Symbolic Execution with Over-Approximation

Yude Lin
orcid.org/0000-0001-7501-0885

Supervisors:
Tim Miller
Harald Søndergaard
Toby Murray

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the
School of Computing and Information Systems

December 2017
Abstract

Symbolic Execution with Over-Approximation

by Yude Lin

Program verification is difficult but crucial in establishing software’s reliability and security. As the software industry grows rapidly, traditional verification methods, such as code reviewing, inspection, and manual testing, become laborious and less effective. Symbolic Execution (SE) is an important automation technique in program verification. SE can analyse a program automatically and obtain information about the feasible paths in the program. For each such path SE captures the condition on the inputs to reach the path, as well as the corresponding program behaviour. SE has received much attention from researchers in the last decade, and is used effectively in the software industry.

The biggest problems with symbolic execution are path explosion (efficiency) and the sophistication of program semantics or software systems (precision). In order to make SE scalable, existing SE approaches frequently use under-approximation—a class of techniques that narrow the search space of SE, such as concretisation and aggressive path pruning. This thesis shows that over-approximation can be more advantageous in some cases, especially when we can differentiate unimportant or irrelevant code from the more critical or interesting code.

In our research, we study the strengths and weaknesses of over-approximation adaptations in SE. We introduce novel techniques based on two types of over-approximation, namely “contextual” and “regional” over-approximation. Contextual over-approximation focuses on symbolically executing specific parts of a program, while disregarding the context information; this particular feature makes results reusable, thus reducing SE redundancy. Regional over-approximation (over-approximating the details) can focus SE on program parts that are the most beneficial (e.g., implying the most code coverage), while disregarding irrelevant details. This saves the cost of SE when the code can be better differentiated, and shows great promise in targeted SE or proof-oriented applications. A drawback of techniques that use over-approximation is that they introduce infeasible program states which could mean false positives. The cost of eliminating such false positives is substantial. We show how incremental constraint solving techniques, in particular, assumption checking is a potential solution in this case. We base our discussion in this thesis on three innovative implementations of SE that centre around the above ideas.
Declaration

I, Yude LIN, declare that

- This thesis comprises only my original work towards the degree of Doctor of Philosophy.
- Due acknowledgement has been made in the text to all other material used.
- This thesis is fewer than 100,000 words, exclusive of tables, maps, bibliographies, and appendices.

Signed: 

Date: 
Preface

The work towards this thesis was carried out in collaboration with my supervisors, Tim Miller, Harald Søndergaard, and Toby Murray. It was solely conducted when I was in the School (formerly Department) of Computing and Information Systems, University of Melbourne. I was mainly responsible for the initial ideas, the implementation of tools, the experiment design and execution, and the writing of draft papers, while my supervisors took part in the above by suggesting general directions, more thoughts, requirements, feedback, and revising the writings. Part of this thesis has appeared in the following papers:


My study has been supported by China Scholarship Council (CSC), Melbourne International Fee Remission Scholarship (MIFRS), and a top-up from The University of Melbourne.
Acknowledgements

I would like to thank my supervisors Tim Miller, Harald Søndergaard, and Toby Murray for their support during my study. I am very grateful to Tim, who is my principal supervisor and who provided me with this awesome opportunity to undertake research at Melbourne University. Tim has given me so much insightful advice, and constantly shows excitement and interests in my research. Tim is always the quickest in understanding me, which is always a great comfort to me. Tim also cares about my personal development and introduced me to Oracle Labs Australia, which has become another valuable experience to me. I have to thank Harald for being very attentive to me, such as giving me the most detailed feedback on my writings and most supportive looks during my seminar talks. I will also not forget the lectures (personal or in public) and the books he gave me, from which I learned a great deal. I think I must have taken a lot of time from him when asking him to lend me help, but he is just such a knowledgeable and pleasant man to spend time with. I will also thank Toby for coming to rescue in my last year of PhD. I am very thankful for his kindly sharing of opinions regarding my current research, potential direction, thesis writing, and raising questions, some of which I wish I have known earlier. I thank my committee chairs, Trevor Cohn, Andrew Turpin, and Steven Bird, for being patient listening to my reports and giving helpful comments.

When I worked as an intern in Oracle, I met Andrew E. Santosa to whom I also owe my gratitude. Andrew led our team for most of my internship, during which time he taught me in depth about Java, JDK, Reflection API, DSE, Prolog, writing our own constraint solver, operational semantics, and many other things that are useful to a researcher as well as a developer. Andrew made my internship even more memorable by being an good friend with me. During my stay in Brisbane I also got to know another of my team leaders Lian Li, who took care of me from the start and I wish we have worked together for longer; Yi Lu, who was also being a nice friend and took me around the city for lunch; Paddy Krishnan, my external supervisor, who has been advising me about my work and my talks; Cristina Cifuentes, who has been paying attention to my progress as well and willing to share industry information; Juliette Hatton, who has found me the best accommodation and shown concern over my well-being; many other researchers and fellow interns. I thank them all for taking a part in this exciting period of my life.

I have to acknowledge the brilliant academics and students in CIS. I learned a lot when spending time with Jiancong Tong, who shared his wisdom and adventure in Melbourne with me. He will be a person I always look up to. I am also grateful for being able to know Qingyu Chen, who is very humorous, generous, willing to exchange thoughts, and who taught me a lot about tutoring. I am thankful for being surrounded by students with a good heart, such as Juan Lu, Rehan Abdul Aziz, Nick Downing, Kathryn Francis, who made it so much easier for me as a new student. I thank all the people in Doug McDonell in particular 6.22 for creating a friendly working environment. I thank the CIS Postgraduate Group for the fun trips and good movies we had and watched together. I have to particularly thank the VASET people, including Eman Alatawi, Knobby Clarke, Peter Schachte, Lee Naish, and of course my supervisors, for being my role models as presenters and the best audience.

