Partitioning Strategies to Enhance Symbolic Execution

Brendan Adrian Marcellino

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
A. Lynn Abbott
Haibo Zeng

August 8, 2015
Blacksburg, Virginia

Keywords: Symbolic Execution, Software Testing,
Static Analysis, Partitioning Strategies

Copyright 2015, Brendan A. Marcellino

Partitioning Strategies to Enhance Symbolic Execution

Brendan A. Marcellino

(ABSTRACT)

Software testing is a fundamental part of the software development process. However, testing is still costly and consumes about half of the development cost. The path explosion problem often necessitates one to consider an extremely large number of paths in order to reach a specific target. Symbolic execution can reduce this cost by using symbolic values and heuristic exploration strategies. Although various exploration strategies have been proposed in the past, the number of Satisfiability Modulo Theories (SMT) solver calls for reaching a target is still large, resulting in longer execution times for programs containing many paths. In this paper, we present two partitioning strategies in order to mitigate this problem, consequently reducing unnecessary SMT solver calls as well. In sequential partitioning, code sections are analyzed sequentially to take advantage of infeasible paths discovered in earlier sections. On the other hand, using dynamic partitioning on SSA-applied code, the code sections are analyzed in a non-consecutive order guided by data dependency metrics within the sections. Experimental results show that both strategies can achieve significant speedup in reducing the number of unnecessary solver calls in large programs. More than 1000× speedup can be achieved in large programs over conflict-driven learning.

∼ To my family ∼

Acknowledgments

This thesis would not have been possible without the guidance of my committee members, friends, and support from my family. I would like to express my sincere gratitude to my advisor, Dr. Michael Hsiao, for his encouragement, patience and support during my research. His excellent teaching in Electronic Design Automation inspired me to join his PROACTIVE research group. I was honored to work with him and was grateful for his assistance and immense knowledge. I would also like to thank Dr. Lynn Abbott and Dr. Haibo Zeng for serving on my thesis committee.

Special thanks to my friends in the PROACTIVE research group for helping me during my research and for all the fun we had together - Sharad Bagri, Kelson Gent, Sarmad Tanwir, Sarvesh Prabhu, Prateek Puri and Yuan Zeying.

I also thank to my friends, Hanna Simon, Alex Sumadijaya, Adi Kusuma, Arezima Tabranintya, Matthew Bailey, Henry Chong, Yiling Xu, and Qingrui Liu for always supporting my study and making my stay at Virginia Tech meaningful and memorable.

Finally, I would like to thank my parents Maraden Marcellino and Sylvia Juliati and my sister, Rachel Amanda Marcellino for their love, support, and encouragement throughout my life. They have always been there for me and supported me both emotionally and spiritually.

Brendan A. Marcellino
August 8, 2015

Contents

List of Figures viii

List of Tables ix

List of Algorithms x

1 Introduction 1

1.1 Motivation 2

1.2 Contribution 4

1.3 Organization 5

2 Background 7

2.1 Symbolic Execution 8

2.2 Concolic Testing 11

2.3 Control Flow Graph 12

2.4 Static Analysis 14

2.4.1 .NET Compiler Platform (“Roslyn”) 15

2.5 Satisfiability Modulo Theories (SMT) Solvers 20
2.6 Related Work 21
2.6.1 DFS-Based Conflict-Driven Backtracking Strategy 23
2.7 Summary 27

3 Sequential Partitioning **28**
3.1 Partitioning Technique 30
3.2 Exploration Strategy 32
3.3 Coverage Analysis 34
3.4 Advantages of Sequential Partitioning 34

4 Dynamic Partitioning **36**
4.1 UNSAT Core 39
4.2 Individual Sections 41
4.3 Section Pairs 42
4.4 Improvements in Dynamic Partitioning 44
4.4.1 Dynamic Partitioning Using the Topology of a Program 44
4.4.2 Sequential and Dynamic Transition 47
4.5 Coverage Analysis 48
4.6 Advantages of Dynamic Partitioning 51

5 Experimental Results **53**

6 Conclusion and Future Work **64**

Bibliography **66**

List of Figures

2.1 An example of translating statements into SSA form 9

2.2 An example of C# program to illustrate symbolic execution 9

2.3 An example of C# program to illustrate concolic testing 11

2.4 CFG representation based on Figure 2.2 13

2.5 Traditional compiler pipeline 16

2.6 .NET compiler pipeline 16

2.7 Roslyn main layers - Compiler and Workspace APIs 17

2.8 The Workspace layer hierarchy 19

2.9 An example of UNSAT core problems related to data dependencies 25

3.1 Flow diagram of sequential partitioning 29

3.2 Partitioning example in C# program 31

4.1 Flow diagram of dynamic partitioning 37

4.2 An example of how arbitrary sections can be merged after translating into SSA form 51

List of Tables

5.1 Results for 3-Section Partitions 55

5.2 Results for 6-Section Partitions 57

5.3 Comparing Dynamic Improvements with Current Strategies 59

5.4 Performance Comparison of All Partitioning Strategies Based on Table 5.3 60

5.5 Results for Real Programs 62

List of Algorithms

2.1 DFS-Based Conflict-Driven Backtracking Strategy 24
3.1 Sequential Partitioning 32
4.1 Dynamic Partitioning 38
4.2 Constraint Solver Execution 38

Chapter 1

Introduction

Software testing plays a critical role in the software development process. There are several benefits for performing such tests in every phase in the software development cycle, such as quality improvement, guidance for both verification and validation and reliability estimation [26]. Regardless of the contributions, testing comes at a price. According to [31], testing makes up roughly 50% of the total development cost. In other words, testing consumes more resources than any other phase in a software project. This situation becomes even worse with the increasing code sizes as both scalability and computational costs pose tremendous challenges for testing. In order to tackle these issues, symbolic execution in recent years have offered some relief as it can explore the program state space with symbolic values and heuristic search strategies.

Several symbolic execution techniques [28, 23, 30, 8, 35, 38, 24] have been proposed for improving coverage analysis. Depth-First Search (DFS) strategy [40], for instance, is a simple and popular

technique. It serves as the basis for several exploration strategies as it systematically traverses each path in a program. However, this strategy is generally not scalable to large programs due to the path explosion problem, in which the number of paths grows exponentially with an increasing program size [6]. Another popular technique is called Breadth-First Search (BFS) strategy [43]. It traverses each path in a program based on the hierarchy levels. This strategy has a “Domino Effect” with the previous sections affecting the subsequent sections. As a result, having fewer number of feasible paths in the earlier sections may be advantageous to the analysis results for certain programs that have infeasible path segments early in the code.

Another approach [38] is based on concolic testing which combines both concrete values and symbolic values to execute a program simultaneously. However, this technique also suffers from scalability issues. Other works have been proposed to improve concolic testing in different ways. [24] shows that using interpolation for assisting concolic execution can prune redundant paths. Such paths can be subsumed if the interpolant is implied as they can be guaranteed to not be buggy. One issue with this approach is that path subsumption is used to skip path execution which can only be guaranteed in programs annotated with assertions.

1.1 Motivation

Several strategies have been proposed to prune the search in a program. State space pruning has traditionally been performed by abstraction of the original program. In addition to abstraction, techniques to find invariants that can block out illegal value combinations can also be used, such as in

[10] and [42]. The method of lazy annotation was introduced in [30] to deduce program annotations in response to search failures, as similar to conflict-driven clause learning (CDCL) in Boolean satisfiability (SAT) solvers. A set of program locations are designated as goals to be reached. Search failures happen if the search fails to reach a goal. This method is based on interpolants to annotate the program with a learned fact that constrains future search while backtracking.

A DFS-based strategy with conflict backtracking was proposed in [28] using a conflict-driven learning strategy. This strategy makes use of the unsatisfiable core (UNSAT core) from the SMT solver to derive any useful information about the conflicting nodes. The idea is when an infeasible path is encountered, it will nonchronologically backtrack to the nearest conflicting nodes from its current position. Using this strategy, a number of infeasible paths can be skipped to achieve better performance in terms of a reduced number of calls as compared to the traditional DFS strategy. Even though such a strategy has better performance than the first technique, there are limitations on using this approach especially in large programs with many data dependencies. In addition, extraction of UNSAT cores may be expensive, and the one returned (from possibly several UNSAT cores) may not be optimal.

Tackling the problems in data dependencies in large programs becomes our motivation in our work since the traditional non-chronological backtracking strategy may not yield the best results. In this thesis, we propose two partitioning strategies that have their own unique abilities in analyzing programs and finding conflicting nodes resulting in a significant reduction of unnecessary SMT solver calls compared to the previous technique.

1.2 Contribution

We propose 2 different partitioning strategies in order to overcome the aforementioned challenges. The idea of the partitioning itself is to divide all paths in a program into several sections so that we can better identify the root cause of any conflicts. Therefore, we can potentially eliminate a significant number of unnecessary solver calls and better handle large programs.

The first technique in our proposed strategy is sequential partitioning. It analyzes the sections in a sequential order based on the feasible paths from the earlier sections. By focusing on the limited set of sections, the reason for any infeasible paths is confined to those sections in question. This is in contrast to conventional learning strategies which consider an entire undivided path and might not return the unsat core that is nearest to the top of the path because there might exist several reasons for an infeasible path. Thus, the sequential partitioning helps the pruning process by indirectly pushing the unsat core to near the top of the path. Any conflicting segments in earlier sections will help to reduce a significant number of solver calls for the subsequent sections. The second strategy, dynamic partitioning, introduces metrics to analyze data dependencies among the sections. Using these metrics, we can analyze the sections to determine those sections that should be grouped first in our analysis. The sections may not be adjacent or consecutive in their order. As a result, we can potentially eliminate a significant number of unnecessary solver calls and better handle large programs. Because the original code is completely represented in static single-assignment (SSA) form, our method will not call a feasible path infeasible, even after some intermediate sections are not included.

In this work, we use Roslyn [1], a .NET compiler platform for static analysis. As for the constraint solver, we use Z3 from Microsoft Research [16]. Experimental results show that both strategies can achieve significant speedups, effectively reducing the number of solver calls especially in large programs. More than 1000× speedup can be achieved in large programs over conflict-driven learning techniques.

To summarize, this thesis makes the following contributions as follows:

- We propose sequential partitioning technique for analyzing programs in a sequential order taking advantage of infeasible paths discovered in earlier sections which will be useful for future search in order to reduce unnecessary solver calls.
- We propose dynamic partitioning technique which utilizes data dependency metrics within the partitions and conflict driven learning technique to determine those that are best to analyze together and prune the search space, respectively.
- We compare our partitioning strategies to the conflict-driven backtracking strategy on several test cases resulting in a speedup of 3 orders of magnitude in large programs.

1.3 Organization

The remainder of this thesis is organized as follows.

Chapter 2 explains the concepts that are required to understand this work starting from symbolic execution to static analysis. This chapter also covers the path explosion problem in DFS technique

and the previous works related to this research.

Chapter 3 introduces the sequential partitioning technique. This chapter also describes the advantages and disadvantages of using this technique.

Chapter 4 explains the limitations of sequential partitioning strategy and introduces a more flexible partitioning technique called dynamic partitioning.

Chapter 5 shows the effectiveness of our partitioning strategies over the previous work by providing some experimental results.

Chapter 6 concludes our thesis and outlines the future work.

