SAT solver based on MiniSat[20] to find satisfying assignments to the fCFG whenever they exist.

MiniSat was chosen because it is a highly efficient[9] and robust open-source implementation of an *incremental CDCL* SAT solver, that has a well-defined and clean interface. Because of these characteristics, MiniSat has been widely used[11, 21], and it has also served as the base to many other successful solvers, such as Glucose [2] and CryptoMiniSat [46].

During the initialization, the decision engine receives the program CFG from the preprocessor, and the fCFG. The engine uses the CFG to determine where the execution is, and which literals represent the next branches. The decision engine also maintains a decision stack, with all literals that represent the decisions taken in the current path.

Every time the interpreter reaches an `if`-statement, the interpreter invokes the decision engine to determine which branch it should follow. The decision engine first checks the current assignment to the fCFG to check if both branches could potentially be taken. If that is the case, the engine arbitrarily chooses which one the interpreter should follow.

After a decision, the interpreter might find that the branch chosen is infeasible. In this case, the interpreter passes a conflict clause to the decision engine that adds the clause to the fCFG. The decision engine invokes its SAT solver to determine whether a new path can be taken, and to which level the execution must backtrack. Because MiniSat is incremental, it reuses the previous solving state to solve the new formula. This mechanism reduces the cost of successive calls to MiniSat performed by the decision engine. When the fCFG becomes unsatisfiable, the decision engine informs the interpreter that the backtrack level is 0, which implies that the symbolic execution can't reach any assertion failure.

If a new satisfying assignment is found, the decision engine visits each

[9] MiniSat won all the industrial categories of the SAT 2005 competition.

literal kept in the decision stack starting from level 1 until it finds a literal that was assigned to false. Consequently, the backtrack level corresponds to the level of the last visited literal that is still true, i.e., the engine keeps every decision that is common between the last executed path, and the new assignment.

In order to reduce the backtrack depth, Kite introduces a modification to MiniSat that allows the solver to restart the search from a certain set of decisions. From now on, this modified Minisat will be named *kMiniSat*.

Besides assumptions, kMiniSat receives a vector of literals that should be assigned to true if possible. Thus, before starting the regular search, kMiniSat includes a preprocessing step that initiates the search by propagating these preferred decisions, whenever possible. Preprocessing finishes if the preferred decisions were propagated, or if a conflict is detected.

In contrast to assumptions, the decisions derived from this vector are not mandatory, and the solver is free to revert them during the regular search loop.

Kite's decision engine may use this new mechanism to try to guide the SAT solver to search a new path similar to the current path taken. For that, the engine may provide the literals respective to the branches taken as the preferred decisions. Otherwise, the engine passes an empty vector, which leave the SAT solver free. Kite implements three different strategies to determine when to guide the SAT solver, and when to let it free:

Black Box: In this strategy, Kite always leaves the SAT solver free to decide, i.e., no preferred assignment is given, and the SAT solver is treated as a black box.

Always Guide: This is the opposite strategy, where Kite always tries to minimize the backtrack level as much as possible. Thus, the decision engine always sets kMiniSat preferred decisions as the last set of branch literals followed.

Maximize Learning: Following the intuition that the smaller the learned clause, the more paths it will potentially prune, this strategy chooses

when to guide kMiniSat according to the size of the last learned clause. The decision engine guides kMiniSat if the learned clause size relative to the path size is smaller or equal to a constant k.

4.6 Known Limitations

Kite is still a prototype, developed to assess the conflict-driven symbolic execution algorithm. Therefore, it is not mature software, and it has some known limitations. These limitations are not inherent to CDSE, but are the result of development time constraints. Kite's main limitations are the following:

Memory Handling: Using Klee and LLVM's structure to implement the proposed technique had one main drawback, memory handling is usually conservative; thus, global memory and pointers are harder to reason about. The absence of phi-nodes and lack of pointer analysis impact on the conflict analysis, making it less efficient and requiring special handling.

Because the use-def analysis implemented in LLVM doesn't include memory accesses, Kite cannot predict which join points in the CFG could potentially change the value of a memory location. Consequently, Kite guards the expression generated from a memory read with the guards that correspond to all branches taken between the read and the write operations. Since this approach over-approximates the possible interference other paths could have in a value read from a memory position, this approach is safe, but not very efficient.

Therefore, Kite's preprocessing step always includes the LLVM optimizations passes that reduce the number of memory accesses. Chapter 7.2 suggests and discusses other preprocessing techniques to delay memory writes, anticipate memory reads, or use memory shadows to improve CDSE learning.

Symbolic Pointer Dereference: Kite does not support dereference of pointers with symbolic values. Whenever a pointer is dereferenced, Kite

checks whether the pointer's value is constant or not. In the former case, the instruction execution can access only one memory object through this pointer. In the latter, more than one memory object could be reached through this pointer, and this would require some adaptations in the expression generation, or in the CDSE learning scheme.

Originally, Klee forks its execution whenever it reaches a pointer dereference which may result in different memory access. Klee creates a new process for each memory position that can be accessed. In order to reuse Klee's approach, Kite could include a fCFG variable v_i associated with which memory position it chose to follow, and guard the dereference command with a guard δ_i. This would allow Kite to learn and backtrack to this decision, and explore paths with different memory dereference.

Another solution to this problem would be the usage of conditional accesses inspired by model checking. The result of the dereference operator '*' is the value of a variable σ, if the pointer's value is equal to the address of σ — the reference $\&\sigma$. For example, if a pointer ρ may hold the address of two variables σ_i or σ_j, the dereference of ρ would be decoded as:

$$*\rho \implies (\rho = \&\sigma_i) \; ? \; \sigma_i \; : \sigma_j \tag{4.1}$$

Furthermore, this change could be applied statically by a code transformation pass executed during preprocessing. Different pointer analysis could be applied to restrict the number of variables that could be dereferenced in a certain instruction.

Function Pointer Call: The same intuition behind symbolic pointer dereference could be applied to encode function pointer calls. The control flow transition will follow a certain path depending on the value of the pointer's value, i.e., it can be translated as nested `if`-statements that

compares function's addresses to the pointer's value. Once more, Kite could replace every function call by applying a code transformation pass together with a pointer analysis during preprocessing stage.

Chapter 5

Results

This chapter presents practical evidence that conflict-driven symbolic execution (CDSE) can perform better than regular symbolic execution, as well as model checking. For that, Kite is compared to the symbolic execution tool Klee [10], and to the model checker CBMC [11].

Additionally, this chapter presents the impact of the decision heuristics on Kite's performance on the benchmarks used. These results support the choice of Kite's best overall configuration.

5.1 Testing Environment

In order to assess Kite, all tests were performed using a benchmark composed of the `ntdrivers-simplified` and `ssh-simplified` instances retrieved from the 2013 Competition on Software Verification (svcomp13) [4]. The tests were slightly modified to comply with Klee's standards and limitations, without breaking CBMC compatibility. The non-deterministic functions were renamed, and every loop was bounded to 50 iterations. In these benchmarks, the ERROR label represents a bug, and the label was replaced by an `assert(false)` statement.