I would also thank Xianwen Shang and his family for being warm hosts for us friends in Melbourne.

Finally, I want to thank my selflessly loving parents. I am very fortunate to have them by my side whenever I need, cheering for my success and worrying when I worry. In these four years, they gave me both support and pressure, which is the first and foremost reason I have come this far.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td>iii</td>
<td></td>
</tr>
<tr>
<td><strong>Declaration</strong></td>
<td>v</td>
<td></td>
</tr>
<tr>
<td><strong>Preface</strong></td>
<td>vii</td>
<td></td>
</tr>
<tr>
<td><strong>Acknowledgements</strong></td>
<td>ix</td>
<td></td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.1 Symbolic Execution and Its Challenges</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.1.1 Existing Solutions</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.1.2 Our Observation</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1.1.3 Problem Definition</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1.4 Subject Languages Used Throughout the Thesis</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2.1 Program Verification, Testing and Fuzzing</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2.1.1 Program Verification</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2.1.2 Program Testing</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2.1.3 Fuzzing</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2.2 Constraint Solving and Solvers</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2.2.1 Constraint Solving Problems and Algorithms</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2.2.2 Incremental Solving</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2.3 Symbolic Execution</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2.3.1 Objectives of SE</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>2.3.2 Faithful Optimisations on Symbolic Execution</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2.3.3 Under-Approximation</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2.3.4 Over-Approximation</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2.3.5 Hybrid Approaches</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td><strong>3 Compositional Symbolic Execution and an Improvement</strong></td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.1 Background</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.1.1 Introduction to Compositional Symbolic Execution</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3.1.2 Direction of Symbolic Execution</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>3.2 Motivation</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>3.2.1 Improving Efficiency</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>3.2.2 Combination with Backward Symbolic Execution</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>3.2.3 Contributions</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>3.3 Fine-Grained Compositional Symbolic Execution</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3.3.1 Variable Versioning</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3.3.2 Entry Condition and Postcondition</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>3.3.3 Concatenation</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>3.3.4 De-Concatenation</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>3.3.5 Composing Path Constraints</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>3.3.6 The Loss of Context</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>3.4 The Problems of Using Summaries</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>3.4.1 Program Characteristics and Summarisation</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>3.4.2 Strategies for Summarisation</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>
3.5 Implementation ................................................. 53
  3.5.1 Summarisation and the Call Ratio Threshold ........ 54
  3.5.2 The Backward Reasoning Method ......................... 54
3.6 Preliminary Evaluation ........................................ 55
  3.6.1 Experiment Design ....................................... 55
  3.6.2 Results .................................................. 56
  3.6.3 Discussion ............................................... 56
3.7 Conclusion and Future Work ................................. 59

4 Stratified Symbolic Execution ............................... 61
  4.1 Motivation ................................................. 61
    4.1.1 Limitations in Understanding and Configuring Search Space 61
    4.1.2 Differentiating the Roles of Code within a Program .... 62
    4.1.3 Verifying Programs Partially ............................ 62
    4.1.4 Optimising Symbolic Execution by Reordering ........... 64
  4.2 Method—Stratifying a Program ............................ 65
    4.2.1 Invocation Depth ..................................... 66
    4.2.2 Skippable and Non-Skippable Function Calls .......... 67
    4.2.3 Skipping and Resolving and the Over-Approximation .... 68
    4.2.4 SE in a Stratified Program ............................. 73
    4.2.5 The Symbolic Execution Tree with Skipping and Resolving* 74
    4.2.6 Convergence to Standard SE ............................ 75
  4.3 A Difficulty and a Solution—The Stratified Searcher ....... 75
    4.3.1 Search Heuristics and Discard Policy ................. 75
    4.3.2 Must-Instructions and May-Instructions ............... 76
    4.3.3 The Search Algorithm in Stratified Symbolic Execution 78
  4.4 Applications ............................................... 80
    4.4.1 Exhaustive Symbolic Execution ........................ 80
    4.4.2 Targeted Exploration .................................. 80
    4.4.3 Partially Proving a Program ........................... 83
    4.4.4 Limitations ........................................... 84
  4.5 Evaluation .................................................. 84
    4.5.1 Evaluating Exhaustive Exploration with Automatic Skipping 85
    4.5.2 Further Analysis ..................................... 87
    4.5.3 Evaluating Targeted Exploration ....................... 91
    4.5.4 Threats to Validity ................................... 95
  4.6 Future Work ............................................... 96
    4.6.1 Identifying Functions Suitable for Skipping .......... 96
    4.6.2 Seeding with Stratified SE ............................ 96
    4.6.3 Skipping Functions in Common Libraries ............... 96
  4.7 Conclusion .................................................. 97