Chapter 2

Background

In this chapter, we review the basics of symbolic execution and satisfiability modulo theories (SMT) solver as they are essential in our work. We also describe the basic depth-first search (DFS) strategy and explain the drawbacks of using such strategy which results in a path explosion problem.

In order to analyze the programs thoroughly, we use static analysis that can break down the structure of a program into several categories. We introduce Rosyln [1] which is a .NET compiler platform tool for static analysis. The tool uses abstract syntax tree (AST) to represent the lexical and syntactic structure of the programs. We also describe further how this tool works and how it helps implementing our techniques and creating the partitions as well.

Finally, we explain some of the related works on pruning the search space using symbolic execution technique which varies from concolic testing to lazy annotation method. We also describe the conflict-driven technique in detail which becomes our motivation for proposing our strategies.

2.1 Symbolic Execution

In symbolic execution [27], a program is executed using symbolic values instead of concrete values. Upon the execution of a program, a trace consisting of symbolic expressions along the paths will be generated. This trace consists of assignment and conditional expressions. The conjunction of all symbolic expressions within a trace will form a path constraint. This path constraint can be solved using a number of constraint solvers, such as the Satisfiability Modulo Theories (SMT) solver. If the SMT solver returns SAT (satisfiable), then there would be a set of inputs that satisfy the current path. The assignment is useful for test data generation as they set the input values to exercise the target path in the program. On the other hand, if the solver returns UNSAT (unsatisfiable), there does not exist any valid inputs that can simultaneously satisfy all constraints along the given path. Even though the path is infeasible, we can still derive useful information to find the root cause of the problems by extracting the UNSAT core. There may exist multiple UNSAT cores for an infeasible path. Typically, a solver will return only one UNSAT core. This core will consist of a set of clauses that the solver has determined to be unsatisfiable. Several strategies took advantage of the UNSAT core to build and enhance their exploration strategies, including [28].

Since a path might involve the same variable multiple times, we need to use Single Static Assignment (SSA), which is commonly used in modern compilers, into our work. Using this technique, we can differentiate variables that are being used in different program points. This is also necessary because the SMT solver will always return UNSAT for an expression that has conflicting constraints involving the same variable. An example of how transforming into SSA form can be shown in

Figure 2.1. The left part is the original source code running sequentially, while the right part shows the results after translating into SSA form. Note that on the third line, we need to make a new variable for x since it is a new assignment to the same variable x.

x = x++;		$x_1 = x_0 + 1$;
y = x;	➔	$y_0 = x_1$;
x = y + x;		$x_2 = y_0 + x_1$;

Figure 2.1: An example of translating statements into SSA form

We show how symbolic execution can be applied in a C# program, illustrated in Figure 2.2. Note that all assignments have been translated into the SSA form before adding them into a path constraint. This should be done to avoid any conflicting constraints within the path constraint. Suppose we have two assignments, $x = y$ and $x = x + 1$, respectively. If we let SMT solver checks these clauses, the output would be unsatisfiable (UNSAT) since variable x has conflicting constraints by having 2 values at the same time. Thus, SSA plays a great role in dealing with this situation by categorizing each variable that has been used individually. Upon translating all assignments, the results would be $x_0 = y_0$ and $x_1 = x_0 + 1$, which are acceptable for the SMT solver.

```
0:     public void A (int x, int y) {
1:        int z = x + y;
2:        x = x * 2;
3:        if (x > z) {
4:           if (y >= 3)
5:              x++;
6:        }
7:        else {
8:           x--;
9:        }
10:    }
```

Figure 2.2: An example of C# program to illustrate symbolic execution

It can be observed that when the program was executed with concrete values $x = 4$ and $y = 3$, a particular path taken by the program would be along line numbers $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 10$. During symbolic execution, the values of symbolic variables change along with the instructions. For example, when the instruction $x = x * 2$ is executed, the symbolic value of x is changed from x to $x + 2$. Similarly, symbolic expressions are formed whenever a branch is encountered and the conditional expression is evaluated based on its *true* or *false* transition. During the concrete run, the condition for $(x > 2)$ is evaluated to true. Hence, the symbolic expression for this conditional is $(x > 2 = true)$ and we can simplify it as $(x > 2)$. On the other hand, if the conditional is evaluated to false, the symbolic expression for this conditional would be $(x > 2 = false)$ or $(not(x > 2))$. This expression is called ***branch constraint***. The path constraint is computed as the conjunction of all clauses from the expressions (including branch constraint) encountered along the path: $(z_0 = x_0 + y_0) \wedge (x_1 = x_0 * 2) \wedge (x_1 > z_0) \wedge (y_0 \geq 3) \wedge (x_2 = x_1 + 1)$.

The solver will analyze this path to determine if the path is satisfiable or not. In this example, the solver will return SAT as there is a set of inputs $(x_0 = 4, y_0 = 3)$ that can satisfy this path constraint. To analyze a different path, the current path constraint can be modified by negating any of the branch predicates and bringing in the associated statements in the associated block. Suppose we are interested in finding out whether line 8 is reachable or not given the existing path constraint from the previous example, we can negate the conditional expression in line 3, giving a new path predicate as: $(z_0 = x_0 + y_0) \wedge (x_1 = x_0 * 2) \wedge (x_1 \leq z_0)$. In this way, all paths can be traversed systematically.

2.2 Concolic Testing

Concolic testing is a hybrid technique that uses concrete execution along with symbolic execution. It uses concrete inputs to execute the program. Upon the execution of a program, a trace consisting of all components that have been exercised earlier will be generated. While traversing the executed path, all constraints of symbolic inputs are recorded. Any unreachable branches will be analyzed using symbolic execution together with solver constraints to generate new inputs in order to maximize the code coverage. As a result, concolic testing can be considered as an improvement over symbolic execution.

In order to understand this technique better, consider an example in Figure 2.3. In this case, we want to find out if the error in line 4 can be reached or not. Random testing might not work efficiently because it would require a large number of tests to reach the error. However, random testing can find simple and uncomplicated bugs quickly. Still, as the bugs are getting hard to find, random testing would have problems in finding them as it requires a lot of combinations.

```
0:    public void B (int x, int y) {
1:        int z = x + y;
2:        if (x == 10) {
3:            if (x > z) {
4:                /* error */
5:            }
6:        }
7:    }
```

Figure 2.3: An example of C# program to illustrate concolic testing

Assuming we have an arbitrary choice for x and y with both variables are assigned to 1. During concrete execution, we set z to 2 from the addition of x and y, and break in line 2 since $1 \neq 10$.

Symbolic execution will also record all constraints starting from line 1, which sets z to $x + y$, to line 2, which compares x to 10. Since we know the conditional in line 2 is not satisfied, a constraint solver will be invoked given all constraints from symbolic execution to get any satisfying inputs. In this case, the result from the solver would be x = 10 and y = 1.

Running this program using the new inputs, it can reach the second conditionals in line 3. However, it cannot reach line 4 since 10 (x) is not greater than 11 (z). The procedure is then repeated by invoking the constraint solver given all expressions from line 1 to 3, in which $(z = x+y)$, $(x == 10)$ and $(x \geq z)$. The constraint solver will return input values that can satisfy the formula (x = 10 and y = 0). This inputs can reach the error.

Apart from their strengths, concolic testing still suffers from scalability issues. According to [36], most concolic testing tools have problems with multi-threaded code. Hence, they assume a program is single-threaded and deterministic. In addition, concolic testing dependent on the effectiveness of its constraint solver. Therefore, when the constraint is beyond the ability of the solver, symbolic execution may not analyze the path and switch to explore new paths. In addition, this technique uses basic depth-first search (DFS) strategy to scan the search space. As a result, it is not suitable for large programs due to the path explosion problem.

2.3 Control Flow Graph

The Control Flow Graph (CFG) is a graph representation of the flow of possible executions in a program [2]. It is constructed as a directed graph $G(V, E)$ with vertices or nodes representing the

execution statement and branch predicates, and edges representing the execution path between nodes. The CFG is also critical to many compiler optimizations and static analysis tools.

Each basic block should consist of maximum program statements under several conditions. First, each block only has a single entry and exit point. All statements that can be reached from another basic block and are leading to the execution of another basic block can be grouped together. Second, ordering within the block is necessary to maintain the program runs in sequential orders.

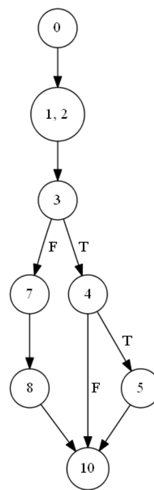


Figure 2.4: CFG representation based on Figure 2.2

Figure 2.4 is the transformation from a program shown in Figure 2.2 to a control flow graph (CFG). The block id is the same as the line number in the program. The second block from the top consists of line 1 and 2 since both statements can be accessed right after the program runs and both are entering the conditional expression in line 3. Furthermore, all conditionals are placed inside individual blocks because their exit points are unique among each other.

CFG is really useful in our work since it can systematically explore the search space in a program. Using the graph, we can record all covered paths and traverse the remaining ones. Many existing

strategies use CFG to make decisions on whether a particular branch constraint need to be reversed in order to explore a new path.

2.4 Static Analysis

There are several testing techniques to detect software bugs including static and dynamic analysis. While static analysis examines the program source code during compile-time, dynamic analysis performs the analysis during run-time. Thus, the two approaches are complementary because no single approach can find error better than the others. Since our work use static analysis mostly to determine further actions based on the structures of a program, we will discuss this analysis in detail.

The cost to fix defects or bugs increases as their complexity. Any bugs will be harder to find as the programs get bigger. In other words, the longer the defect persists, the more expensive it gets. Therefore, detecting bugs at an early phase of the software development cycle will reduce the cost significantly. Static analysis is useful to understand the structure of the code. It examines all possible execution paths and variable values, not just those that are invoked during execution. Thus, static analysis can detect bugs or errors at an early stage.

Static analysis tools and compiler are oriented to improve the quality of the source code and minimizing any potential bugs that are hard to find and debug. Both rely on static analysis of the source code during compilation to generate warnings and diagnostics. Many compilers have different capabilities in diagnosing specific pattern bugs, so using different compilers will improve

the quality of the code. However, different compilers may have different languages and structures that other compilers do not support. Thus, in most cases, a program cannot be compiled with many compilers. On the other hand, most static analysis tools have smarter algorithm so that it can be more flexible with the source code. In this work, we use Rosyln, a .NET compiler platform, from Microsoft as our static analysis tool [1]. It provides open-source C# compilers with rich APIs.

2.4.1 .NET Compiler Platform (“Roslyn”)

Compilers are usually presented as black boxes, in which all source codes are presented in the beginning, and output files in any forms, such as executable files or assemblies, are generated in the end to the user. It is quite hard to know what happens in the middle of the process. While compilers perform their tasks, they build up deep understanding to the code but this information in unavailable to anyone but the compiler implementation wizards.

In order to improve our code quality, we rely on code analysis tools that have integrated development environment (IDE) features such as IntelliSense for debugging, “Find all references”, refactoring and intelligent renaming. As these tools get smarter, they need access to the deep code knowledge that the compiler possess. This is the primary goal of the .NET Compiler Platform (“Roslyn”). It opens up the black boxes and allow users to share and use these kind of information in their code.

Roslyn provides the APIs that mirror a traditional compiler pipeline. There are 4 phases in total. First, the source code is parsed into syntax that follows the language rules. Second, declarations from source code and imported metadata are analyzed to form symbols. Third, the binding process

matches identifiers with symbols. Finally, all information is emitted as an assembly. The complete pipeline is illustrated in Figure 2.5.