Because the name of the tests can exceed 50 characters, the tests were renamed to make all tables and graphs more readable. The renaming followed the test classification according to the existence of a property violation. A test can either be considered *safe* or *unsafe*. The former indicates that all assertions should hold, while the latter indicates that the test has a bug, and at least one assertion is violated. Thus, the tests were renamed to either safe_n or unsafe_n, where n is a unique identification number. Appendix A presents a table with the mapping between the original test name and their

simplified name.

The CBMC version used was 4.7, which corresponds to the latest release by the date the tests were performed. Since Klee doesn't have any officially numbered releases, every test was performed using two different changesets. The first version corresponds to Klee's changeset from which Kite was forked[10], and the second one corresponds to Klee's latest changeset by the time the tests were executed[11]. However, this chapter only presents the results obtained by the forked version, because this version performed better than the latest version. Additionally, the forked version is closer to Kite's implementation. See Appendix A for the performance obtained using Klee's latest changeset.

Additionally, this chapter presents many different versions of Kite according to the different decision heuristics employed (see Section 4.5 for definitions). This chapter will use the following names to differentiate each heuristic:

Kite K_G: Represents the Always Guide strategy.

Kite K_B: Represents the Black Box strategy.

Kite $K_{M(k)}$: Represents the Maximize Learning strategy, where a constant k is given as the threshold parameter. For example, $K_{M(10)}$ corresponds to the heuristic that guides the SAT search if the last learned clause size is smaller or equal to 10% of the explored path size (in the number of branches taken).

All the results presented were obtained from a single machine that has an Intel(R) Core(TM) i7-2600K processor and 16GB memory. The machine's operating system was openSUSE version 12.1. Additionally, the tests had no memory limit, but they had a time limit of 200s per test.

[10]Changeset 9b5e99905e6732d64522d0efc212f3f1ce290ccc from September 12th, 2012
[11]Changeset e49c1e1958e863195b01d99c92194289b4034bbb from January 21st, 2014

5.2 Overall Evaluation

This first analysis compares Kite's performance to Klee's [10] and CBMC's [11] performance. As will be discussed in Section 5.3, engine $K_{M(10)}$ had the best performance overall; thus, this engine is used by default, and it is the one presented in this section.

Kite and Klee were executed with the following switches:

`--optimize:` Runs LLVM optimization passes before symbolic execution.

`--exit-on-error:` Aborts symbolic execution whenever an error is found.

`--no-output:` Doesn't generate test cases for each complete path explored.

CBMC was run using the following switch:

`--no-unwinding-assertions:` Doesn't generate unwinding assertions for the program loops.

These switches were chosen to optimize performance, and to mimic the same behavior for all tests. Every tool was set to run until it found an assertion failure, or proved that all assertions hold.

All tools concluded the verification before the time limit established, and every result was consistent to the properties safety. No false errors were detected in any safe test, and every tool reported one assertion violation in every unsafe test.

Table 5.1 presents the time taken by each tool to solve each safe instance, and the total time to solve the entire safe test set. In this test set, Kite had the fastest solving time ($203.55s$), and it was $1.55\times$ faster than Klee ($315.60s$). CBMC presented the worst performance, and it spent $1410.99s$ to solve the test set; thus, CBMC was $6.93\times$ slower than Kite and $4.47\times$ slower than Klee.

Even though the focus of this thesis is on the impact of adding learning capability to symbolic execution, it is important to highlight that both symbolic execution tools performed better than the model checking tool. As seen in the table 5.1, Klee and Kite were superior to CBMC in all instances.

Test	**CBMC**	**Klee**	**Kite**
safe_01	1.35	**0.16**	0.28
safe_02	0.41	0.45	**0.17**
safe_03	0.30	0.06	**0.04**
safe_04	0.51	**0.14**	0.25
safe_05	0.11	**0.02**	**0.02**
safe_06	0.16	0.04	**0.02**
safe_07	52.51	**0.93**	1.28
safe_08	53.92	**0.92**	1.26
safe_09	67.37	**0.95**	1.25
safe_10	54.45	**0.92**	1.27
safe_11†	169.76	68.58	**11.60**
safe_12	2.82	**0.09**	0.61
safe_13	0.61	**0.09**	0.13
safe_14†	157.06	40.72	**18.76**
safe_15†	158.55	40.36	**15.91**
safe_16†	159.71	**40.39**	42.01
safe_17†	184.12	40.62	**34.36**
safe_18†	172.86	40.30	**38.33**
safe_19†	174.41	39.86	**36.00**
Total	1410.99	315.60	**203.55**

Table 5.1: Table showing the execution time (in seconds) spent by Klee, CBMC and Kite to prove that each test case respects their assertions. Since every tool spent more than 10s to solve the 7 instances marked with a '†', they will be classified as hard, in contrast to the other instances, classified as easy, that could be solved in less than 1s by at least one tool.

This shows that using symbolic execution to prove properties given an upper loop bound is feasible, and that symbolic execution can be more efficient in certain cases.

Comparing Kite against Klee, the former performed significantly better in almost all hard instances — where every tool spent more than 10s to solve. Kite was The greatest speed-up in these instances happened while solving test *safe_11*, where Kite was 5.9× faster than Klee. Kite was slower than Klee only in one hard instance, *safe_16*, but the time difference was of only 4%.

However, Kite did not outperform Klee in the easy instances. The two main reasons for this behavior are Kite's initial overhead, and Kite's backtracking implementation. First, the time spent by Kite's preprocessor depends on the program control flow graph size, and it is significant only when the total solving time is small. Besides that, the cost of backtracking past unblocked branches may overcome the benefit of eliminating some paths from the entire analysis (refer to Section 4.3 at page 50 for more details). Section 7.2 suggests an optimization that can be applied to reduce the number of instructions executed due to backtracking.

In the unsafe instances, Klee had the best overall performance, as shown in Table 5.2. Klee took 4.23s, while Kite spent 5.50s and CBMC spent 1602.30s. Both, Kite's and Klee's search heuristics were efficient at finding the assertion violations, and they solved every instance in less than 2s. As mentioned before, in instances where the solving time is short, Kite's implementation overhead had a significant impact on the solving time.

CBMC had worse performance than Kite and Klee. This result supports the intuition that symbolic execution is usually faster than model checking to find property violations. This big difference is also a consequence of the CBMC loop unrolling strategy. CBMC unrolls every program loop to their upper bound prior to solving the instance. Therefore, it generates a big formula, even though the assertion violation might be reached by executions that perform fewer loop iterations. Because of that, CBMC's performance is closely related to the loop bound provided by the user.

Overall, Kite had the best performance in the benchmarks used. As shown in Table 5.3, Kite solved all instances in 209.05s, while Klee spent 319.83s, and CBMC took 3013.29s.

Besides the improvement in symbolic execution performance, the benefit of CDSE becomes clearer when viewed in terms of the number of instructions symbolically executed. This number can be reduced when the engine prunes paths out of its analysis. Figure 5.1(a) and 5.1(b) depict the impact of Kite's learning scheme in the number of instructions symbolically executed in safe and unsafe instances, respectively.