5 Handling Over-Approximation Using Incremental Solving .... 99
  5.1 Introduction ............................................... 99
    5.1.1 Compositional Symbolic Execution .................... 99
    5.1.2 Incremental Constraint Solving ....................... 100
    5.1.3 The Attempts to Use Incremental Solving in SE ....... 100
    5.1.4 Assumption Checking .................................. 102
    5.1.5 Chapter Outline ...................................... 102
  5.2 Motivation for Incremental Solving in CSE ................ 102
    5.2.1 Compensation for the Shortcoming of Summarisation 102
    5.2.2 The Potential Value of Summaries .................... 105
  5.3 Implementation ............................................. 108
    5.3.1 A More Sophisticated Implementation of FGCSE ........ 108
    5.3.2 Integration of Incremental Solving into (Compositional) SE 109
    5.3.3 Limitations .......................................... 110
  5.4 Evaluation .................................................. 111
    5.4.1 Setup ............................................... 111
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example illustrating symbolic execution.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A comparison of under-approximation and over-approximation.</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>An example illustrating the use of a very general over-approximation.</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>A high-level summary of the thesis and the tool implementations.</td>
<td>8</td>
</tr>
<tr>
<td>1.5</td>
<td>Ozpin syntax.</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>An example illustrating unsound concretisation.</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>An example illustrating the importance of exploring every path in SE.</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>A motivating example for fine-grained summaries.</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplified CFG of a loop body illustrating the use of fine-grained summaries.</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>Generating a condition pair for each individual instruction.</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Example code fragment.</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>An example of building condition pairs.</td>
<td>47</td>
</tr>
<tr>
<td>3.6</td>
<td>An example of the loss of context.</td>
<td>48</td>
</tr>
<tr>
<td>3.7</td>
<td>Illustration of summarisation strategies.</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Differentiating the roles of code: traditional depth vs reality.</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>A simplified OpenSSL code snippet.</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>How Stratified Symbolic Execution captures the different layers of a program.</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Example program illustrating skipping and resolving.</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>The symbolic states during skipping and resolving.</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Example program illustrating skipping/resolving of a function taking a pointer.</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>The symbolic states during skipping/resolving of a function taking a pointer.</td>
<td>72</td>
</tr>
<tr>
<td>4.8</td>
<td>Example program with may-instructions.</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>Discarding tests in SE for a program with may-instructions.</td>
<td>77</td>
</tr>
<tr>
<td>4.10</td>
<td>Must- and may-coverage of states.</td>
<td>78</td>
</tr>
<tr>
<td>4.11</td>
<td>The prioritisation function in Stratified SE (relative to ( \text{Score}(\eta) )).</td>
<td>79</td>
</tr>
<tr>
<td>4.12</td>
<td>Using selective resolving to do targeted exploration.</td>
<td>81</td>
</tr>
<tr>
<td>4.13</td>
<td>Example program illustrating selective resolving.</td>
<td>82</td>
</tr>
<tr>
<td>4.14</td>
<td>The symbolic states during targeted exploration using Stratified SE.</td>
<td>82</td>
</tr>
<tr>
<td>4.15</td>
<td>Comparison of Stratus and KLEE in exhaustive and unsupervised exploration.</td>
<td>86</td>
</tr>
<tr>
<td>4.16</td>
<td>The ten most skipped functions in the experiment.</td>
<td>87</td>
</tr>
<tr>
<td>4.17</td>
<td>Code snippet of program <code>unexpand</code>.</td>
<td>89</td>
</tr>
<tr>
<td>4.18</td>
<td>Case study of program <code>unexpand</code>.</td>
<td>90</td>
</tr>
<tr>
<td>4.19</td>
<td>Function skipping configuration in targeted exploration.</td>
<td>92</td>
</tr>
<tr>
<td>5.1</td>
<td>Example showing the advantage of incremental solving.</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>An example showing the loss of context.</td>
<td>103</td>
</tr>
<tr>
<td>5.3</td>
<td>An example showing the summaries and the possible similarity between them.</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>A code fragment (simplified) in <code>cat</code> from Coreutils.</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>Searchers’ paradigms and how they affect search results and query consonance.</td>
<td>107</td>
</tr>
<tr>
<td>5.6</td>
<td>Interaction between symbolic execution and an incremental solver.</td>
<td>109</td>
</tr>
<tr>
<td>5.7</td>
<td>Configurations used in the comparison of the multiple variations of CSE.</td>
<td>112</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison of CSE with and without assumption checking.</td>
<td>113</td>
</tr>
<tr>
<td>A.1</td>
<td>Example of tree relations.</td>
<td>119</td>
</tr>
<tr>
<td>A.2</td>
<td>Example program to explain the sets of feasible transformations and states.</td>
<td>121</td>
</tr>
<tr>
<td>A.3</td>
<td>A state transition graph.</td>
<td>121</td>
</tr>
</tbody>
</table>
A.4 Example program for splitting. ................................. 125
A.5 Example of splitting. ............................................ 126
A.6 Example program for merging. ............................... 127
A.7 Example of merging. ............................................. 128
A.8 Example program for under-approximation. ............... 128
A.9 Example of under-approximation. ............................ 129
A.10 Example program for over-approximation. ................. 130
A.11 Example of over-approximation. ............................ 130

B.1 The experiment on program directedse1. ....................... 137
B.2 The experiment on program directedse2. ....................... 138
B.3 The experiment on program directedse3. ....................... 139
B.4 The experiment on program multipass. ......................... 140
List of Tables