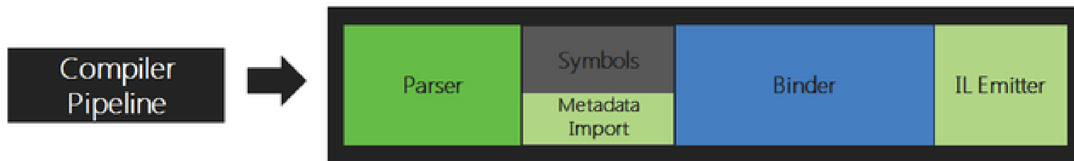


Figure 2.5: Traditional compiler pipeline

Roslyn breaks all phases into single objects that can be accessed individually. The parsing phase is transformed as a syntax tree, the symbols and imported metadata phase are represented in symbol table, the binding phase exposes the result of the compiler's semantic analysis and the emit phase for producing Intermediate Language (IL) bytes code. The pipeline can be shown in Figure 2.6.

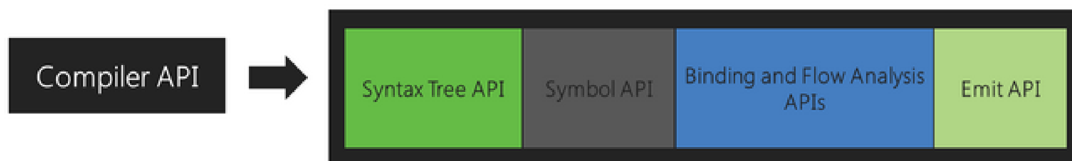


Figure 2.6: .NET compiler pipeline

There are 2 main layers of APIs in Roslyn as shown in FIgure 2.7 - the Compiler APIs and Workspace APIs. The compiler layer contains objects that directly connect all information from the compiler pipeline, both syntactic and semantic. The compiler APIs can also produce a set of diagnostics to be plugged into a Compilation and user-defined diagnostics. Thus, user can have the benefits by interacting naturally through Visual Studio such as halting a build and showing squiggles in the editor and generating code fixes. The team is currently reviewing the hosting/scripting APIs for executing code snippets as part of the compiler layer. The workspace layer, on the other hand,

contains all information needed for refactoring and doing code analysis over the entire solution. It passes all information about the project into a single object model without having to parse the file, configure options and manage the project dependencies.

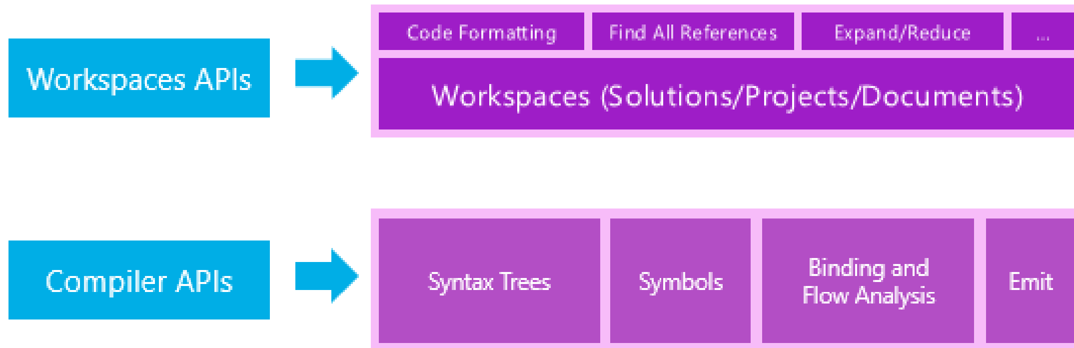


Figure 2.7: Roslyn main layers - Compiler and Workspace APIs

The most important structure exposed in the compiler layer is the syntax tree as it represents the lexical and syntactic structure of the code. Thus, it can process and analyze the syntactic structure of source code in a project. In addition, syntax trees are useful to rearrange the source code without having to do it manually. By controlling the trees, we can create and rearrange the source code in a more flexible way.

Syntax trees have 3 main attributes. The first attribute is that syntax trees keep all information in full support. This means that every information found in the source code including whitespace and comments are stored. Even if the programs are not compilable, syntax trees will show the missing tokens. The second attribute is that a text representation from the source code can be obtained by analyzing the syntax trees. This means that from any syntax node, it is possible to retrieve the text representation by looking at its sub-nodes. As a result, editing the source code is possible by

creating a new tree out of an existing syntax tree. The third attribute of syntax trees is that they are immutable. This means that the original syntax tree cannot be modified and changed. Thus, multiple users can use the same syntax tree at the same time without any race conditions. The trees are also efficient in terms of memory usage because the new version will reuse the underlying nodes. This will make creating a new tree becomes faster as well.

There are several elements that build syntax trees such as syntax nodes, tokens and trivia. Syntax nodes are one of the most important structures that control the trees. They represent the declarations, expressions and statements from the source code. Each of them are categorized into separate classes under SyntaxNode library. All syntax nodes are non-terminal nodes which means that they always have child nodes consisted of tokens or other nodes. In addition, the child nodes are represented in sequential orders according to their position in the source code.

Syntax tokens are the smallest structure of the code representing keywords, identifiers and literals. Thus, they never became parents of other tokens or nodes. Unlike syntax nodes, there is only a single structure for all kinds of tokens. A numeric value, for instance, will be represented as integer literal tokens with the Value property that tells you the decoded numbers. On the other hand, a string will be decoded from the ValueText property.

Syntax trivia are consisted of insignificant structures of the code such as comments and whitespace. Since they are not important in understanding the code, they are not included as a node within a syntax tree. However, as they are important in refactoring and to keep the source code complete, they are considered as part of the syntax tree. The only way to access them is by looking at the properties within a token as LeadingTrivia or TrailingTrivia. Unlike syntax nodes and tokens, syntax

trivia do not have parents. They do have the same parents as the token they are associated with.

During static analysis, it is useful to know the location of several structures within the source code. Roslyn provides the TextSpan class to know the position of each node, token and trivia within the source text. It consists of the beginning position and a count of characters which are represented as integers. Each node has 2 TextSpan properties: Span and FullSpan. The only difference is that the FullSpan property consider any leading and trailing trivia, while the Span property does not.

The Workspace layer organizes all information about the project in a solution into a single object model that connects directly with the syntax tree, compilation and semantic models. A workspace is consisted of all documents within a project. Any changes within a source document will trigger an event to show that the overall model of the solution has been changed. Figure 2.8 is a representation of how the Workspace relates to all elements in a model. Each solution consists of projects and documents and its elements change (represented as the superscripts) as we modify the solution by constructing new instances based on existing solutions.

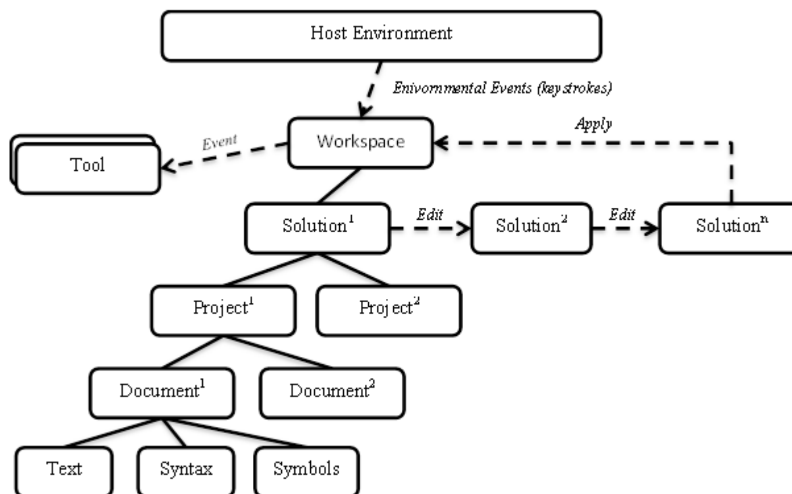


Figure 2.8: The Workspace layer hierarchy

2.5 Satisfiability Modulo Theories (SMT) Solvers

Satisfiability is one of the most fundamental problems in theoretical computer science, namely the problem of determining whether a formula expressing a constraint has a solution. The most well-known constraint satisfaction problem is propositional satisfiability (SAT), where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables [17]. Other problems are formulated more naturally and compactly in classical logics, such as first-order or higher-order logics, with a more expressive language that includes non-Boolean variables, function and predicate symbols (with positive arity) and quantifiers. The problem of evaluating the satisfiability of first order formulas with respect to some background theories is called Satisfiability Modulo Theory (SMT) [4, 11].

The current SMT technology that exists is built upon 40 years of theory and empirical studies. Most of today's solvers are based on the Davis-Putnam procedure that was created in 1960 [15] and then revised for performance in 1962 [14]. For historical reasons, this is now known as the DPLL (Davis-Putnam-Logemann-Loveland) procedure. This family of satisfiability solvers leverages the separation of clauses in the conjunctive normal form (CNF) of propositional formulas [25]. A CNF formula is a conjunction of clauses where a clause is a disjunction of literals, and a literal is in turn, an atomic formula (atom) or its negation. It is similar to the product of sums form used in circuit theory. The formula is satisfiable if there exists an assignment of truth values to the used Boolean variables in order for the entire formula to be true. If there does not exist such an assignment, the formula is said to be unsatisfiable.

The most successful approach relies on a modern SAT procedure to deal with the boolean search, and on dedicated theory solvers (which are able to decide the satisfiability of theory atoms with respect to the background theory [32, 39, 20, 7]. SMT solvers decide satisfiability of functions modulo theories. A theory is a set of axioms and rules of inference in which first order logic predicates are interpreted. Formally, a theory T is the set of axioms and all deducible formulas from the rules of inference (all true statements). A formula F is T-satisfiable if F $\wedge$ T is is satisfiable in the first order sense. If not, F is T-inconsistent, or T-unsatisfiable [25].

There are 2 major approaches for implementing SMT solvers, referred to as the eager and lazy approach. The eager approach translates the inputs into a propositional conjunctive normal form (CNF) formula which will be checked by a SAT solver for satisfiability. The lazy approach, on the other hand, treats all atoms as propositional formula until a satisfying assignment is found. This formula is solved by a SAT solver whose results will be checked with dedicated theory solvers. If the model is consistent with the theory, then the formula is satisfiable. Otherwise, any violated clauses are added to the original formula and the search continues.

There are several popular SMT solvers including Yices [19], Z3 [16] and CVC4 [3]. In this research, we used Z3 from Microsoft Research.

2.6 Related Work

In order to enhance symbolic execution, several heuristic exploration strategies have been proposed in the past. One such technique is to randomly choose the values over several possible inputs [5, 34].

Random testing is known to be very fast in finding major software bugs [21]. However, there are 2 problems with this strategy. First, there could be higher chances to have redundant paths among different inputs which will result in having the same coverage areas. As a result, random testing has a poor code coverage. Second, the probability of selecting several inputs that can catch all buggy behaviors is very small [33]. In order to to overcome the aforementioned challenges, a tool called DART [22] was created. It uses automatic random testing by combining it with automatic interface extraction to avoid redundant paths and implements dynamic test generation to drive the program along alternative conditional branches. However, DART could handle only integer constraints, not pointer constraints. As a result, some paths are still skipped. CUTE [38] was created with a more advanced technique by using concolic testing to improve test coverage and be able to solve pointer constraints in sequential programs.