Despite the backtracking cost, Kite still tends to execute fewer instruc-

Test	CBMC	Klee	Kite
unsafe_01	0.31	0.24	**0.03**
unsafe_02	0.49	0.14	**0.06**
unsafe_03	0.18	0.04	**0.02**
unsafe_04	1.83	**0.07**	0.19
unsafe_05	130.86	0.04	**0.03**
unsafe_06	178.63	**0.62**	0.78
unsafe_07	197.32	**1.60**	1.62
unsafe_08	185.79	**0.27**	1.35
unsafe_09	172.38	0.15	**0.05**
unsafe_10	167.25	0.09	**0.06**
unsafe_11	154.35	0.10	**0.07**
unsafe_12	52.92	**0.13**	0.46
unsafe_13	183.94	**0.04**	**0.04**
unsafe_14	67.48	**0.18**	0.43
unsafe_15	54.34	0.14	**0.06**
unsafe_16	54.23	0.38	**0.25**
Total	1602.30	**4.23**	5.50

Table 5.2: Table showing the execution time (in seconds) spent by Klee, CBMC and Kite to find a bug in each test case.

tions than Klee, for both instance types. In all hard instances, Kite executed fewer instructions, and this gain came close to one order of magnitude. See Appendix A for a complete table with the exact number of instructions symbolically executed by Klee and Kite.

5.3 Evaluation of the Decision Engines

The decision engine plays an important role in any conflict-driven clause learning (CDCL) approach, and the same applies to CDSE. Tables 5.4 and 5.5 show the different performances for safe and unsafe instances obtained by changing Kite's decision heuristic.

Engine *All Guide* (K_G) had the best performance in the safe test set. This engine tries to minimize the number of levels backtracked by guiding its SAT solver; consequently, it reduces the penalty of discarding unblocked

Instances Type	CBMC	Klee	Kite
Safe	1410.99	315.60	**203.55**
Unsafe	1602.30	**4.23**	5.50
Total	3013.29	319.83	**209.05**

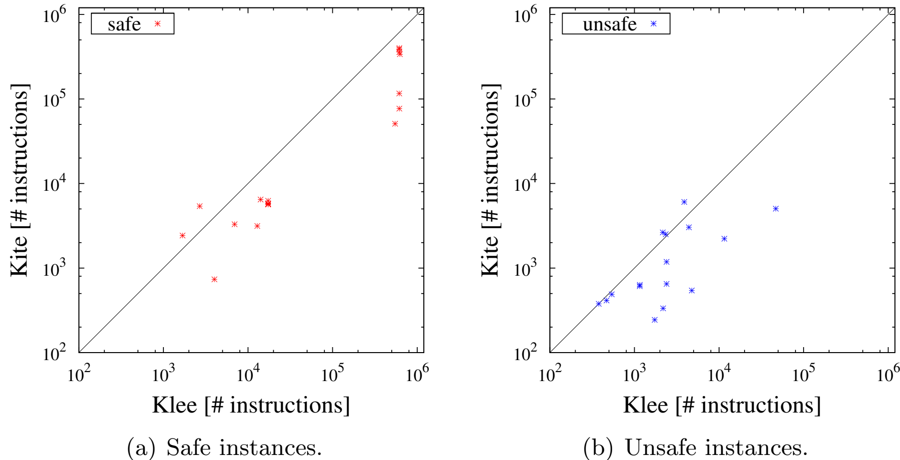
Table 5.3: Total time spent (in seconds) by each tool to solve the benchmarks. Overall, Kite had the best performance, and CBMC had the worst performance.

processes. However, this engine tends to explore the same search space for a long time. Therefore, this strategy impacted the time to find the assertion violation in test *unsafe_7*.

In contrast to K_G, the engine derived from the *Black Box* strategy (K_B) never guides the SAT solver. Because the SAT solver used has no knowledge about the branch precedence, nor which path was taken, this engine backtracking strategy is more random. Thus, this engine tends to explore different parts of the execution tree, and pays a high price for discarding many unblocked processes. This impacted the engine performance in all hard safe instances. This engine also did not perform well in the test *unsafe_7*, although it was almost 10× faster than the other engines in the *unsafe_8* instance.

The *Maximize Learning* strategy tries to balance the cost and benefit of guiding the SAT solver during CDSE. This heuristic chooses when to guide the SAT solver according to the "quality" of the search space. The engine quantifies the quality of the search space by computing the relation between the learned clause size and the current path size. The search space is classified as good enough, if this relation is smaller than a given threshold k.

Three different thresholds were tested: 5%, 10% and 15%. The higher the threshold, the more often the decision engine tends to guide the SAT solver. Thus, the engine $K_{M(15)}$ had a behavior similar to the K_G engine. Between the three Maximize Learning engines, $K_{M(15)}$ had the best performance in the safe instances, but spent a significant amount of time to solve *unsafe_7*.



(a) Safe instances.

(b) Unsafe instances.

Figure 5.1: Number of instructions symbolically executed by Klee and Kite on safe and unsafe instances. As expected, Kite symbolically executed fewer instructions in the majority of the tests, even in unsafe instances where Klee has better performance.

The poor performance of $K_{M(15)}$ on *unsafe_7* had a large impact on its overall performance, and this one long runtime made $K_{M(15)}$ overall worse than either $K_{M(10)}$ (the best performer) or $K_{M(5)}$. The $K_{M(10)}$ engine was also superior to $K_{M(5)}$. In fact, $K_{M(10)}$ overcame $K_{M(5)}$ in the safe and unsafe test sets.

Therefore, the best overall result was obtained using the Maximize Learning strategy with 10% threshold, $K_{M(10)}$. Although this engine was not the best in the safe instances, $K_{M(10)}$ still performed well on them. Moreover, this engine was the best one in the unsafe test set by a significant difference. $K_{M(10)}$ was more than 2× faster than the other engines in the unsafe instances.

Notice that the tests shown in this section is limited to two benchmarks, although $K_{M(10)}$ had the best performance in these benchmarks, no conclusion can be held about other instances. Further testing is necessary to establish a better pattern, and to establish which engine is better in which cases.

Test	$\mathbf{K}_B$	$\mathbf{K}_G$	$\mathbf{K}_{M(5)}$	$\mathbf{K}_{M(10)}$	$\mathbf{K}_{M(15)}$
safe_01	0.62	**0.27**	0.30	0.28	**0.27**
safe_02	0.34	0.19	0.25	**0.17**	0.18
safe_03	**0.04**	**0.04**	**0.04**	**0.04**	0.05
safe_04	0.76	**0.25**	**0.25**	**0.25**	**0.25**
safe_05	**0.01**	**0.01**	**0.01**	0.02	0.02
safe_06	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**
safe_07	2.02	1.30	1.35	1.28	**1.27**
safe_08	2.31	1.33	1.31	**1.26**	1.46
safe_09	1.98	**1.25**	1.29	**1.25**	1.30
safe_10	2.08	**1.25**	1.32	1.27	1.27
safe_11	33.26	**11.57**	11.78	11.60	12.66
safe_12	0.58	**0.56**	0.57	0.61	0.58
safe_13	0.17	**0.12**	0.13	0.13	0.13
safe_14	46.48	**16.93**	17.26	18.76	19.07
safe_15	41.42	**15.41**	16.75	15.91	17.17
safe_16	71.25	**31.73**	44.70	42.01	40.14
safe_17	64.35	**24.91**	37.10	34.36	30.22
safe_18	58.39	**25.68**	40.74	38.33	34.51
safe_19	67.48	**27.28**	39.36	36.00	32.41
Total	393.56	**160.10**	214.53	203.55	192.98

Table 5.4: Table showing the effect of different decision engines on Kite's execution time (in seconds) to verify the safe instances.