2.1 Comparison of the two types of summarisation. 31
3.1 Summarisation strategies and their features. 53
3.2 Objects of analysis under Cirrus. 55
3.3 Experiment results on Cirrus. 57
3.4 Solver usage in the experiment on Cirrus. 58
4.1 Summary of experiment of exhaustive exploration. 85
4.2 The worst cases from the evaluation of Stratus. 87
4.3 Comparing the time to reach targets using Stratus and KLEE. 93
4.4 Comparing the time to prove target unreachability using Stratus and KLEE. 94
5.1 Summary of the experiment on CSE with incremental solving. 112
B.1 Stratus options. 136
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>BSE</td>
<td>Backward Symbolic Execution</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CNF</td>
<td>Conjunctive Normal Form</td>
</tr>
<tr>
<td>CSE</td>
<td>Compositional Symbolic Execution</td>
</tr>
<tr>
<td>CSE-I</td>
<td>CSE with Incremental solving</td>
</tr>
<tr>
<td>CSE-IA</td>
<td>CSE with Incremental solving incl. Assumption checking</td>
</tr>
<tr>
<td>CSE-NI</td>
<td>CSE with No Incremental solving</td>
</tr>
<tr>
<td>Cirrus</td>
<td>Our initial FGCSE implementation</td>
</tr>
<tr>
<td>Cirrocumulus</td>
<td>Our FGCSE tool featuring incremental solving</td>
</tr>
<tr>
<td>Concolic</td>
<td>Concrete plus symbolic, an SE technique</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth-First Search</td>
</tr>
<tr>
<td>DNF</td>
<td>Disjunctive Normal Form</td>
</tr>
<tr>
<td>DSE</td>
<td>Dynamic Symbolic Execution</td>
</tr>
<tr>
<td>EC</td>
<td>Equivalence Class, not to be confused with EC</td>
</tr>
<tr>
<td>FGCSE</td>
<td>Fine-Grained CSE</td>
</tr>
<tr>
<td>FSET</td>
<td>Faithful Symbolic Execution Tree</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ITE</td>
<td>If-Then-Else (statement)</td>
</tr>
<tr>
<td>IV</td>
<td>Induction Variable</td>
</tr>
<tr>
<td>KLEE</td>
<td>A symbolic execution tool [26]</td>
</tr>
<tr>
<td>LASUM</td>
<td>Loop-body-Acyclic SUMmarisation</td>
</tr>
<tr>
<td>LLVM</td>
<td>A compiler toolchain [87]</td>
</tr>
<tr>
<td>LSUM</td>
<td>Loop-body SUMmarisation</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>MonSE</td>
<td>Monolithic Symbolic Execution</td>
</tr>
<tr>
<td>NP</td>
<td>Nondeterministic Polynomial time (complexity)</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean SATisfiability problem</td>
</tr>
<tr>
<td>SE</td>
<td>Symbolic Execution</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>SSE</td>
<td>Static Symbolic Execution</td>
</tr>
<tr>
<td>SSUM</td>
<td>Small-step SUMmarisation</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>Stratus</td>
<td>Our Stratified SE tool</td>
</tr>
<tr>
<td>Unsat core</td>
<td>Unsatisfiable core</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>Z3</td>
<td>An SMT solver [47]</td>
</tr>
</tbody>
</table>
List of Symbols and Notations

$P$ A program
$F, f, g$ Functions; code fragments
$\text{op}$ A binary operator, assuming usual meaning
$X, Y, Z, \ldots$ Symbolic variables (corresponding to $x, y, z, \ldots$)
$c, \phi, \psi$ Logic formulae; constraints
$C$ A set of (disjoint) logic formulae; a partitioning based on it
$I$ An instruction
$R$ A sequence of instructions; a path in an execution tree
$r$ An execution, i.e., a finite list of states
$G$ A control flow graph
$\text{EntN}, \text{ExiN}$ Entry and exit nodes in a CFG
$\text{EC}(R)$ The entry condition of sequence $R$
$\text{PC}(R)$ The postcondition of sequence $R$
$\text{CP}(R)$ The condition pair of sequence $R$
$\otimes$ The concatenation operator
$\text{vars}(c)$ The set of variables in $c$
$\text{CallRatio}$ Number of summaries and number of solver calls ratio
# $\ldots$ "The number of..."
@ $\ldots$ "The memory space of..."
$\eta, \mu$ Symbolic states
$\text{Score}(\eta), \text{Score}'(\eta)$ Prioritisation scores of symbolic state $\eta$
$p$ The number of must-instructions of a symbolic state
$q$ The number of may-instructions of a symbolic state
$n, u, v, w$ Nodes (regarding an execution tree); sets of states
$(u, v)$ A pair that represents an edge from $u$ to $v$
$N$ A set of nodes (regarding an execution tree)
$E$ A set of edges (regarding an execution tree)
$T$ An execution tree
$\mathbb{T}$ A set of trees
$E \Rightarrow R \Rightarrow u, v$ A path in an execution tree; a sequence of instructions
$\text{nodes}(E), \text{nodes}(T)$ The set of nodes in $E$ or $T$
$\text{edges}(R)$ The set of edges in $R$
$\text{root}(T)$ The root of $T$
$\text{children}(n)$ The set of children of $n$ (regarding a tree)
$\text{leaves}(T)$ The set of leaves of $T$
$\text{path}(T, u)$ The path from $\text{root}(T)$ to $u$ in $T$
$\text{paths}_E(u, v)$ The set of paths from $u$ to $v$ via $E$
$E \Rightarrow u, v, \Rightarrow v$ The reachability from $u$ to $v$ via $E$ or $R$
$\subseteq$ Subset relation; tree subset relation
$\subseteq_n$ Tree subset relation with a specific subtree root
$T[n]$ The maximum subtree of $T$ with a specific root
$T \setminus T'$ Tree set minus operation
$S$ A concrete program state
$S(x)$ State used like a function on a variable $x$
$l$ A line number
$\text{pc}$ The program counter
$\text{pc}(S)$ The program counter of $S$
The set of program counters of states in \( n \)