DART and CUTE use the basic depth-first search (DFS) strategy for path exploration. After each execution, the last branch constraint is analyzed to check whether its counter branch has been explored or not in order to explore a new path. Thus, this strategy can explore all paths in a program to reach 100% coverage in theory. However, this strategy is not suitable for large programs due to the path explosion problem, in which the number of paths increases exponentially with an increasing of program size.

In order to improve the concolic testing, Jaffar [24] uses interpolation for assisting concolic execution process to prune redundant paths. This strategy mitigates the path explosion problem of concolic testing by subsuming paths that can be guaranteed to not be buggy. However, such guarantee is feasible only if the program is annotated with assertions. Without assertions, each path is unique and

cannot be considered equivalent because there is not enough basis to relate one path from another.

Best-first search is a search technique proposed in [9], which is basically a DFS enhanced with breadth-first aspects. At intervals, all active choice points are ranked according to some internal heuristic and the best branch is expanded.

To tackle the scalability challenges, several strategies use conflict-driven learning to prune the search space in a program. Lazy annotation method [30] works by deducing program annotations in response to search failures while backtracking. Another approach has been proposed in [28] which uses reachability graph of a program to make a decision whether the current branch must be explored or not and introduces a conflict-driven backtracking strategy. Since our work is trying to improve the conflict backtracking strategy, it is necessary to discuss this strategy in detail.

2.6.1 DFS-Based Conflict-Driven Backtracking Strategy

A DFS-based conflict-driven backtracking strategy [28] performs the coverage using the reachability graph of a program. This strategy uses DFS to traverse all paths within a program and invokes the SMT solver to check the feasibility of a path. In their approach, an infeasible path will be recorded and an UNSAT core will be extracted. Since all paths that involve the same conflict will be infeasible, we can backtrack immediately to the conflicting node that is at the highest decision level. This can significantly improve the traditional DFS strategy, in which any backtrack is performed one at a time chronologically which contributes to many unnecessary solver calls. The difference in the number of unnecessary calls can be quite large with increasing code sizes. As a result,

non-chronological backtracking strategy will perform better than the traditional DFS approach. The complete algorithm of this strategy is shown in Algorithm 2.1 [28].

Algorithm 2.1 DFS-Based Conflict-Driven Backtracking Strategy

```
1:  Path ← ∅, Tests ← ∅
2:  CurrNode ← initial node
3:  append CurrNode to Path
4:  while Path is not empty do
5:      if not covered any branches of CurrNode then
6:          if (branch assignment == false) violates any learned conflict clauses then
7:              transition ← true branch of CurrNode
8:          else
9:              transition ← false branch of CurrNode
10:         end if
11:         if transition can reach any unvisited important branches then
12:             if transition does not lead to terminal node then
13:                 append transition to Path
14:             else
15:                 if path constraint of Path has solution then
16:                     record Tests
17:                 else
18:                     extract unsat core and map core to branches
19:                     blevel ← compute backtrack level
20:                     update conflict clause database
21:                     record Path as infeasible
22:                 end if
23:             end if
24:         end if
25:     else if covered only one branch of CurrNode then
26:         /*symmetric case for counter-branch*/
27:     else
28:         if Path was infeasible then
29:             remove all nodes following blevel from Path
30:             CurrNode ← node in blevel /*backjump*/
31:         else
32:             remove CurrNode from Path
33:             CurrNode ← previous node in Path /*backtrack*/
34:         end if
35:     end if
36: end while
```

Performing non-chronological backtracking can be implemented using a SMT solver to generate an unsatisfiable (UNSAT) core. An unsat core is a set of clauses which makes the path constraint unsatisfiable. However, there are limitations on such an approach especially in programs with many data dependencies. Note that the solver may not return the minimum nor the best unsatisfiable core. It will also not generate all possible sets of conflicting clauses from the original formula. Hence, the UNSAT core returned is one among several from the infeasible path.

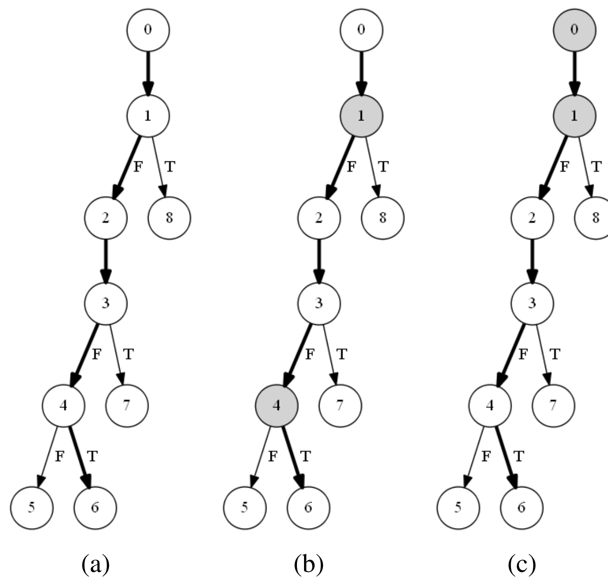


Figure 2.9: An example of UNSAT core problems related to data dependencies

An example CFG is shown in Figure 2.9 to illustrate the aforementioned problem. Note that nodes with a single successor are assignments, while those nodes having two successors are conditionals. Suppose we are interested in analyzing a particular path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$ as highlighted in bold in Figure 2.9(a). Suppose this path is infeasible, the solver will return UNSAT and produce a set of conflicting clauses (unsat core) which is denoted as node 1 and 4 (grayed out) shown in Figure 2.9(b). Using the non-chronological backtracking strategy, it will jump backward multiple

levels to the node at the highest decision level according to the unsat core, which is node 4. The next step is to store all conflicting nodes into a database so that it will skip any paths that contain any of the unsat constraints. It will then change the direction by negating node 4 in order to reach node 5 and continue the searching process from this point onward.

However, there may exist other reasons that can explain the original infeasible path. One of which is the combination of node 0 and 1 as shown in Figure 2.9(c). If this UNSAT core was returned by the solver instead, it will allow the search to backtrack more levels to node 1. After negating this node, it will start analyzing in another direction towards node 8. The difference in the number of solver calls between Figure 2.9(b) and 2.9(c) can be significant. This situation will become even pronounced if we add more data dependencies in the program because there might be nested problems within the analyzed path.

Tackling the problems in data dependencies in large programs becomes our motivation since the SMT solver may not return the best UNSAT core. Our strategy makes use of graph partitioning. The idea is to find any conflicting nodes restricted to a given partition and store them in the database so that we can skip any paths that contain them. Using our approach, we can determine that nodes 0 and 1 are conflicting for a path segment in the first section of the code and change the direction into node 8 with only a single analysis. We propose two partitioning strategies that have their own unique abilities to analyze a program and find conflicting nodes resulting in a significant reduction of unnecessary SMT solver calls compared to the previous techniques.

2.7 Summary

This chapter presented a brief overview of symbolic and concolic execution technique and satisfiability modulo theories (SMT) solver as to introduce the content of the thesis more formally. The usage of control flow graph (CFG) in a program was described. The idea of using single static assignment (SSA) in path constraints was explained. Static analysis technique and its tool were described in further detail. Several related works were discussed and a motivating example was presented.

Chapter 3

Sequential Partitioning

Our proposed partitioning strategies use static analysis to divide a program into several sections based on their structures. In our work, the sections are categorized based on the primary control statements of the programs. In other words, there will not be any nested conditions happening among the main partitions themselves. All partitions will be treated and analyzed in a completely different manner, giving each strategies its own advantages among others.

Sequential partitioning is named for the fact that the sections are analyzed in a sequential order. The analysis is based on the feasible paths discovered from earlier sections. Depth-First Search (DFS) strategy is first performed to break down each section into individual paths which will be stored in a database. There are two reasons for creating individual paths in each section. First, the analysis for the first section of the code will be performed for each path. Second, since this strategy aims to find the root cause of the problems in an infeasible path, determining which paths can be skipped

will be much easier if we can break each path into several parts. For example, suppose we have 10 sections, and an infeasible path is found while analyzing sections 1 and 2. Then, all paths that are extended through sections 3 to 10 that contain the infeasible segment in sections 1 and 2 can be skipped. This, in return, will reduce a number of unnecessary solver calls which might result in the reduction of the execution time as well.

Figure 3.1: Flow diagram of sequential partitioning

In general, there are three main phases in sequential partitioning. First, a program is provided in beginning of the process. Static analysis can scan all code but compilable programs are preferred. Second, using a partitioning technique, several sections will be created by categorizing the program based on the primary control statements. There would be at least a single section in a program. Finally, all sections will be analyzed in a sequential order based on the feasible paths from the earlier sections. In case of having only a single section, the results would be the same as compared to the basic DFS strategy due to lack of optimization. The flow diagram can be seen in Figure 3.1.

The next few sections of this chapter will discuss several important aspects in sequential partitioning. The partitioning technique itself is important for our sequential strategy as our analysis will rely on the partition results. We will also explain our exploration strategy in further detail. Coverage analysis is also important to demonstrate the capabilities of our technique. Several advantages of using sequential strategy are presented as well.

3.1 Partitioning Technique

Partitioning a program is one of the most important role in this thesis because our partitioning strategies will rely on the partition results. Poor partitions will affect the results as they are ineffective in separating valuable information. As described earlier, the sections or partitions are categorized based on the primary control statements of the programs. Since all test cases in the experiments are control-intensive programs, the partitioning technique will focus on control statements. Any loops are unrolled and function calls are inlined.

The definition of primary control statements itself covers all conditionals that are located in parallel with the first expression being executed within their blocks, either in main programs or function calls. In other words, they do not have any conditional parents, instead their parents are the function declarations. As a result, there will not be any nested conditions happening among the primary control statements but it is possible within each of them.

This partitioning technique is related with the control flow graph (CFG) of a program. As described in Chapter 2.3, CFG is useful in our work as it can systematically explore the search space in a program and make a list of all paths that runs through it. In fact, all primary control statements are the main entry and exit points of all paths within the graph. For all paths within a section, they must come through these control statements in order to reach subsequent sections. By controlling all primary control statements, our strategies are able to analyze all sections more accurate and efficient.

Consider an example in Figure 3.2. It consists of 2 figures, in which the original source code running

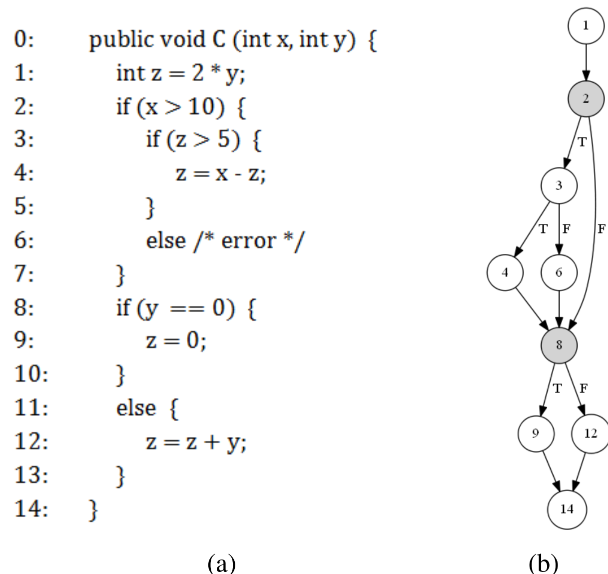


Figure 3.2: Partitioning example in C# program

sequentially is shown in Figure 3.2(a) and the translation into CFG form is shown in Figure 3.2(b). Here, the block ID is the same as the line number in the source code.