Test	$\mathbf{K}_B$	$\mathbf{K}_G$	$\mathbf{K}_{M(5)}$	$\mathbf{K}_{M(10)}$	$\mathbf{K}_{M(15)}$
unsafe_01	0.05	**0.03**	0.04	**0.03**	0.04
unsafe_02	0.07	**0.06**	**0.06**	**0.06**	**0.06**
unsafe_03	0.03	**0.02**	**0.02**	**0.02**	**0.02**
unsafe_04	0.52	0.19	**0.11**	0.19	0.19
unsafe_05	0.05	**0.03**	**0.03**	**0.03**	**0.03**
unsafe_06	**0.49**	0.55	0.74	0.78	0.56
unsafe_07	Timeout	Timeout	6.77	**1.62**	27.88
unsafe_08	**0.17**	1.35	2.73	1.35	1.36
unsafe_09	0.06	0.05	**0.04**	0.05	**0.04**
unsafe_10	**0.50**	0.06	0.07	0.06	0.06
unsafe_11	0.52	0.09	0.09	**0.07**	0.08
unsafe_12	**0.37**	0.45	1.09	0.46	**0.44**
unsafe_13	0.04	0.04	**0.03**	0.04	0.04
unsafe_14	0.69	**0.43**	0.45	**0.43**	**0.43**
unsafe_15	0.59	**0.06**	0.07	**0.06**	**0.06**
unsafe_16	0.44	0.46	**0.08**	0.25	0.24
Total	>204.59	>203.87	12.42	**5.50**	31.53

Table 5.5: Table showing the effect of different decision engines on Kite's execution time (in seconds) to find a property violation in the unsafe instances. In the instance *unsafe_07*, two engines did not finish before the 200s timeout. Thus, the total time reported for these engines are not precise.

Chapter 6

Related Work

There is a vast literature on software verification which goes beyond the scope of this thesis to survey. This chapter considers only works that aim to reduce the number of paths explored during symbolic execution. To the best of my knowledge, there are only other four approaches reported in the literature that can successfully reduce this number. All of them learn facts once a conflict is reached, i.e., when a certain branch is proven infeasible under a certain path condition. These approaches, however, differ on which facts are learned, and how they employ them in order to prune paths from the search.

This chapter describes these approaches, and contrasts their learning schemes to the one introduced in this thesis. These approaches are described in two sections: the first section introduces learning schemes based on interpolants, while the second section describes approaches based on conflict clauses, which are closer to conflict-driven symbolic execution (CDSE).

Before presenting the different learning schemes, it is important to highlight the distinction between *early conflict detection* and *conflict preemption*. In both cases, the symbolic execution engine stores all the knowledge it acquires from previous executions in a database, and uses this knowledge to prune the search space. However, these scenarios differ according to how the engine can employ this knowledge.

Early Conflict Detection: In early conflict detection, the symbolic execution engine uses its knowledge to interrupt the symbolic execution of a path, before reaching an infeasible branch. In other words, the engine uses the database to constrain the symbolic execution, and may *trigger a conflict while symbolically executing a feasible branch.* There-

fore, the engine can determine if the next branch would never lead the current exploration into a target violation.

Conflict Preemption: In conflict preemption, the symbolic execution engine uses its database to restrict the set of paths that can be followed. Thus, the database offers a mechanism to avoid the paths that would trigger a known conflict. Once the engine starts executing a new path, *the engine will not hit a conflict that is kept in the database.* It will either find a valid counterexample, or it will reach an infeasible branch, which represents a new conflict.

For example, Section 3.1 explains how CDSE explores only two execution paths to prove that the program in Listing 1.1 (at page 5) does not violate its embedded assertion, even though there are 8 feasible paths that lead the program to the execution of the assertion statement. Consider the program control flow graph (CFG) depicted in Figure 6.1, after exploring paths $\{b_1, b_3, b_4, b_6, b_7, b_9, b_{10}\}$ and $\{b_1, b_3, b_4, b_6, b_8, b_{10}\}$ CDSE learns the following clauses:

$$(\neg b_7 \vee \neg b_9 \vee \neg b_{10}) \wedge (\neg b_8 \vee \neg b_{10})$$

Each learned clause is derived from a unique conflict.

Notice that these two clauses alone are not sufficient to prove the property's correctness without exploring other paths — the two clauses by themselves form a satisfiable formula. However, because the CDSE engine initially populates its database with the constraints extracted from the CFG, it can conclude that there isn't any feasible path that will trigger the assertion failure in this case. Thus, the CDSE engine uses the fCFG to prune all paths that violates a certain learned clause. CDSE can preempt which decisions would lead the search to hit a known conflict, and it will *never try to follow conflicting decisions.*

However, the same is not true for a learning scheme that can still learn only these two clauses, but that does not integrate any other knowledge about the CFG. Such an engine doesn't know yet that all paths that lead the execution to b_{10} have to include either b_7 and b_9, or b_8. Thus, it has to

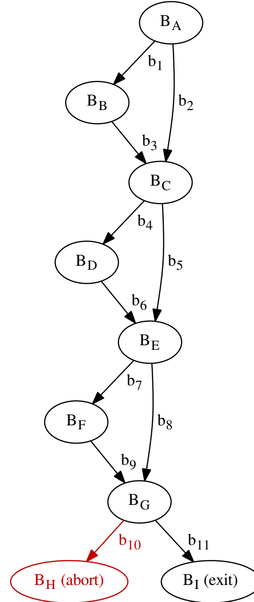


Figure 6.1: The control flow graph for the program described in Listing 1.1, where basic block B_H represents the assertion failure.

continue exploring the search space to guarantee that the assertion failure can't be reached. Nevertheless, these two clauses can still be used to detect a conflict before reaching an infeasible branch. For example, if the engine follows path $\{b_1, b_3, b_5, b_7, b_9\}$, these decisions are enough for the engine to know that b_{10} must be false. Thus, it will be able to detect the conflict after executing B_G, but before checking if the branch b_{10} is feasible.

6.1 Interpolation-Based Learning Schemes

Given a pair of logical formulas φ_A and φ_B, where $\varphi_A \wedge \varphi_B$ is unsatisfiable, a Craig interpolant [13] is a set of constraints I with the following properties:

1. $\varphi_A \implies I$
2. $I \wedge \varphi_B$ is unsatisfiable.
3. I contains only variables from $\varphi_A \bigcap \varphi_B$

The interpolant I can be used to abstract φ_A by keeping the invariant that the conjunction with φ_B cannot be satisfied. Therefore, interpolation-based approaches extract interpolants from unsatisfiable queries, where a certain branch condition β is infeasible under a path condition pc, and use them to block state transitions that trigger conflicts previously learned.