\( \kappa \)
The domain of program states

\( \Gamma \)
A transformation function

\( \Gamma(S) \)
Transforming a state

\( \Gamma[n] \)
The set of resulting states transforming states in \( n \)

\( T \)
The predecessor of an initial state

\( \perp \)
The result of transforming a terminal state

\( r \)
An execution, i.e., a finite list of states

\( K \)
A set of states

\( K_0 \)
The set of initial states

\( K_t \)
The set of terminal states

\( K_c \)
The set of conditional states

\( K_n \)
The set of non-conditional states

\( \text{conds}(n) \)
The set of conditional states in \( n \)

\( \text{nconds}(n) \)
The set of unconditional states in \( n \)

\( \text{sens}(n) \)
The set of sensitive states in \( n \)

\( \text{insens}(n) \)
The set of insensitive states in \( n \)

\( \sqsubseteq \)
Node-wise subset relation on trees

\( \preceq \downarrow \)
Under-approximation relation on trees

\( \preceq \uparrow \)
Over-approximation relation on trees

\( c, c(S) \)
A logic formula and its evaluation based on \( S \)

\( C \)
A set of (disjoint) logic formulae; a partitioning based on it

\( C(S) \)
Partitioning a state

\( \text{ec}_i(n) \)
The \( i \)th equivalence class in \( n \)

\( T(c) \)
The under-approximation of \( T \) satisfying a precondition

\( |T| \)
The size of an execution tree

\( \epsilon \)
An executor

\( T^* \overset{n}{\doteq} T \)
\( T^* \) being the split tree of \( T \) with \( n \) split

\( T^* \overset{\text{sp}}{=\;} T \)
\( T^* \) being the split tree of \( T \)

\( T^* \overset{v}{\doteq} T \)
\( T^* \) being the merged tree of \( T \) with \( v \) merged

\( T^* \overset{\text{mg}}{=\;} T \)
\( T^* \) being the merged tree of \( T \)

\( \text{sp}(T) \)
Applying split to some nodes in \( T \)

\( \text{mg}(T) \)
Applying merge to some nodes in \( T \)

\( u\text{approx}(T) \)
Applying under-approximation on \( T \)

\( o\text{approx}(T) \)
Applying over-approximation on \( T \)
Chapter 1

Introduction

Program verification is difficult but crucial in establishing software’s reliability and security. Some 35 years ago, Brooks suggested 50% of development time is spent on testing, including component and system testing, which are crucial procedures in quality assurance [21]. Later, Hailpern and Santhanam [69] confirmed this estimate, suggesting that the cost of debugging, testing, and verification activities in commercial software development easily exceeded 50% of the total cost (up to 75%), despite 20 years of technology evolution in between these statements.

Although these are rough estimates, program verification only seems to become more prominent over time. This is because software also grows rapidly in size and complexity. It makes some traditional methods such as code reviewing, code inspection, and manual testing very expensive and yet unreliable. Integrating automation into these procedures therefore becomes increasingly important. Symbolic Execution (SE) is a useful program analysis that can discover, automatically, the relationship between program states and the program inputs that reach them. In combination with SE, a test generator can choose input data more efficiently and effectively. It also eases the subsequent debugging process by providing information such as error locations, which cannot be obtained through traditional dynamic testing.

Symbolic execution is also challenged by the complexity of programs. A real-world program is a state machine with a huge or infinite number of program states. Moreover, large programs often contain diverse and sophisticated semantics. Our research addresses the scalability problem of symbolic execution.

1.1 Symbolic Execution and Its Challenges

Symbolic execution is the execution of a program with symbolic representation of inputs instead of concrete values. Concrete executors, e.g., an interpreter, can only evaluate expressions consisting of concrete values. Symbolic executors, on the other hand, allow the existence of symbolic values in the expressions, that is, expressions of concrete values as well as names that have not been given values. By allowing inputs supplied as symbolic values, a symbolic executor can construct a relation between inputs and outputs along any execution path.

To appreciate the basic idea of symbolic execution, consider Figure 1.1. If the parameter \( x \) is fed a symbolic value \( X \) during the execution of the function \( \text{foo} \), we say we are now doing symbolic execution on this function. During the execution, the two \( \text{if} \) statements can no longer control the flow of the program, because \( X \) can have any value from the context. The trick is to construct a logical expression capturing the relation between \( \text{res} \)'s value and
Chapter 1. Introduction

```c
foo(x) {
    res := 0
    if (x < 0) { res := -1 }
    if (x > 10) { res := 1 }
    return res
}
```

**Figure 1.1**: Example illustrating symbolic execution. The language specification is given in Section 1.4. It is similar for the other examples in this thesis.

the input, such as

\[
\begin{align*}
X < 0 & \land res = -1 \quad \lor \\
X \geq 0 & \land X \leq 10 \land res = 0 \quad \lor \\
X > 10 & \land res = 1
\end{align*}
\]

This is a powerful program description that tells all that is important in an input to the program. It can be understood as a specification of path constraint and the exhibited behaviour for each feasible program path. A path constraint indicates the condition for an input, in this case \( x \), to meet so that the execution will go down a particular path. For example, the path that goes through the false branches of both `if` statements has a path constraint of \( X \geq 0 \land X \leq 10 \), and the exhibited behaviour is given by \( res = 0 \). The path constraint has an assignment \((res \mapsto 0, X \mapsto 0)\) under which the constraint evaluates to True, which makes this path feasible. On the other hand, the path constraint for the path going through the true branches of both `if` statements is infeasible because it requires \( X < 0 \land X > 10 \) which is unsatisfiable.