Following the rules, there are 2 primary control statements in the program (grayed out), such as $(x > 10)$ in line number 2 and $(y == 0)$ in line number 8. Thus, we have 2 sections in total. The expression assignment in line 1 is merged into the first section because its exit point refers to the first primary control statement (line 2). In general, the first section covers line 1 to 7, and the second section covers line 8 to 14. It has a total of 3 and 2 paths in the first and second sections, respectively. Note that in order for any paths in the first section traverses another path in the second section, it must go through the second primary control statement, which is line number 8. This way will control which paths can be skipped given an infeasible path from the previous sections.

Algorithm 3.1 Sequential Partitioning

```
1: Initialize and transform programs to CFG
2: Partition programs into sections
3: Ω ← total sections
4: for each sections do
5:     PathComb ← store path combinations
6: end for
7: i ← 0
8: while i < Ω do
9:     FilterPath ← filter PathComb within section i
10:    if preceding sections exist then
11:        PrevPath ← store all previous feasible paths
12: combinations
13:        FilterPath ← FilterPath ∪ PrevPath
14:    end if
15:    for each FilterPath do
16:        Transform all predicates into path constraint
17:        solve() and update all feasible paths
18:    end for
19:    increment i by 1
20: end while
```

3.2 Exploration Strategy

In this approach, we are not using either the ability of the SMT solver to generate the unsat core or any conflict-driven learning strategies to perform nonchronological backtracking. Instead, we are only interested in finding feasible paths in each section that can be combined with other paths in other sections. The first step is to analyze each path in the first section of the code and store the feasible paths to the database. The process continues forward to the subsequent section by extending each of the previously recorded feasible paths with the new paths in the subsequent sections. This process is continued until the final section is reached. A detailed algorithm of this strategy is given in Algorithm 3.1.

This sequential partitioning strategy is useful for quickly finding conflicting segments due to its ability to scan each sections meticulously starting from the beginning to the end of the program. As a result, this strategy will have a “Domino Effect” with the previous sections affecting the subsequent sections. In other words, fewer number of feasible paths in the earlier sections will lead to a smaller number of solver calls in the end.

In order to understand this effect better, we illustrate two scenarios. Suppose we have two sections with 10 paths each. The first scenario is that after analyzing the first section, we found that there were no infeasible paths in the section. Following the algorithm, all feasible paths will be extended with the paths from the second section. This results in the need to analyze a total of 100 paths when analyzing section 2 (10 paths from both the first and second sections). The total solver calls would be 110; that is, 10 solver calls in section 1 and 100 solver calls in section 2. The second scenario is having only 1 feasible path instead of 10 in the first section. This results in a huge reduction in the number of paths towards the subsequent section. In this case, we only need to analyze 10 paths in the second section (1 path from the first section combined with 10 paths from the second section). The total number of solver calls for this scenario would now be 20 (10 + 10). The difference in number of calls between the first and second scenarios is 90 calls giving a reduction of more than 80%. This process continues by extending those feasible paths found in sections 1 and 2 with the paths in section 3.

In the above example, if no partitioning is done, one might need to make more than 20 solver calls, even with conflict-driven learning, especially if the reason for the conflict returned by the solver is not the best one, that is, an unsat core fully contained in the first section.

3.3 Coverage Analysis

This strategy analyzes each section, taking advantage of infeasible paths discovered in earlier sections. All infeasible paths will not be extended for further analysis as the final results would always be the same. All feasible paths, on the other hand, will be extended to subsequent sections and this process continues until the last section. Therefore, this strategy will cover all feasible paths in the program in the same way as DFS and conflict backtracking strategy will do. However, because the sections are added gradually, the UNSAT cores for the infeasible paths will be restricted to the earlier portions of the paths.

Note that this strategy works effectively given a smaller number of feasible paths in the earlier sections due to the “Domino effect”. Sometimes it is hard to know in advance if many infeasible segments will be within the beginning sections of the code. If not, sequential partitioning will miss opportunities to prune the search space. To remedy this, we propose another approach which examines some section pairs that could lead to a significant number of infeasible paths. This second approach is described in the next chapter.

3.4 Advantages of Sequential Partitioning

The advantages of the proposed sequential partitioning are listed below.

1. Sequential partitioning strategy quickly explores all paths by analyzing the beginning segments of the code. As a result, all infeasible paths that are found can be used to skip many

paths in future search, thus reducing the execution times needed to explore especially in large programs.

2. As this strategy can discover many infeasible paths in the program through early detections, it becomes scalable because the number of paths explored will increase linearly rather than exponentially with an increasing program size.

3. Many unnecessary solver calls can be avoided due to discovering infeasible paths in the beginning section of the code.

Chapter 4

Dynamic Partitioning

The preceding sequential partitioning strategy works efficiently in programs that have infeasible path segments early in the code. However, this may not always be the case in many programs. In order to make the analysis more flexible, we need to determine those sections that are best to analyze together to maximize pruning opportunities.

In general, the dynamic strategy works in a way similar to the sequential partitioning during the initialization phase by partitioning a program into several sections based on its structures and enumerating individual paths within each section using DFS strategy. However, there are two major distinctions between these two approaches. First, this strategy uses the extracted unsat cores generated by the SMT solver for all the infeasible paths to obtain the infeasible segments. This is necessary due to the need of optimization to analyze all sections both efficiently and effectively. All infeasible segments will be recorded to avoid making the same mistake in the future search. The

procedure for extracting the unsat core is described in Algorithm 4.2. Second, we introduce metrics to determine which sections should be grouped and analyzed together. The metrics use the level of data dependencies because there might be a higher chance of getting infeasible paths in the parts of the code that have many data dependencies.

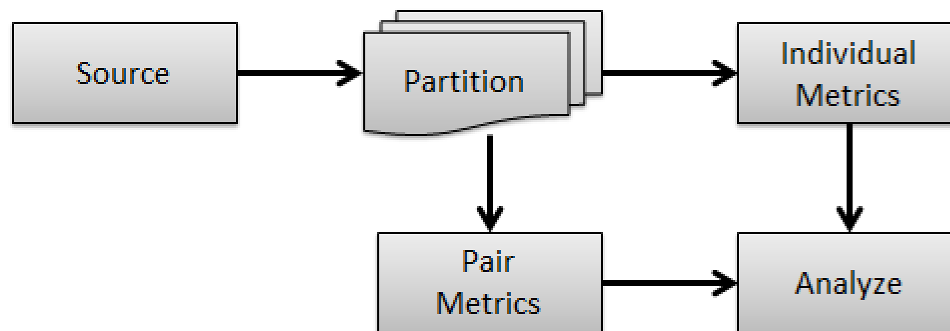


Figure 4.1: Flow diagram of dynamic partitioning

There are several phases in dynamic partitioning. A source code is provided in the beginning of the process. The same partitioning technique as in sequential strategy is applied to the program or code which will generate a number of sections. Individual and pair metrics will be calculated based on these sections which rely on their data dependencies. Analysis will be done on both sides, individually and by section pairs. In case of having a single section, individual analysis will be performed instead of pairwise analysis. The flow diagram is provided in Figure 4.1.

In our work, data dependencies are defined as the relationships between the assignment and conditional expressions of a variable in a program. A path that assigns some variables in certain statements and uses them in other conditional expressions along the path is more likely to be infeasible, when compared to paths that do not have data dependencies. In order to analyze data dependencies, we use static analysis which utilizes the abstract syntax tree (AST) created by Rosyln

[1].

Algorithm 4.1 Dynamic Partitioning

```
1:  Initialize and transform programs to CFG
2:  Partition programs into sections
3:  for each sections do
4:      PathComb ← store path combination
5:      α ← compute metrics individually
6:  end for
7:  for each α > Threshold do
8:      FilterPath ← store PathComb within α
9:      if FilterPath does not contain UnsatPath then
10:         solve() and update all feasible paths
11:     end if
12: end for
13: Ω ← list all sections
14: while |Ω| > 1 do
15:     for each Ω combinations do
16:         β ← metrics for each pairwise combinations
17:     end for
18:     SectionComb ← find the largest metrics in β
19:     FilterPath ← store path combinations in SectionComb
20:     solve() and update all feasible paths
21:     Merge sections in SectionComb and update Ω
22: end while
```

Algorithm 4.2 Constraint Solver Execution

```
1:  PathConstraint ← predicates to be added into the solver
2:  PathList ← list of a path which consists of nodes
3:  Insert PathConstraint into the solver
4:  solve()
5:  if SATISFIABLE then
6:      Extract the solution from the solver
7:      Print the results
8:  else if UNSATISFIABLE then
9:      γ ← extract the UNSAT cores
10:     core ← find the last / biggest core in γ
11:     PathList ← store each nodes in PathList and stop after reaches core
12:     UnsatPath ← add PathList
13: end if
```

First, data dependency is computed along each path in a given section in the initialization phase. If the amount of data dependency for the paths in a section is above a given threshold, then the corresponding section will be analyzed, and only those feasible paths are kept. Note that conflict-driven learning is used in this initialization phase in order to remove any paths that have conflicting segments. For example, suppose there are ten paths in a section, sequential partitioning will always analyze all ten paths. On the other hand, the dynamic approach will first check whether a path contains a previously identified unsat core, and thus may result in having less than 10 solver calls.

The next step is to compute data dependency for each section pair. The sections with the highest levels of data dependency will be chosen and merged into a single group. Note that the original code is already in SSA form, so the merged code has the correct versions of the variable instances in the original context. Since the union of both sections is considered as a new section, data dependency should be re-calculated. Due to the merger process, the total number of sections will decrease with each section merger. For example, if there are ten sections in the code initially, after sections 3 and 6 are merged to form one group, only nine sections remain. This process continues until there is only a single group left. The complete algorithm of dynamic partitioning is described in Algorithm 4.1.

4.1 UNSAT Core

An unsatisfiable core can be defined as any subset of the original formula that is unsatisfiable. Consequently, there may exist many different unsatisfiable cores, with different number of clauses, for the same problem instance, such that some of these cores are subsets of others. Also, and in the

worst case, the unsatisfiable core corresponds exactly to the set of original clauses [29].

Given an unsatisfiable CNF formula ρ, we say that an unsatisfiable CNF formula ψ is an unsatisfiable core of ρ iff $\rho = \psi \wedge \psi'$ for some (possibly empty) CNF formula ψ'. Intuitively, ψ is a subset of the clauses in ρ causing the unsatisfiability of ρ. An unsatisfiable core ψ is minimal iff the formula obtained by removing any of the clauses of ψ is satisfiable. A minimum unsat core is a minimal unsat core with the smallest possible cardinality [13].

A naive algorithm for minimal unsatisfiable core extraction works as follows: For every clause C in an unsatisfiable formula F, the algorithm checks if it belongs to the minimal core by invoking a propositional satisfiability (SAT) solver on F, but without clause C. Clause C does not belong to a minimal core if and only if the solver finds that F$\setminus\{$C$\}$ is unsatisfiable, in which case C is removed from F. In the end, F contains a minimal unsatisfiable core [18].