There are two interpolation approaches described in the research literature, Lazy Annotation [34] and Abstraction Learning [26]. These approaches were developed independently, but both roughly implement the same algorithm to verify loop-free programs. The algorithm description below will follow the Lazy Annotation implementation.

For loop-free programs, Lazy Annotation proceeds as follow. The engine symbolically executes paths in a depth-first manner, and it follows a path until a conflict rises, or until a violation is reached. The interesting case happens when there is a conflict. In this case, the engine extracts an interpolant that summarizes a reason for this conflict, and that blocks the current execution state. The algorithm annotates the infeasible edge with the extracted interpolant, and backtracks to a non-blocked state. Whenever the execution reaches an edge with some annotation, the engine uses the annotated interpolant I to determine if pc will trigger a conflict detected in previous iterations. If $pc \implies I$, the pc would trigger the same conflict used to generate I, i.e., I blocks the current execution state; thus, the engine interrupts the current search, and backtracks.

If the engine detects a new conflict in an edge already annotated, the engine extracts a new interpolant, and extends the previous annotation creating a disjunction with the new interpolant. Moreover, whenever all outgoing edges of a node are infeasible under a certain path, the engine extracts a new interpolant for the given node. This interpolant is derived from the conjunction of the conditions that block the outgoing edges.

On one hand, interpolants offer a safe abstraction that summarizes why certain path conditions lead the program execution into a conflicting state. Because the interpolants express path conditions that may trigger a conflict in certain parts of the code, interpolants offer a more fine grained abstraction than the conflict clauses extracted in CDSE. Thus, they can potentially

detect conflicts that CDSE can't.

Additionally, interpolation can be used to perform unbounded analysis. As shown by McMillan [35], interpolation provides an approximate image operator, which can be used iteratively to compute an over-approximation of the set of reachable states that does not trigger a property violation. If this computation reaches a fixpoint, the over-approximate set is an inductive invariant that proves that the property won't be violated by any reachable state. This unbounded analysis in the context of symbolic execution is better explored in Abstraction Learning, which is capable of proving correctness of programs with loops in a reasonable amount of time [26].

On the other hand, neither interpolation-based approach is capable of reasoning about the restrictions represented in the CFG, and they cannot preempt future conflicts as CDSE does. In Lazy Annotation, and in Abstraction Learning, the execution identifies a blocked state only when the execution reaches a branch where the current state implies the annotated interpolant. The engine has to visit every feasible edge at least once.

The example shown in Listing 6.1 retrieved from the Lazy Annotation paper [34] shows an example on how the interpolation-based approaches employ their knowledge to early conflict detection.

For this example, Lazy Annotation explores three different conflicts, and one blocked path, in order prove that the `error` function is infeasible. Following the original description [34], the algorithm could start by choosing to execute the path that visits `L3` and `L7`. The first conflict is detected when the engine tries to follow branch `L10` $\rightarrow$ `L11`. The algorithm would backtrack to node `L6` and annotate `L6` $\rightarrow$ `L7` with label p.

Then, a new conflict is detected while trying to execute `L6` $\rightarrow$ `L9`. The engine annotates this edge with the label a, and annotates `L6` with $p \wedge a$. The engine backtracks until `L2`, and it executes the third path via `L5` until it reaches `L6`.

The engine doesn't follow the branch `L6` $\rightarrow$ `L7`, since it is blocked by label p, and the current path constraints implies p. Then, the engine follows only the branch `L6` $\rightarrow$ `L9`. It eventually triggers a conflict while trying to reach `L11`. Thus, it will label `L6` $\rightarrow$ `L9` with p. Location `L6` is also labeled

```
diamond:
L1   assert(p);
L2   if (*)
L3       a = 1;
L4   else
L5       a = 0;
L6   if (a)
L7       x = x + 1;
L8   else
L9       x = x - 1;
L10  if (!p)
L11      error();
```

Listing 6.1: A program fragment retrieved from [34]. In this example, the `error()` function represents an assertion(false), which cannot be reached during any program execution. The assert(p) statement states that every execution considered in this example should assume that `p` is initially set to true.

with p, since `L6`'s label is derived from the disjunction of its old label $p \wedge a$ and p.

This example shows how Lazy Annotation can prune the search space by detecting a conflict once it reaches `L6` through `L5`, even though the edge `L6` $\rightarrow$ `L7` is feasible. However, this is also a good example that shows the benefit of integrating structural information about the program to the learning scheme. Notice that `L11` is only reachable if $!p$ is true in `L10`, but the variable p is never modified, and its value comes from statement in `L1`. Thus, it is possible to prove this property by executing only one single path.

6.2 Clause Learning Schemes

In the same direction taken in this thesis, [30] and [25] also present symbolic execution approaches that have similar structure to CDCL SAT solvers. At a high level, both approaches are closely related to CDSE, since they also summarize a conflict found as a function of the branches taken in the current executed path. Unfortunately, both clause learning approaches do not have a publicly available implementation; therefore, it was not possible to include

them in the tests presented in Chapter 5.

The solution presented by Krishnamoorthy, Hsiao, and Lingappan [30], explores the program in a depth-first search manner, and implements nonchronological backtracking. Whenever a conflict is detected, this approach applies conflict analysis included in its constraint solvers to detect which assignments in a query (the conjunction of path condition pc with a branch condition β) are responsible for the conflict. The engine has to map the conflicting assignments to the branch conditions where they were made.

Then, the algorithm learns how to avoid this conflict, and backtracks to the most recently visited node that does not imply this conflict, and that has a non-blocked outgoing edge. This approach does not encode the CFG as a CNF, and it does not use a SAT solver to enumerate paths. Consequently, it only detects a conflict when the engine selects a new branch to explore.

After choosing which branch to take next, the engine checks if the path violates one of the clauses in the database. If a violation is detected, the engine does not follow the chosen branch. Instead, the engine tries to take the other branch that originated in the current location. If all branches in a location under the current path are blocked, the engine backtracks. Otherwise, it continues executing the branch chosen.

Because the learned clauses are checked at every decision, this algorithm is able to perform early conflict detection. However, this learning scheme only stores the constraints that were learned during conflict analysis; consequently, the engine does not employ any knowledge about the unblocked paths that are available in a certain search space, and it cannot preempt known conflicts.

The algorithm description is not clear about how conflicting assignments are mapped to the branches. Additionally, this approach does not apply any other information about the CFG to reduce the number of assignments that can be considered conflicting. The encoding may have a negative impact the strength of the clauses learned.

The Satisfiability Modulo Path Program [25] technique, or just SMPP, is the other approach that learns facts about the branches taken in a path that led the execution into a conflict. For loop-free programs, SMPP resem-

bles CDSE, since SMPP also encodes the program CFG as a conjunctive normal form (CNF) formula, and it also uses a SAT solver to enumerate the paths. For each satisfying assignment, the engine follows the branches that determine the chosen path, and the execution stops whenever an infeasible branch, or an error is found. In the case of an infeasible path, the algorithm also learns a clause that forces the algorithm to avoid the refined set of conflicting branches. Thus, SMPP can also preempt conflicts in the same way that CDSE does.