Typical symbolic execution handles each path separately in a symbolic state\(^1\). It essentially is a container of a path constraint and records the program behaviours. SE propagates such a state towards the end of each path, during which procedure symbolic execution identifies feasible and infeasible paths by asking a constraint solver. In our example in Figure 1.1, there are four different paths, three of which are feasible. SE finds a conflict in its path constraint as stated earlier, and it abandons this path because it does not represent a program behaviour that can actually happen. If constraint solving instead returns a satisfying value assignment for a given constraint, then the path is feasible and we can use the found values as test inputs to re-exhibit the associated program behaviour.

Constraint solving techniques are not the focus of our research. However, the application of constraint solver functionalities have significant impact on the performance of symbolic execution. We briefly introduce constraint solving in Section 2.2, and return to its application in Chapter 5.

The purpose of doing symbolic execution on programs is to verify program properties (e.g., pointer overflow detection), and/or to generate tests using the path constraints. Symbolic execution makes it possible for testing tools to achieve high path coverage with fewer tests (with the assumption that this improves the error detection rate). Symbolic execution has a great impact on modern software development. For example, SAGE [64], using Dynamic Symbolic Execution (DSE), is routinely run on Microsoft applications since 2008 and has demonstrated its irreplaceable role by finding bugs that no other method could find during the development of Windows 7 [19].

\(^1\)More generally, symbolic execution can choose to partition the input domain in an arbitrary way, e.g., putting more than one path in a state.
The practical cost and yield of SE depend on the path-wise complexity of the target program, and the frequency of appearance of hard-to-reason-about instructions (e.g., nonlinear operations, memory operations, external calls). In particular, the number of paths of a program can be very large or infinite even for a small program, when there are input-dependent loops or recursive calls. This is the problem referred to as path explosion, which describes a situation where the number of paths grows so rapidly (exponentially to the number of conditionals) that it is like an explosion. Moreover, sophisticated programs incorporate multiple components, modules, and even machines, each of which puts different requirements on a symbolic execution tool and its underlying solver. In other words, symbolic execution often encounters code beyond its capability of reasoning, or code that is unavailable during testing. The effort to address one of the above problems is in conflict with the other. For example, to verify a program accurately we need to symbolically execute the libraries it uses, which increases the cost; ignoring the library improves the scalability, but introduces inaccuracy. These are the main obstacles for symbolic execution to have a wider range of application.

1.1.1 Existing Solutions

To tackle these problems, modern symbolic execution tools adopt various kinds of optimisations and approximations. Optimisations include but are not limited to, the use of intelligent searchers that prioritise the paths that produce the most useful test cases or have the most interesting behaviours, the integration of other program verification techniques (e.g., using static analysis to find loop invariants), and the use of query optimisation techniques to speed up the solving process. We introduce them in Section 2.3.2 as faithful optimisations because they are, in principle, loyal to the set of program behaviours have can actually happen.

Our research is based on the various approximation techniques used in symbolic execution. Compared to the above, approximations bring inaccuracy into symbolic execution. They either consider a subset or a superset of the original set of program behaviours. However, with careful deployment, they greatly improve the efficiency and the ability to handle hard-to-reason-about code, while guaranteeing sufficiently meaningful results. Notably, some techniques are widely used (all of these as well as other techniques with approximation are explained in detail in Chapter 2):

Concretisation This includes concretising symbolic variables or using concrete execution directly, both trying to simplify a problem. For example, dereferencing symbolic pointers is considerably harder than other kinds of operations involving symbolic variables. Some approaches will concretise those pointers in such cases to reduce the complexity. This is an approximation because the problem is not equivalent to the original problem after concretising.

Pruning and specialising Depending on the objectives of the applications, symbolic execution can consider some parts or paths in the program redundant or less important. For example, some analysis focuses on line coverage, and new paths that do not produce additional line coverage can be considered redundant and will not be explored further; some approaches guess the program parts that are most likely to have errors using heuristics and explore them exclusively. This constitutes a kind of approximation as well.
Summarising Some tools are able to run symbolic execution on program parts instead of the whole program. Its main advantage is that it can reach and explore specific functions very quickly, but the drawback is that it may raise false alarms. Furthermore, such techniques can be combined with normal symbolic execution, while reusing the SE results on program parts as summaries to reduce the cost of repeatedly visiting them. This is called (bottom-up) Compositional Symbolic Execution (CSE) [58].

1.1.2 Our Observation

Let us consider the search space of SE. The search space of SE consists of all the program states that symbolic execution needs to verify. The concrete execution of a program defines the concrete search space, which is the set of states that can appear during this process. We call them feasible states, and we call the states that do not appear in any cases infeasible states. Compared to faithful optimisations, approximation techniques try to alter the search space of SE. Specifically, under-approximation tries to create and search in a subset of the concrete search space, whereas over-approximation tries to create and search in a superset of it.