There are several approaches that contribute to the extraction of unsat core [29, 13, 12, 18, 37, 41]. SMT solver CVC3 and MathSAT can compute unsatisfiable cores as a byproduct of the generation of proofs which is similar to [41]. The idea is to analyze the proof of unsatisfiability backwards, and to return an unsatisfiable core that is a collection of the assumptions (i.e. the clauses of the original problem) that are used in the proof to derive contradiction. The approach used by Z3 is an adoption of the method by Lynce and Marques-Silva [29]: for each clause C_i in the problem, a selector variable S_i is created; then, each C_i is replaced by ($S_i \rightarrow C_i$); finally, before starting the search each S_i is forced to true. In this way, when a conflict at decision level zero is found by the solver the conflict clause contains only selector variables, and the unsat core returned is the union of the clauses whose selectors appear in such conflict clause.

In this thesis, a path constraint consists of several clauses including expression and conditional statements. The unsatisfiable core includes all clauses that made the path infeasible. For example, consider a path constraint of $(x = 10) \wedge (x > 5) \wedge (x < 0)$. In this case, the solver will return unsatisfiable, meaning that this path is infeasible. Furthermore, the unsat core will generate two clauses $(x > 5)$ and $(x < 0)$ as both of them cannot be satisfied together. Other paths that consist these clauses will also be infeasible. However, the extracted unsat core is not guaranteed to be minimal. In fact, there is another conflicting clauses in this path, such as $(x = 10)$ and $(x < 0)$.

In Z3, each constraint is labelled with a unique ID. Whenever the path constraint is unsatisfiable, the solver will return the IDs for all conflicting clauses that make the path infeasible. We can retrieve the clauses corresponding to the IDs and we call them unsat constraint. They will be stored in a database. Before invoking the SMT solver given a new path, we can check if any clauses within the path constraint contain any of the unsat constraint. If so, we can skip the solver call assuming this path is infeasible and move on to the next path. Using the extracted unsat core is useful in reducing many unnecessary solver calls especially in large programs.

4.2 Individual Sections

Dynamic partitioning aims to group sections in a non-sequential order. To do that, we need to determine which sections should be analyzed first. One approach is to determine which section has the largest number of infeasible paths. Data dependency plays an important role in determining such predictions since the chance of getting infeasible paths increases as the expressions are more

dependent in each other along the paths.

In this work, we use data dependencies as a metric for each section. Recall that data dependency exists in a path when some variables are used in conditional expressions and the same variables are assigned in other statements along this path. The calculation is based on the percentage of paths having data dependencies involving more than one variable. As the percentage increases, there is more data dependencies affecting those sections. Users also have the flexibility to analyze the program by controlling the minimum threshold. A lower threshold will allow more sections for merger, while a higher threshold will increase the prediction accuracy by considering fewer sections at a time.

4.3 Section Pairs

Analysis in individual sections gives a good start in reducing the number of feasible paths. We want to extend those feasible paths in a section with other paths from different sections. However, which other section should be brought in? In this work, we analyze by pairing all sections since it is only quadratic in complexity.

Even though the concept of measuring data dependency is similar between individual sections and section pairs, the process is somewhat different. Since we are interested in finding the relationship between two sections, all paths combinations should be considered. If we have two sections with ten paths each, the percentage of having data dependency for this pair is based on 100 paths. This will make a fair calculation with respect to individual metrics.

The calculation is slightly modified by other sections'presence. While the metric is calculated based on the existence of some variables used in both assignments and conditionals within a single path, it is now based on each path in each corresponding sections. Since we know there will always be two sections (a pair) that are being analyzed, there are two things that we should do. First, collect all assigned variables from the first or earliest section in the group. Second, collect all conditionals from the second or latest section in the group. Having all information needed, the calculation is based on the occurrences of having matched variables between the two sections among all path combinations. By doing so, we can easily summarize which section pairs are best to analyze with. As the metric increases, there is more data dependencies affecting these pair combinations. If we have the same metrics for different pairs, the algorithm will pick up the first pair from the list.

Analyzing sections that have the largest data dependency is necessary in order to find infeasible paths. The next step is to merge these sections into a group consisted of all path combinations with their components as well. The merged group is named after the earliest section in the group. For example, suppose we have four sections, and let sections 1 and 3 have the greatest data dependency. After the analysis, they are merged into one single group, called section 1. Section 3 is then deleted from the database, as it is absorbed into section 1. Now, the data dependency needs to be computed for section pairs (1, 2), (1, 4), and (2, 4). Since the group ordering is important, we use an intelligent algorithm to divide and merge all assignments when analyzing any selected pair of sections. When merging the sections, all the involved sections should be ordered correctly (after SSA is applied). Hence, the analysis will still be accurate even though the sections are added non-sequentially.

4.4 Improvements in Dynamic Partitioning

The dynamic strategy can analyze all sections in non-consecutive orders using individual and section pair metrics which utilize the data dependency of a program. Thus, this strategy can tackle several issues in finding more infeasible paths within different section pair combinations. As a result, we can expect to have fewer solver calls and faster execution times in general as compared to the sequential strategy and previous work.

Even though the dynamic strategy can perform both efficiently and effectively, there are also several possible extensions that can be applied directly to the current strategy in order to reach better performance. In this work, we present two possible improvements, such as dynamic partitioning using the topology of a program and sequential and dynamic transition. Both improvements still use individual and section pair metrics to analyze all sections with several modifications involved. Dynamic partitioning with topology uses data dependency along with the structure of a program to determine the best section pairs to be analyzed together. On the other hand, sequential and dynamic transition tries to combine the main characteristic of both the sequential and dynamic strategy into consideration. This section will discuss both strategies in detail.

4.4.1 Dynamic Partitioning Using the Topology of a Program

Data dependency plays a great role in finding infeasible paths of a program because any dependent objects might be conflicting with each other. In symbolic execution, conflicting means that there does not exist any valid inputs that can satisfy all constraints within the same path. The SMT solver

will return unsatisfiable and all conflicting nodes can be extracted from the unsat core. Considering the benefits of utilizing data dependency, this strategy still uses the original algorithm for calculating individual and section pair metrics to determine which sections to be analyzed.

Besides data dependency, there is another contributing factor for controlling the number of solver calls and that is the number of path combinations in section pairs analysis. In other words, the structure or topology of the program itself should be considered in our analysis. As a rule of thumb, any sections with fewer number of paths will have a smaller number of path combinations while being paired with other sections. Thus, their analysis become our first priority in order to reduce a number of solver calls in total. Those who have a larger number of paths will be analyzed in a later stage. The whole concept can be illustrated in the following example. Suppose we have 3 sections, and each of them has 10, 20 and 30 paths. In this case, there are 3 possible section pairs analysis which include (10, 20), (10, 30) and (20, 30). In order to show the benefits of using this strategy, we are only interested in analyzing the smallest and biggest pair combinations which are 200 paths (10, 20) and 600 paths (20, 30), respectively. A reduction of 50 infeasible paths from the smallest pair will produce 4500 paths (150 and 30 paths) in total after combining it with the last section. On the other hand, a reduction of 100 infeasible paths from the biggest pair, whose reduction is larger than the previous pair, will result in having 5000 paths (500 and 10 paths) in total from the last section. If the number of reduction is the same, the difference in the number of total calls between both pair combinations would be wider. This shows that analyzing sections with fewer paths might increase the chance of reducing a significant number of solver calls.

The analysis is not as simple as finding section pairs that have the smallest number of path

combinations. Failure to find corresponding pairs will result in having little to no infeasible paths and thus waste many unnecessary calls. In order to find which section pairs that can possess the maximum benefits from this strategy, data dependency can be used as a guidance for finding the relationship and behaviour between each pair. After calculating the section pair metrics, we choose the top 3 metrics that have the biggest value and collect all section pairs that correspond to them. Note that there might be different pairs under the same metric. In this case, there can be more than 3 section pairs that are collected in total. The reason for choosing only the top 3 metrics is because they mostly produce several infeasible paths during the experiments. Having more metrics involved, on the other hand, will increase the chance of choosing a section pair that does not have significant relationship. The next step is to find section pairs that has the smallest number of path combinations. These steps are needed so that we can ensure the analysis has not only a smaller number of path combinations, but also a higher chance of having infeasible paths.

This strategy also adds a new rule to the algorithm which controls the priority for analyzing all section pairs. Since we know that a smaller number of paths can reduce the combinations with other sections, the analysis should take advantage of it. During the analysis, if there is only a single feasible path exist in a particular section, all pairs that contain this section will be collected and the one with the highest metric among them will be analyzed regardless of the fact that there exist any pairs with higher dependencies. The reason for having this priority is because a single path can duplicate other sections in terms of their number of paths. Hence, it has a higher possibility to get fewer number of feasible paths after the analysis as compared to other sections that have more than a single feasible path.

4.4.2 Sequential and Dynamic Transition

As described in Chapter 3, sequential partitioning takes advantage of infeasible paths discovered in earlier sections. Thus, this strategy is useful only for programs that have infeasible segments in the beginning of the code. This limitation, however, might skip useful information that can reduce many unnecessary solver calls in future search. Dynamic strategy, on the other hand, is more flexible in analyzing all sections of the code by performing the analysis in non-sequential orders based on the data dependency of a program. However, there might be some adjacent sections that are best to analyze together which are not recognized by the data dependency metrics. These problems become the motivation behind proposing our sequential and dynamic transition strategy. In this approach, we try to merge the main characteristics from both sequential and dynamic strategies to tackle some of the previously mentioned challenges.

The main characteristic in sequential partitioning is its ability to analyze sections in a sequential order. Thus, the analysis within this new strategy will also be implemented in a sequential order. To differentiate with the original sequential strategy, we add more flexibility that the dynamic approach has into our analysis. Instead of analyzing subsequent sections in order, we analyze these sections in non-consecutive orders based on the data dependency metrics from each section pairs.

There are several steps involved in exercising this strategy. First, all individual sections are grouped in pairs in the way that the sequential partitioning works. Second, these pairs will be analyzed individually based on their metrics as similar to the dynamic partitioning approach. Finally, all analyzed section pairs will be merged together and the process is repeated until there exist a single

section during the analysis. The ordering cannot be violated meaning that all section pairs must follow some rules to prevent any sections from merging with another section that is located above them. This will ensure the program is still functionally correct. Suppose we have 5 sections with section 1 and 5 have the highest and lowest location, respectively. Using this strategy, all section pairs will be formed as (1, 2), (2, 3), (3, 4) and (4, 5). Each section pairs has its own metric and the analysis will be based on the highest rank. Let's say (2, 3) is chosen and therefore being analyzed. After performing the analysis, the grouping would now be (1, 2), (2, 4), and (4, 5). This continues until there is only a single section left.

4.5 Coverage Analysis

Our dynamic approach can achieve a full coverage similar to existing methods, even though this strategy partitions the programs into several sections and analyze them both individually and in pairwise combinations. However, we can achieve full coverage much more quickly. We maintain the coverage by exercising all possible section-pairs which is executed from line 13 to 21 in Algorithm 4.1. Here we analyze each pairwise combinations and merge them before performing the next analysis. All sections will eventually be merged together into a single section at the end. Note that all infeasible paths discovered at any time can be skipped immediately. Despite not exploring these paths further, their coverage have been recorded as they can be predetermined to be infeasible. The feasible paths, on the other hand, will always be analyzed with other feasible paths in different sections.

In addition, DFS and conflict backtracking strategies call the SMT solver on complete paths, starting from top to the bottom of the programs. Using our dynamic partitioning, all feasible complete paths will also be identified eventually. Note that all programs should be translated into SSA form before performing the analysis.