SMPP has been proposed for different verification approaches, and extended to symbolic execution. Despite the similarity between SMPP and CDSE, these algorithms have subtle differences, that can greatly impact the algorithm's performance and efficiency in practice, such as:

Conflict Analysis : One key difference between SMPP and CDSE is the conflict analysis procedure. SMPP requires the constraint solver to determine an unsatisfiability core[12] for an unsatisfiable query. Additionally, the SMPP engine has to map each conflicting assignment returned by the constraint solver, to the branch previously taken (conditional or unconditional). Then, the algorithm performs an interference analysis to refine the set of branches responsible for the conflict. This analysis determines which other paths intersect with the current one, which could result in a different set of assignments that might not lead the execution to trigger the same conflict.

Therefore, SMPP demands an interference analysis every time a new conflict is detected, while CDSE doesn't. CDSE avoids the extra work because the symbolic encoding includes some information about the program structure. Additionally, this encoding leverages SSA analysis to reduce the information included, and consequently reduces the conflict clause detected by the constraint solver. Because CDSE uses the assumption mechanism implemented by incremental constraint solvers, it also does not require the constraint solver to determine the unsat-

[12] As defined in Section 2.3.3, given an unsatisfiable formula φ, an unsatisfiability core is a sub-formula φ', $\varphi' \subseteq \varphi$, that is also unsatisfiable

isfiability core of a conflicting query.

Query Optimizations: In CDSE, the instruction guards provide a clean way to implement all query optimizations included in modern symbolic execution tools. As described in Section 4.3, these optimizations are applied not only to accelerate the solving process as in Klee. Constraint simplification techniques often find that in certain paths some branch conditions are constant. In these cases, the guards mechanism allows the CDSE engine to learn what caused the condition to be constant. The engine uses such information to learn facts even before attempting to take an infeasible branch, i.e., before the execution reaches a conflict.

***f*CFG encoding:** Another difference between SMPP and CDSE is the CFG formula encoding. SMPP encodes the CFG formula to be satisfied only if the variables assigned to true represent one single path. In other words, the constraints added to the CFG are stronger, and a node variable can be assigned to true, only if *exactly one* of its predecessors is true, and if *exactly one* of the node's successors is also true. Therefore, these constraints are translated to `XOR` functions. Since the description of XOR functions with n variables in CNF requires 2^{n-1} clauses, SMPP encoding result in a bigger and more complex CNF formula.

Loop Handling: For programs with loops, the SMPP can use an alternative CFG without loops by reducing the original graph using Maximal Strongly Connected Components. Although they suggest loop unrolling to deal with loops in symbolic execution, their implemented approach performs an over-approximated symbolic execution, where all the assignments inside a loop are replaced by non-deterministic values.

Chapter 7

Conclusion

The classic symbolic execution algorithm introduced by King [28] suffers from a problem known as path explosion. Assuming that a program to be verified has a finite execution tree, symbolic execution explores every feasible execution path. In the worst case scenario, the number of execution paths is exponential to the number of conditional branches in a program.

As presented in Chapter 6, there are just a few approaches that address this problem, and most symbolic execution tools still explore the entire execution tree. This thesis presents a novel symbolic execution algorithm that successfully reduces the number of paths necessary to prove property correctness.

7.1 Contributions

This thesis's main contribution is a novel algorithm that can dynamically reduce the number of paths explored during symbolic execution. This algorithm is capable of learning from conflicts detected while symbolically executing a path. Because this algorithm was inspired by the insights introduced in conflict-driven clause learning (CDCL) SAT solvers, I named it Conflict-Driven Symbolic Execution (CDSE).

The algorithm presented in this thesis has many interesting properties that makes it a unique symbolic execution extension. These properties are outlined bellow:

Lightweight learning scheme: The CDSE learning scheme requires just a few changes to symbolic execution, and it takes advantage of the compiler's SSA format to reduce the number of guarded instructions. Ad-

ditionally, this learning scheme leverages incremental solver techniques to generate lightweight conflict analysis, and to efficiently solve the control flow graph formula (fCFG) after adding new learned clauses.

Small CFG formula encoding: In contrast to the SMPP algorithm [25], CDSE does not use XOR operations to encode the program control flow graph (CFG). Consequently, the CDSE encoding (fCFG) can be easily satisfied, and the size of the fCFG is linear to the size of the CFG.

Error preemption: Because CDSE combines the CFG structure and the facts learned as a unique formula, CDSE can use a SAT solver to restrict the search. Consequently, CDSE can preempt conflicts already learned, and it only executes paths that have not been blocked yet.

Learning without conflicts: The CDSE algorithm can leverage constant propagation method not only to eliminate queries, but also to learn potential conflicts without hitting them. A constant branch condition always implies that one direction cannot be taken; thus, the CDSE engine can learn why one direction is infeasible under the current constraints, even if this direction is not the one being followed.

Another important contribution of this thesis work is the proof-of-concept CDSE tool Kite. Kite is an open-source tool, which can verify software described in C. Furthermore, Kite can be used in the development of future work.

I developed Kite in order to assess the CDSE algorithm, and to compare it to existing verification techniques. The results presented in Chapter 5 give empirical evidence that CDSE can be better than regular symbolic execution in proving property correctness. These results also show that there is more space for improving the CDSE algorithm.

7.2 Future Work

As future work, this thesis suggests the following topics that can improve the CDSE algorithm:

Decision Heuristics: The results in Section 5.3 show how the engine efficiency is highly dependent on the decision heuristic. Kite introduced a small change in the SAT solver included in the decision engine, that allows the engine to induce the SAT solver to start its search under a similar path as the last one executed. This small change was associated with a mechanism to choose when this induction should be applied. The engines derived from this change performed better than leaving the SAT solver free to decide the fCFG assignments.

CDSE may still benefit from more sophisticated approaches. For example, one interesting research direction would be changing the SAT solver decision heuristic to prioritize assignments to branch variables that are closer to the entry node.

Additionally, the decision heuristics introduced in this thesis could benefit from more diverse test cases. This would improve the heuristic's tuning, and it may provide more information about their strong and their weak points.

Loop Handling: Section 3.5 presents two techniques to handle loops, the static and dynamic CFG unrolling. Another interesting approach would be combining CDSE with some unbounded model checking technique. In the same way that the dynamic CFG unrolling uses assumptions in order to detect which loops may influence a property's proof, this mechanism can be used to find which loops are relevant to a property's proof. For these loops, CDSE could integrate unbounded model checking (UMC) techniques instead of unrolling them. Potentially, this approach would allow the reduction of the logic that has to be analyzed by UMC, since this technique is usually more expensive than bounded symbolic execution.

Parallel Computation: One last important improvement would be the use of multiple CDSE engines at the same time. Inspired by parallel SAT solvers, these CDSE engines could be configured to search in different search spaces, and share every conflict learned. Furthermore, the engines could run different decision heuristics, creating a portfolio of CDSE solvers.

Additionally, there are more practical directions that could be followed to improve Kite's applicability, and efficiency. Some of these improvements would be:

State Discarding: Currently, Kite always discards every state in the decision stack that is higher than the backtrack level. Consequently, Kite has to re-execute a redundant sequence of instructions to restore a discarded state, if needed. However, this limitation could be overcome by keeping track of all unblocked processes, and a mechanism to recover them. The main challenge of this approach would be how to determine which processes are alive, and which ones have been completely blocked.