Figure 1.2a is an illustration of techniques that are based on approximation and how it affects their search space. “Concrete” represents the possibilities in concrete execution. What is interesting is how different approaches decide the boundary of feasibility. Under-approximation assumes too strong conditions on states and it misses some feasible paths; over-approximation assumes too weakly, and falsely considers some infeasible paths feasible. It might realise these mistakes after collecting more information down the paths, or it might continue executing based on wrong assumptions. This incurs more cost. We list a simple but concrete example in Figure 1.2b. Here, there are three assumes representing the assumptions in different techniques. Again, different techniques have different assumptions due to scalability issues or the different orders in collecting constraints.

Let us return to the techniques we mentioned earlier. Concretisation and pruning techniques belong to under-approximation, because they eliminate some feasible program states. Symbolic execution that runs on program parts accepts infeasible states due to lack of calling context, therefore it belongs to over-approximation. Techniques such as context-insensitive summarising might accept infeasible states temporarily for the same reason, but will eventually reject those states. We classify them as over-approximation as well. Additionally, there are hybrid approaches or idealisations, e.g., interpreting a machine integer as an ideal integer. They accept states that cannot happen in a real machine, while also missing states that only happen in a real machine, e.g., integer overflows. This makes them neither strict under-approximation nor strict over-approximation.

Definition 1 (Over-approximation and under-approximation) In this thesis, we define over-approximation during SE as the process of accepting or delaying the rejection of infeasible states; we define under-approximation during SE as the process of eliminating feasible states. An under-approximation approach is one that possibly under-approximates but never over-approximates, and vice versa.

We observe that the approximation techniques introduced into symbolic execution for its scalability are mostly under-approximation. Symbolic execution tools usually avoid an approach that has over-approximation because of the said cost.
1.1. Symbolic Execution and Its Challenges

(A) The search space of symbolic execution and a comparison of techniques with approximation.

```plaintext
x := read
// assume x > -10 // over-approximation
assume x > 0 // concrete/context
// assume x = 5 // under-approximation
if (x > 0) {
  if (x < 10)
    f1(x)
  else
    f2(x) // missed by under-approximation
} else {
  f3(x) // unreachable but reached by over-approximation
}
```

(B) A comparison using a concrete example.

Another reason is that, while under-approximation and over-approximation both bring inaccuracy into symbolic execution, under most circumstances, symbolic execution is a problem of fault detection where it searches for targets, as opposed to the problem of proving where a proof system tries to eliminate the possibility of the existence of targets. The complexity of modern software often makes the search incomplete, thus rules out a proof system using pure symbolic execution. Since a proof is practically unachievable, the value of symbolic execution lies mainly in its fault detection ability. In this respect, under-approximation is preferred over over-approximation which suffers from false positives. In this thesis, false positive assumes an error-detection-like problem, i.e., it means to report an error that does not exist. Conversely, false negative means to miss real errors. They correspond to unsoundness and incompleteness respectively, defined in the same way as related work [59].

However, we can justify over-approximation by arguing that

- Although there is additional cost in handling infeasible states, it can possibly be compensated for by handling feasible and infeasible states together. SE essentially considers a set of possibilities collectively. Arranging them together might achieve shorter or cleaner constraints.

- Accepting infeasible states makes an unsound analysis, but delaying the rejection of them still makes an accurate analysis. We might benefit from temporarily accepting infeasible states (lazily rejecting them).

We are also motivated by many concrete examples and successful existing applications of over-approximation, which shall be introduced in this thesis gradually. We find that there is
Chapter 1. Introduction

```c
complex(x) { /* Expensive for SE */ }
```

```c
f(x) {
  if (x > 0) {
    complex(x)
  }
  if (x < 0)
    error()
}
```

**Figure 1.3:** From the most general over-approximation of the call to `complex` we can still conclude the falsehood of the reachability of the call to `error`.

still some distance to fully realise its potentials which demands more research on this class of techniques.

**A Motivating Example**

Let us see an example where over-approximation can be more efficient. For the symbolic execution of `f` in Figure 1.3, suppose `complex` is a complicated function with input-dependent loops and conditionals but which does not modify the variable `x` (here, it is passed by value). We can see right away that the call to `error` is unreachable. However, traditional symbolic execution will need to explore every path in the program, including those inside `complex`, before it can draw the same conclusion.

By using the most general over-approximation—assuming the call to `complex` can do anything a function with such declaration can do, we can very quickly prove that the call to `error` is unreachable without reasoning about `complex` at all. Why is such an assumption an over-approximation? That is because we now consider all of the possibilities allowed by such a function declaration. Since we know none of those functions can modify `x`, we can get our proof.

On the contrary, under-approximation techniques, e.g., concretising the call to `complex`, can also pass the function very fast, but they cannot obtain a proof. There are existing proof methods which incorporate over-approximation, e.g., using uninterpreted functions [1], to achieve a similar proof. Unlike that approach, in Chapter 4, we show a method that allows over-approximating a more general kind of function and “recovering” from the over-approximation, which further allows us to iteratively prove programs.

Figure 1.3 only shows one simple way of using over-approximation in symbolic execution. For example, when the interface of `complex` shows that it can modify `x`, this over-approximation based on analysing the interfaces no longer works. We show how to solve this more general problem in Chapter 4. As a high-level summary, we think the advantages of over-approximation lie in

- Producing context-insensitive, reusable information to reduce repeated work.
- Producing abstractions to obtain program proofs.

**1.1.3 Problem Definition**

In summary, our research is mainly to tackle symbolic execution’s scalability problems using techniques based on over-approximation. We ask the questions: “How, when, and to what extent can the various methods of applying over-approximation improve the scalability of SE?”
1.2 Contributions

By categorising symbolic execution approaches based on the approximation techniques they use, we notice an area that has not been investigated much. In this thesis, we explore different strategies of using over-approximation to address the difficulties in the scalability of symbolic execution.