Theorem 4.5.1 *If a contiguous path is feasible, our dynamic approach will never declare any subset of this path as infeasible.*

Proof Given a feasible contiguous path, the conjunction of all symbolic expressions along the path will form its corresponding path constraint. If this path constraint is feasible, then there exists a set of valid inputs that can satisfy all constraints on this path. Thus, any subset of this path constraint will also be satisfied by this valid input set.

Suppose there is a formula $f = (\text{S1, S2, S3})$ which intersects with 3 sequential sections (S1, S2, and S3) in a program. If f is satisfiable, then any subset of sections along this path is also satisfiable, even if an intermediate section is not considered in f.

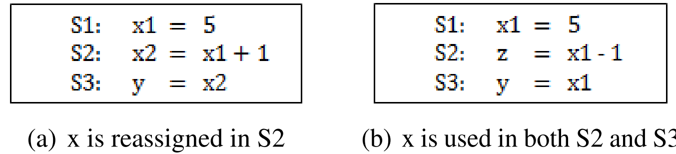
Theorem 4.5.2 *If a contiguous path is infeasible, our dynamic approach may declare any subset of this path as either feasible or infeasible.*

Proof Given an infeasible contiguous path, its path constraint would be unsatisfiable. For every unsatisfiable instance, there exist one or more unsatisfiable cores. If an unsat core is fully contained in the set of constraints for a subset of the path in question, then this subset would also be

unsatisfiable. However, if no unsat core is contained in this subset, this subset may be declared as feasible. Therefore, our approach does not have the capability to determine in advance which subset of the path is conflicting before analyzing the extracted UNSAT core generated from the SMT solver.

Consider an infeasible contiguous path whose path constraint is $f = (\text{S1, S2, S3})$. Suppose an UNSAT core involves nodes in S1 and S2 as they cannot be satisfied together. Hence, a path analysis involving S1 and S3 might yield satisfiable since there is not enough information to make such a decision. This is not a bad result, since this shows that our approach is conservative. That is, we will never mistake a feasible path as infeasible.

In order to understand how arbitrary sections can be merged, we will analyze two cases as shown in Figure 4.2. In both examples shown in the figure, S1 and S3 will be merged into a single section to be analyzed first, leaving S2 to be analyzed later. In both cases, variable x is either reassigned or used in all sections, making this variable is heavily dependent across the sections. Before partitioning is performed, the program is first converted to SSA form. In Figure 4.2(a), the path constraint after merging S1 and S3 would be $(x1 = 5) \wedge (y = x2)$. In this case, $x2$ is treated as an independent input variable, since its definition is in S2. As a result, the analysis will be based on over-approximation because the real $x2$ is not being considered. On the other hand, Figure 4.2(b) shows that variable $x1$ is being used in both sections S2 and S3. The use of $x1$ in S2 will have no effect on the merging S1 and S3 since there is no data dependency among all sections. Thus, our approach is sound as it will eventually identify all infeasible paths as well.



(a) x is reassigned in S2 (b) x is used in both S2 and S3

Figure 4.2: An example of how arbitrary sections can be merged after translating into SSA form

Our dynamic approach also orders the sections to be analyzed. This must be done because the program runs in a sequential order. Changing the order of the sections must not change the structure of the programs. Given an example in Figure 4.2, after merging S1 and S3, we will add S2 into our analysis and place it between S1 and S3. Therefore, when this analysis is conducted, the path will traverse from S1 to S3 sequentially as in their original order.

4.6 Advantages of Dynamic Partitioning

The advantages of the proposed dynamic partitioning are listed below.

1. Dynamic partitioning can cover all paths more quickly than sequential partitioning because it can find infeasible path faster even though it needs to partition the programs and analyze them both individually and in pairwise combinations.

2. Dynamic partitioning is more flexible than the sequential partitioning because it can analyze sections in non-consecutive orders giving the former more chances to find infeasible paths in a program. This strategy also has individual and section pairs analysis which give thorough analysis on all sections.

3. Similar to sequential partitioning, this technique is also suitable for large programs because it can skip many infeasible paths.

4. Data dependency plays a great role in finding infeasible path and thus reducing many unnecessary solver calls.

Chapter 5

Experimental Results

The experiments were performed on a 3.2GHz Intel Xeon machine with 8GB of RAM. They were tested on a number of control-intensive C# test cases. Any loops are unrolled and we analyze only the body of the loop to demonstrate the scalability of the proposed approach. The first set of test cases are generated randomly with the number of conditionals as the controlling factor. The values of several variables are modified to be used in later parts of the programs to create data dependencies among those variables. A second set of test cases consisted of real-world programs are also reported. Tables 5.1 and 5.2 consist of the random test cases that are extracted into 3 and 6 sections, respectively. We compared the number of SMT solver calls by our partitioning strategies with the traditional Depth-First Search (DFS) strategy and the previous work which uses conflict-driven backtracking strategy (DFS + CDL) [28]. The speedup was calculated to see the impact of our partitioning strategies (sequential and dynamic approach) to the recent work (DFS + CDL). # **conds** denotes the number of conditionals in the test cases. In order to make a

fair comparison, we used the same SMT solver, Z3 from Microsoft Research [16], to evaluate all strategies in each test cases.

In this experiments, we set the threshold for our dynamic approach to be 80%, because we prefer a more accurate prediction by analyzing sections that have many data dependencies resulting in less sections that will be analyzed individually. A lower threshold will increase the number of solver calls due to the fact that the strategy will analyze more sections but might end up with having fewer infeasible paths. Increasing the threshold all the way up to 100%, on the other hand, will not reduce the number of calls as well because having a section that has 100% data dependency for each of its path is quite unlikely and thus will skip the individual analysis.

Table 5.1 shows the comparisons between our partitioning strategies (code divided into 3 sections) with the recent work (DFS + CDL). The DFS strategy is used for evaluating the total number of paths available in each test cases. In other words, the computation cost for DFS would be the worst among others due to the lack of search optimization and will not be considered for computing the speedup. Instead, the speedup achieved is compared only against (DFS + CDL).

It can be seen from Table 5.1 that even sequential partitioning can outperform the conflict backtracking strategy in most cases. Our dynamic partitioning achieves another substantial speedup over sequential partitioning for many instances. The first test, tc1, is one of the smallest programs in our experiments with only 12 conditional statements. The conflict backtracking approach performs worse than our dynamic strategy but better compared to the sequential strategy. Results for all other test cases showed speedups from our approach, indicating that the non-partitioning strategy does not scale for larger programs. This test case, tc1, also has fewer data dependencies making

Table 5.1: Results for 3-Section Partitions

Test Case	# conds	DFS	DFS+CDL		Sequential			Dynamic		
			calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
tc1	24	5184	129	5.4	88	5.42	-	47	4.28	1.26
tc2	24	5184	471	13.8	350	12.36	1.16	114	6.34	2.17
tc3	32	50176	7612	120	204	8.1	14.81	77	5.21	23.03
tc4	44	321489	25269	600	639	22.07	27.18	140	6.7	89.55
tc5	48	531441	29695	660	405	13.95	47.31	127	6.19	106
tc6	48	531441	33515	780	729	24.86	31.37	153	7.06	110
tc7	52	810000	44604	1020	850	30.62	33.31	169	7.26	140
tc8	54	705600	68755	1560	1320	32.39	48.16	212	8.72	179
tc9	58	1.1M	87880	1980	1491	57.58	34.38	225	8.87	223
tc10	62	1.8M	145742	3660	1944	80.08	45.7	261	9.71	377
tc11	64	2.2M	186005	4560	2166	95.09	47.95	286	10.58	431
tc12	68	3.4M	-	-	2386	123	> 58.53	350	12.94	> 556

’-’: timeout of 2 hours

dynamic strategy unable to take full advantage as its ability is based on the relationship between all section pairs. With three sections, the sequential partitioning strategy can achieve up to nearly 2 orders of magnitude speedup while the dynamic partitioning strategy was able to achieve up to nearly 3 orders of magnitude speedup over DFS + CDL. In fact, as the programs get bigger and data dependencies starts playing a role, dynamic approach achieved greater speedups over the DFS + CDL strategy compared to sequential approach. It can also be seen that as the programs become bigger, the speedup increases, suggesting that the differences on the number of solver calls between our strategies and DFS + CDL are growing wider. The largest speedup was seen in tc12, in which the dynamic approach was faster more than 556× than DFS + CDL.

Table 5.2 reports results for the same test cases that are divided into 6 sections. Both of our partitioning strategies showed effectiveness in dealing with large programs by having significant speedups in most cases. The biggest speedup was achieved by dynamic approach in tc12 that has more than 3.4 million paths which is able to perform 791× faster than DFS + CDL. For this test case, the DFS + CDL timed out after 2 hours. While DFS + CDL were able to perform better compared to the sequential strategy in tc1 from Table 5.1, it performed worse in Table 5.2 for the same test case showing that dividing more sections in a program makes the partitioning strategy more effective.

In some cases, even when our partitioning strategies have the same number of solver calls as DFS+CDL, the execution times can be quite different. This is because with partitioning, many of the paths that need to be solved involve only a subset of the sections rather than the complete path, which result in shorter formula with fewer path-predicates. In tc8 from Table 5.1 and tc9 from Table

Table 5.2: Results for 6-Section Partitions

Test Case	# conds	DFS	DFS+CDL		Sequential			Dynamic		
			calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
tc1	24	5184	129	5.4	47	3.66	1.59	44	3.63	1.61
tc2	24	5184	471	13.8	201	6.9	1.99	127	5.49	2.52
tc3	32	50176	7612	120	69	4.15	28.91	59	4.01	29.92
tc4	44	321489	25269	600	159	6.02	99.67	111	5.14	117
tc5	48	531441	29695	660	99	4.66	142	105	4.7	140
tc6	48	531441	33515	780	171	6.39	122	111	5.13	152
tc7	52	810000	44604	1020	184	6.65	153	147	6.24	163
tc8	54	705600	68755	1560	296	9.36	167	202	7.18	217
tc9	58	1.1M	87880	1980	313	9.84	201	222	7.97	248
tc10	62	1.8M	145742	3660	378	11.68	313	256	8.65	423
tc11	64	2.2M	186005	4560	412	12.6	362	235	8.42	541
tc12	68	3.4M	-	-	428	13.59	> 530	255	9.1	> 791

'-': timeout of 2 hours

5.2, for instance, the total number of solver calls for both test cases in our dynamic strategy were 212 and 222, respectively. However, tc9 had a smaller execution time compared to tc8 because more predicates are included on average in Table 5.1 than in Table 5.2 since it partitions fewer sections (3 sections) within the same program.

increasing the number of sections from 3 to 6 generally will reduce the total number of solver calls, which also significantly reduce the execution times needed. By having fewer sections (each section is bigger), the total number of individual paths that need to be analyzed will become larger, increasing the overall analysis time. In tc12, for instance, the number of solver calls in Table 5.2 for tc12 was only 428, which is $5\times$ smaller than 2386 in Table 5.1, resulting in a $9\times$ faster execution than the 3-section partition. On the other hand, dynamic strategy had an average speedup of 1.4 in large programs from increasing the number of sections. This is due to the fact that dynamic strategy has exercised the conflict driven learning that is applied to every analysis in both individual and section pairs.

The execution times taken for DFS + CDL for all test cases are reported in minutes rather than in seconds. In Table 5.2, sequential and dynamic strategies took less than 15 and 10 seconds, respectively, for all test cases while DFS + CDL took less than a minute for only 2 test cases (tc12 and tc22). Starting from tc32, the number of calls performed by DFS + CDL increased significantly which also significantly increased the execution times.