Memory Handling: The CDSE learning scheme employs the knowledge of use-def chains to efficiently determine which merging points in the CFG could carry different variable definitions. In Kite, this knowledge is not available for memory accesses; thus, Kite assumes any merging point between a memory write and a memory read could impact on the memory value.

As discussed in Section 4.6, this approach could be improved by adding a use-def analysis of memory access. Even though this use-def analysis is sometimes imprecise, it would allow Kite to optimize simple, but common, memory accesses with constant indexes. For more complicated cases, Kite could also take advantage of pointer analysis, and determine a superset of relevant merging points to some memory region.

Kite would benefit from these changes while solving instances where memory accesses are more complex. These memory accesses cannot be eliminated by the optimization passes included in LLVM.

Symbolic Pointers: This is another limitation in Kite's implementation presented in Section 4.6. This limitation restricted the choice of tests used Section 5.

As discussed in Section 4.6, Kite could either fork at each pointer dereference, one process per possible address, or Kite could encode all possibilities as a conditional access. The first solution, which is implemented in Klee, would require a change in the fCFG. The fCFG would need to include the points in the program where these forks could occur. The second approach would require Kite to enumerate all possible values of a pointer, and encode conditional reads, as well as conditional writes.

Because larger examples tend to require more complete memory handling and support for symbolic pointers, with these improvements, Kite could extend the range of problems it can solve. Moreover, Kite could be used to investigate how scalable CDSE is. Since Kite excels over other tools for larger examples, it is reasonable to expect that CDSE may scale to problems that traditional symbolic execution can't.

Bibliography

[1] Mars Climate Orbiter Mishap Investigation Board. Phase I Report, NASA, November 1999.

[2] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[3] Domagoj Babic and Alan J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 211–220, New York, NY, USA, 2008. ACM.

[4] Dirk Beyer. Competition on Software Verification. `http://sv-comp.sosy-lab.org/2013/`, 2013.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, October 2007.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[7] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Ma-

chines and Languages with Only Two Formation Rules. *Commun. ACM*, 9(5):366–371, May 1966.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[9] J. Burnim and Koushik Sen. Heuristics for Scalable Dynamic Test Generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446, 2008.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[12] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[13] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[14] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[15] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[16] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract Conflict Driven Learning . In *Principles of Programming Languages (POPL)*, pages 143–154. ACM, 2013.

[17] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric Bounds Analysis with Conflict-Driven Learning . In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *LNCS*, pages 48–63. Springer, 2012.

[18] Niklas Eén, Alan Mishchenko, and Nina Amla. A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-based Abstraction. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 181–188, Austin, TX, 2010. FMCAD Inc.

[19] Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[20] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[21] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531. Springer-Verlag, 2007.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[23] Eugene Goldberg and Yakov Novikov. BerkMin: A Fast and Robust SAT-solver. *Discrete Appl. Math.*, 155(12):1549–1561, June 2007.

[24] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the Fifteenth*

National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[25] William R. Harris, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 71–82, New York, NY, USA, 2010. ACM.

[26] Joxan Jaffar, Jorge Navas, and Andrew Santosa. Abstraction Learning. In *Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis*, ATVA'10, pages 17–17. Springer-Verlag, 2010.

[27] James C. King. A New Approach to Program Testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

[28] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[29] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[30] S. Krishnamoorthy, M.S. Hsiao, and L. Lingappan. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *19th IEEE Asian Test Symposium (ATS), 2010*, pages 59–64, 2010.

[31] Eric Larson and Todd Austin. High coverage detection of input-related security facults. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[32] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Nancy G. Leveson. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18–41, 1993.

[34] Kenneth L. McMillan. Lazy Annotation for Program Testing and Verification. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 104–118. Springer-Verlag, 2010.

[35] K.L. McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2003.

[36] M.W. Moskewicz, C.F. Madigan, Ying Zhao, Lintao Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[37] NASA-JPL. Mars Climate Orbiter Fact Sheet. `http://web.archive.org/web/20130921065519/mars.jpl.nasa.gov/msp98/orbiter/fact.html`, 1999. Archived from the original on September, 2013.

[38] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. Amuse: A minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 518–523, New York, NY, USA, 2004. ACM.

[39] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In Natasha Sharygina and Helmut Veith, editors, *Com-*

puter Aided Verification, volume 8044 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, 2013.

[40] Corina S. Păsăreanu and Willem Visser. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.

[41] The LLVM Project. LLVM Download Page. `http://web.archive.org/web/20131107071125/http://llvm.org/releases/download.html`, 2013. Archieved from the original on November, 2013.

[42] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[43] Roberto Sebastiani. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3(3-4):141–224, 2007.

[44] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[45] João P. Marques Silva and Karem A. Sakallah. GRASP — A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[46] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 244–257. Springer-Verlag, 2009.

[47] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3, National Institute of Standards and Technology, 2002.

[48] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[49] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[50] Lintao Zhang. *Searching for Truth: Techniques for Satisfiability of Boolean Formulas.* PhD thesis, Princeton, NJ, USA, 2003.

[51] Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10880–10885, Washington, DC, USA, 2003. IEEE Computer Society.

Appendix A

Extended Results

This appendix presents the result data that was not included in Chapter 5. First, Tables A.1 and A.2 show the map between each instance used and their simplified name.

Benchmark	Instance	Simplified Name
ntdrivers-simplified	cdaudio_simpl1_safe.cil.c	safe_01
	diskperf_simpl1_safe.cil.c	safe_02
	floppy_simpl3_safe.cil.c	safe_03
	floppy_simpl4_safe.cil.c	safe_04
	kbfiltr_simpl1_safe.cil.c	safe_05
	kbfiltr_simpl2_safe.cil.c	safe_06
ssh-simplified	s3_clnt_1_safe.cil.c	safe_07
	s3_clnt_2_safe.cil.c	safe_08
	s3_clnt_3_safe.cil.c	safe_09
	s3_clnt_4_safe.cil.c	safe_10
	s3_srvr_1_safe.cil.c	safe_11
	s3_srvr_1a_safe.cil.c	safe_12
	s3_srvr_1b_safe.cil.c	safe_13
	s3_srvr_2_safe.cil.c	safe_14
	s3_srvr_3_safe.cil.c	safe_15
	s3_srvr_4_safe.cil.c	safe_16
	s3_srvr_6_safe.cil.c	safe_17
	s3_srvr_7_safe.cil.c	safe_18
	s3_srvr_8_safe.cil.c	safe_19

Table A.1: Renaming of Safe Instances.

Tables A.3 and A.4 shows the performance of both versions of Klee: Klee_f corresponds to the changeset from which Kite was forked[13], and Klee_l

[13] Changeset 9b5e99905e6732d64522d0efc212f3f1ce290ccc from September 12th, 2012

Benchmark	Instance	Simplified Name
ntdrivers-simplified	floppy_simpl3_unsafe.cil.c	unsafe_01
	floppy_simpl4_unsafe.cil.c	unsafe_02
	kbfiltr_simpl2_unsafe.cil.c	unsafe_03
	cdaudio_simpl1_unsafe.cil.c	unsafe_04
ssh-simplified	s3_srvr_10_unsafe.cil.c	unsafe_05
	s3_srvr_11_unsafe.cil.c	unsafe_06
	s3_srvr_12_unsafe.cil.c	unsafe_07
	s3_srvr_13_unsafe.cil.c	unsafe_08
	s3_srvr_14_unsafe.cil.c	unsafe_09
	s3_srvr_1_unsafe.cil.c	unsafe_10
	s3_srvr_2_unsafe.cil.c	unsafe_11
	s3_clnt_1_unsafe.cil.c	unsafe_12
	s3_srvr_6_unsafe.cil.c	unsafe_13
	s3_clnt_3_unsafe.cil.c	unsafe_14
	s3_clnt_4_unsafe.cil.c	unsafe_15
	s3_clnt_2_unsafe.cil.c	unsafe_16

Table A.2: Renaming of Unsafe Instances.

corresponds to the latest changeset[14] by the time the tests were executed.

Klee$_f$ had better performance than Klee$_l$ for all safe instances. For the unsafe instances, the Klee versions had similar performance. Thus, Klee$_f$ had the best overall performance, which explains why this was the version used in Chapter 5.

Finally, Tables A.5 and A.6 show the number of instructions symbolically executed by both Klee versions and all Kite versions to solve the safe and unsafe instances, respectively. As expected, the number of instructions executed by both versions of Klee to solve the safe instances are the same, since they both execute all feasible paths in the program (Table A.5).

[14]Changeset e49c1e1958e863195b01d99c92194289b4034bbb from January 21st, 2014

Test	$\textbf{Klee}_f$	$\textbf{Klee}_l$
safe_01	0.16	0.25
safe_02	0.45	0.47
safe_03	0.06	0.09
safe_04	0.14	0.19
safe_05	0.02	0.03
safe_06	0.04	0.05
safe_07	0.93	0.98
safe_08	0.92	0.98
safe_09	0.95	1.07
safe_10	0.92	0.98
safe_11	68.58	102.84
safe_12	0.09	0.16
safe_13	0.09	0.12
safe_14	40.72	66.16
safe_15	40.36	66.52
safe_16	40.39	64.99
safe_17	40.62	66.30
safe_18	40.30	66.06
safe_19	39.86	65.73
Total	**315.60**	**503.97**

Table A.3: The execution time (in seconds) spent by the two versions of Klee to solve the safe instances. Klee_f corresponds to the forked version, while Klee_l corresponds to the latest one.

Test	**Klee$_f$**	**Klee$_l$**
unsafe_01	0.24	0.14
unsafe_02	0.14	0.16
unsafe_03	0.04	0.04
unsafe_04	0.07	0.11
unsafe_05	0.04	0.03
unsafe_06	0.62	0.53
unsafe_07	1.60	2.29
unsafe_08	0.27	0.18
unsafe_09	0.15	0.07
unsafe_10	0.09	0.10
unsafe_11	0.10	0.10
unsafe_12	0.13	0.16
unsafe_13	0.04	0.03
unsafe_14	0.18	0.16
unsafe_15	0.14	0.16
unsafe_16	0.38	0.15
Total	**4.23**	**4.45**

Table A.4: The execution time (in seconds) spent by the two versions of Klee to solve the unsafe instances. Klee$_f$ corresponds to the forked version, while Klee$_l$ corresponds to the latest one.

Test	**Klee**$_f$	**Klee**$_l$	**Kite**$_B$	**Kite**$_G$	**Kite**$_{M(5)}$	**Kite**$_{M(10)}$	**Kite**$_{M(15)}$
safe_01	14,095	14,095	23,099	6,465	11,357	6,465	6,465
safe_02	12,876	12,876	6,046	3,602	4,122	**3,143**	3,421
safe_03	4,003	4,003	**723**	736	736	736	736
safe_04	6,945	6,945	13,913	3,257	3,326	3,292	**3,257**
safe_05	655	655	**0**	**0**	**0**	**0**	**0**
safe_06	1,909	1,909	**0**	**0**	**0**	**0**	**0**
safe_07	17,298	17,298	16,557	**6,243**	7,481	**6,243**	**6,243**
safe_08	17,284	17,284	18,008	**5,642**	6,034	**5,642**	**5,642**
safe_09	17,322	17,322	14,819	5,790	6,077	**5,790**	**5,790**
safe_10	17,284	17,284	16,413	5,726	6,074	**5,726**	**5,726**
safe_11	549,056	549,056	277,252	50,935	**46,063**	50,935	50,935
safe_12	**2,684**	**2,684**	6,166	2,824	5,846	5,399	5,612
safe_13	1,683	1,683	3,977	**1,563**	2,490	2,424	2,314
safe_14	614,650	614,650	480,643	**82,164**	101,893	116,105	102,153
safe_15	615,130	615,130	411,546	**71,751**	95,342	76,877	86,574
safe_16	613,162	613,162	793,898	**219,609**	416,020	387,664	351,395
safe_17	625,116	625,116	722,191	**187,700**	377,336	338,916	283,695
safe_18	616,034	616,034	677,706	**207,993**	423,542	399,795	346,147
safe_19	615,124	615,124	770,891	**204,738**	412,356	359,172	306,548

Table A.5: Table showing the number of instructions that were symbolically executed by all versions of Klee and Kite to solve the safe instances. In instances `safe_05` and `safe_06`, Kite was capable of proving property's correctness just by applying the code optimization passes, and the slicing algorithm.

Test	**Klee**$_f$	**Klee**$_l$	**Kite**$_B$	**Kite**$_G$	**Kite**$_{M(5)}$	**Kite**$_{M(10)}$	**Kite**$_{M(15)}$
unsafe_01	2,183	2,285	**334**	**334**	**334**	**334**	**334**
unsafe_02	4,762	5,215	819	**543**	**543**	**543**	**543**
unsafe_03	1,736	1,597	**244**	**244**	**244**	**244**	**244**
unsafe_04	3,873	5,016	16,064	6,056	**1,085**	6,056	6,056
unsafe_05	467	468	602	**414**	**414**	**414**	**414**
unsafe_06	11,549	12,710	2,658	**2,071**	2,858	2,224	**2,071**
unsafe_07	46,911	45,692	Timeout	Timeout	79,136	**5,034**	228,371
unsafe_08	4,406	**3,022**	1,052	3,033	12,038	3,033	3,033
unsafe_09	539	556	500	**489**	496	**489**	**489**
unsafe_10	1,153	1,200	3,959	**610**	677	**610**	**610**
unsafe_11	1,156	1,207	6,427	**632**	**632**	**632**	**632**
unsafe_12	**2,162**	2,330	2,710	2,638	6,280	2,638	2,638
unsafe_13	378	378	**377**	**377**	**377**	**377**	**377**
unsafe_14	2,339	**2,177**	4,200	2,513	2,501	2,513	2,513
unsafe_15	2,397	2,377	2,964	**649**	729	**649**	**649**
unsafe_16	2,397	2,155	2,550	1,183	**759**	1,183	1,183

Table A.6: Table showing the number of instructions that were symbolically executed by all versions of Klee and Kite to solve the unsafe instances.