As described in Section 4.4, there are 2 possible extensions that can be applied to the dynamic partitioning strategy to improve the performance. Table 5.3 reports results from comparing these extensions with other strategies for the same test cases that are divided into 6 sections. We know that

Table 5.3: Comparing Dynamic Improvements with Current Strategies

Test Case	# conds	DFS	DFS+CDL	Sequential	Dynamic	Dyn+Topo	Seq+Dyn
tc1	24	5184	129	47	44	36	53
tc2	24	5184	471	201	127	87	159
tc3	32	50176	7612	69	59	59	88
tc4	44	321489	25269	159	111	83	104
tc5	48	531441	29695	99	105	77	105
tc6	48	531441	33515	171	111	119	230
tc7	52	810000	44604	184	147	135	123
tc8	54	705600	68755	296	202	204	192
tc9	58	1.1M	87880	313	222	208	197
tc10	62	1.8M	145742	378	256	242	247
tc11	64	2.2M	186005	412	235	233	336
tc12	68	3.4M	-	428	255	257	218

’-’: timeout of 2 hours

the number of partitions makes the partitioning strategy more effective and thus we use 6-section instead of 3-section partitions. The experiments were conducted to show the number of solver calls needed for each tests. **Dyn+Topo** represents the extension that uses the topology of a program on top of the dynamic strategy. **Seq+Dyn** denotes the sequential and dynamic transition strategy.

Both extensions cannot perform well in all test cases due to the characteristic of the programs. However, these extensions show great promise in reducing many unnecessary solver calls in most test cases. In tc2, for instance, the total number of solver calls in the dynamic with topology strategy was 87, which is 40 calls smaller than 127 in the original dynamic strategy. In this case, there are several section pairs that have not only a smaller number of path combinations but also a higher data dependencies. As a result, the probability of having infeasible paths from these pairs increases. This,

Table 5.4: Performance Comparison of All Partitioning Strategies Based on Table 5.3

Test Case	Fewer Calls → More Calls			
	1	2	3	4
tc1	D+T	D	S	S+D
tc2	D+T	D	S+D	S
tc3	D & D+T		S	S+D
tc4	D+T	S+D	D	S
tc5	D+T	S	D & S+D	
tc6	D	D+T	S	S+D
tc7	S+D	D+T	D	S
tc8	S+D	D	D+T	S
tc9	S+D	D+T	D	S
tc10	D+T	S+D	D	S
tc11	D+T	D	S+D	S
tc12	S+D	D	D+T	S

in return, will reduce a significant number of path combinations with other sections and feasible paths as well, thus a smaller number of solver calls can be reached.

The sequential and dynamic transition strategy was able to perform effectively in tc12. In this test, most infeasible paths can be found during the analysis on adjacent sections. In fact, based on the pair metrics, the analysis is similar to the sequential partitioning strategy as it flows in a sequential order. However, the original sequential strategy has 428 solver calls in total, which is a lot more than this strategy has, which is 218 calls. This indicates the benefits of using individual analysis as well as the extracted unsat core for reducing many unnecessary solver calls.

Table 5.4 shows the comparison of our partitioning strategies based on their number of solver calls from Table 5.3. Group 1 has the best performance for having the least amount of calls while

group 4 has the worst performance for having the most number of calls among all strategies. The abbreviations used in this table are listed as follows: S = Sequential Partitioning; D = Dynamic Partitioning; D+T = Dynamic Strategy with Topology; S+D = Sequential and Dynamic Transition.

The first improvement in dynamic strategy that uses the topology of a program performs better in most cases by having the smallest number of solver calls in 7 out of 12 test cases. This indicates that the topology of a program should be considered in our analysis to reduce more solver calls. The second improvement, sequential and dynamic transition strategy, has the best performance in 4 out of 12 test cases. However, this strategy shows unstable behaviour during the experiments. Even though it performs better than other strategies in several cases, it also performs the worst in other 4 test cases. The main reason is because the characteristic is similar to the sequential strategy which in most cases performs the worst than all partitioning strategies. In some programs, adjacent sections are not the best to be analyzed together. To make things even worse, the individual analysis contributes to more costs if there does exist only a small number of infeasible paths, in which the sequential partitioning can avoid.

So far, all test cases are generated randomly based on the number of conditionals resulting in a lower number of feasible paths. Since our motivation is to handle the unsat core problems that the previous work suffers from, having more infeasible paths will increase the utility of using our partitioning strategies. As a result, both Table 5.1 and 5.2 have shown our effectiveness by having significant speedups in most test cases.

In order to see the effect of our strategy in real programs, we chose five benchmarks as shown in Table 5.5 and see the performance of both the previous work (DFS + CDL) and sequential and

Table 5.5: Results for Real Programs

Test Case	# conds	# sections	DFS	DFS+CDL		Sequential			Dynamic		
				calls	Time (s)	calls	Time (s)	Speedup	calls	Time (s)	Speedup
score	11	11	2048	276	8.83	22	2.99	2.95	22	3.3	2.67
score_2	22	22	4.1M	-	-	44	5.37	> 1340	44	5.2	> 1387
sparsemtx	16	16	65536	145	10.08	32	3.63	2.77	31	3.37	2.99
sparsemtx_2	32	32	4.3B	-	-	64	5.27	> 1366	63	4.89	> 1472
sequence	42	7	823543	5538	143	49	3.6	39.72	42	3.42	41.81
sequence_2	84	14	> 500B	-	-	98	6.89	> 1044	91	5.94	> 1212
bubblesort	28	28	268M	77	4.61	56	3.68	1.25	55	3.76	1.22
bubblesort_2	56	56	> 1000B	229	14.47	112	9.23	1.56	111	8.87	1.63
shellshort	61	61	536M	218	12.85	122	8.94	1.43	121	7.65	1.68
shellshort_2	122	122	> 1000B	751	62.51	244	34.87	1.81	243	32.38	1.93

'-': timeout of 2 hours

dynamic strategies. These programs consist of arrays, intensive loops and arithmetic/relational operations. Since our strategy analyze the loop-free portions of the code, any loops need to be unrolled before being able to start the analysis. We also duplicated their structures to assess the performance of all strategies in large programs. # **sections** denotes the number of partitions in the test cases.

It can be seen from Table 5.5 that both sequential and dynamic strategies can outperform the previous work in all cases. In some cases, such as score, sparsemtx, and sequence, where the loop unrolling is doubled, DFS + CDL was timed out resulting in a speedup of more than 1000$\times$ with both strategies. This shows that the amount of savings is not linear to the size of the program, but could increase exponentially.

Score consists of multiple cascaded IF-ELSE statement without any nested conditions. Sequential and dynamic strategies have the same number of solver calls because any unreachable IF condition will result in skipping the same number of subsequent unnecessary solver calls. However, DFS + CDL was not able to produce the same result as this strategy does not always get the best UNSAT core for an infeasible path. In addition, there is no nested conditions in bubblesort and shellsort making sequential and dynamic strategies had similar results. The speedup was similar even after duplicating the structures since the previous work was still able to reduce many solver calls and avoid many unsat core problems.

Chapter 6

Conclusion and Future Work

In recent years, symbolic execution is becoming popular for path exploration technique. However, it still suffers from scalability issues especially in large programs. In this thesis, we presented two partitioning strategies for improving symbolic execution by reducing the number of unnecessary solver calls while still cover all paths. Sequential partitioning analyzes the program in sequential orders based on the feasible paths discovered from earlier sections. Dynamic partitioning improves the flexibility by introducing metrics to determine which sections should be grouped and analyzed together. This strategy also uses conflict-driven learning to avoid redundant paths.

Both strategies were tested on several randomly generated test cases and real programs as well. Experimental results show that our partitioning strategies are able to achieve significant speedups in most cases especially in large programs. Dynamic approach performs better among all strategies due to its ability to perform in non-sequential ordering of program sections based on data dependency

metrics. The number of solver calls were significantly reduced, especially for larger programs.

The results also show that the number of partitions affects the performance. Having more partitions gives both partitioning strategies better speedups. In fact, sequential partitioning has significant impacts rather than dynamic strategy since it relies heavily on the results from each sections. Dynamic partitioning, on the other hand, does not have much effects because it is able to analyze all sections meticulously and thoroughly.

Two improvements have been proposed on the dynamic partitioning strategy which utilize the topology of a program and the main characteristics from both sequential and dynamic strategies. Using the topology of a program, we show that more solver calls can be reduced as it has the best performance in most cases. The sequential and dynamic transition strategy shows unstable behaviour and is subject to further testing in order to get better results.

Although the concept of partitioning strategy is applicable to all structures in the programs, the current strategies target only for arithmetic operations. In the future, the algorithms can be modified to handle broader classes of programs. The current algorithms also cannot handle loops and inline functions. Further analysis should be done on semantic and data flow analysis to overcome these problems. In addition, the calculation for data dependency metrics in dynamic strategy can be investigated for better results, thus reducing more unnecessary solver calls.

Bibliography

[1] .net compiler platform ("roslyn"). https://roslyn.codeplex.com/.

[2] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[5] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[6] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[7] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. Van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Computer Aided Verification*, pages 335–349. Springer, 2005.

[8] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[10] X. Cheng and M. S. Hsiao. Simulation-directed invariant mining for software verification. In *Proceedings of the conference on Design, automation and test in Europe*, pages 682–687. ACM, 2008.

[11] A. Cimatti. Beyond boolean sat: Satisfiability modulo theories. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 68–73. IEEE, 2008.

[12] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 334–339. Springer, 2007.

[13] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, pages 701–728, 2011.

[14] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[17] L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009.

[18] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. *SAT*, 4121:36–41, 2006.

[19] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[20] J.-C. Filliâtre, S. Owre, H. Rue, N. Shankar, et al. Ics: Integrated canonizer and solver? In *Computer Aided Verification*, pages 246–249. Springer, 2001.

[21] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68. Seattle, 2000.

[22] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[23] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[24] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 48–58. ACM, 2013.

[25] I. Johnson. Formal verification with smt solvers: Why and how. 2009.

[26] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

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

[28] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *Test Symposium (ATS), 2010 19th IEEE Asian*, pages 59–64. IEEE, 2010.

[29] I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. In *International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.

[30] K. L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, pages 104–118. Springer, 2010.

[31] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[32] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[33] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 195–200. ACM, 1996.

[34] C. Pacheco and M. D. Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

[35] S. Prabhu, M. S. Hsiao, S. Krishnamoorthy, L. Lingappan, V. Gangaram, and J. Grundy. An efficient 2-phase strategy to achieve high branch coverage. In *Test Symposium (ATS), 2011 20th Asian*, pages 167–174. IEEE, 2011.

[36] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 117–126. IEEE, 2011.

[37] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 174–187. Springer, 2011.

[38] K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[39] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM (JACM)*, 26(2):351–360, 1979.

[40] N. R. Vempaty, V. Kumar, and R. E. Korf. Depth-first versus best-first search. In *AAAI*, pages 434–440, 1991.

[41] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.

[42] L. Zhang, M. R. Prasad, and M. S. Hsiao. Incremental deductive & inductive reasoning for sat-based bounded model checking. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 502–509. IEEE Computer Society, 2004.

[43] R. Zhou and E. A. Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *IJCAI*, volume 9, pages 640–645, 2009.