Our contributions are summarised below:

- We have generalised CSE to using Fine-Grained summarisation (FGCSE) that works on any code fragments, which was envisioned but not implemented in the existing CSE approaches (e.g., SMART [58]). We have implemented an experimental fine-grained CSE tool Cirrus and show that fine-grained summarisation can improve SE’s efficiency. Since it works on a different (finer) granularity, it can be used in combination with existing function-level summarisation. As an over-approximation technique, summarisation needs good target selection strategies for the summarisation module, especially in fine-grained summarisation, to avoid too many summaries. We improved CSE by designing three strategies and compare their performance, with the best of them achieved 20% time reduction of SE in total. Additionally, our implementation is capable of using backward reasoning which we consider another novel combination.

- We introduce Stratified Symbolic Execution (Stratified SE), which allows skipping and resolving of function calls in symbolic execution. Stratified symbolic execution over-approximates during the skipping process, which for a state means to jump over a call without collecting any constraints such as what we could do with function complex in Figure 1.3. The resolving process for a state, on the other hand, is to jump back to the call site that was skipped and resume normal symbolic execution in the function being called, recovering from the over-approximation. We show that Stratified SE better differentiates the code and provides better search space configuration. We show that it can alleviate the path explosion problem in some cases by temporarily omitting low-level code and obtaining program information in a different order, and this becomes particularly beneficial when low-level code is well-identified. Traditionally, because of the complexity of SE, it is hard for SE to prove properties such as unreachability or the absence of certain types of faults. As such, SE is typically reserved for generating test inputs or finding specific types of faults. We show that with these properties, Stratified SE is good at applications such as targeted SE and partial/iterative program proving. Stratified SE is implemented in a tool called Stratus on top of KLEE. In particular, on average, Stratus can reach targets using 15% less time, or prove the unreachability of targets using 24% less time, while timing out less often.

- We show that incremental solving is a way to suppress the negative effects of over-approximation. We combined the CSE with modern solver features, in particular, incremental solving and unsat core computation. We show that the combination is beneficial in that the solver technique promises great performance gains and provides a solution to the weakness of CSE (i.e., too many summaries), and that CSE is intuitively a better host for incremental solving than other types of SE because it is less dependent on search heuristics which incremental solving is not compatible with. We integrated incremental solving into a more mature implementation of FGCSE called Cirrocumulus, which is based on the LLVM infrastructure. Cirrocumulus is ≈1.8 times as fast as FGCSE without incremental solving.
Chapter 1. Introduction

For the calling context/general context: FGCSE in Chapter 3, 5

For the calling details: Stratified SE in Chapter 4

To suppress over-approximation: using assumptions in Chapter 5

An abstraction of transformations: using execution trees in Appendix A

(A) A high-level summary of the different aspects of the over-approximation process in symbolic execution. We can map one of each chapter’s key messages to one of the different aspects of this process, so that we can see the relationship between the chapters.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Order of implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cirrus</td>
<td></td>
</tr>
<tr>
<td>Cirrocumulus</td>
<td></td>
</tr>
<tr>
<td>Stratus</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target of over-approximation</th>
<th>Cirro</th>
<th>Cirrocumulus</th>
<th>Stratus</th>
</tr>
</thead>
<tbody>
<tr>
<td>The calling context or the general context</td>
<td>The calling details</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target language</th>
<th>Tool implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A toy language</td>
<td>Toolbox Cirrus Cirrocumulus Stratus</td>
</tr>
<tr>
<td>LLVM</td>
<td>LLVM (KLEE)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technique</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGCSE</td>
<td>Chapter 3</td>
</tr>
<tr>
<td>Combining FGCSE with incremental solving</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>Stratified SE</td>
<td>Chapter 4</td>
</tr>
</tbody>
</table>

(Figure 1.4) A high-level summary of the thesis and the tool implementations.

Our tool implementation is centred around the idea of over-approximation. Figure 1.4b summarises the ideas behind the tool. The names come from different kinds of clouds.

1.3 Thesis Outline

Chapter 2 covers related work with emphasis on the different solutions to the path explosion problem in SE. It also includes a brief introduction to related concepts such as constraint solving.

Chapter 3 covers our fine-grained summarisation techniques as well as strategies for summarisation. Chapter 4 introduces the stratified symbolic execution techniques: the use of skipping and resolving in symbolic execution and its specialised searcher; the application of it in targeted symbolic execution; the application of it in program proving. Chapter 5 is about the combination of incremental solving and compositional symbolic execution.

In Appendix A, we introduce our conceptual framework that defines the terms and notations that are helpful to the other chapters. These definitions are familiar to readers from similar background. The main chapters may use certain notations and definitions defined in Appendix A to aid the discussion, although they are understandable without Appendix A. Some specific sections that directly depend on them are marked with a symbol '*' in the titles.

Figure 1.4a actually characterises the over-approximation techniques discussed in this thesis. We can map one of each chapter’s key messages to one of the different aspects of this
1.4. Subject Languages Used Throughout the Thesis

We define a simple language that is used by all of our examples in the thesis. This is adapted from the subject language of Cirrus, our FGCSE prototype (see Figure 1.4b). Later implementations (Cirrocumulus and Stratus), however, use LLVM IR (Intermediate Representation) for the purpose of more thorough evaluation. We unify the presentation of examples and related discussions in this thesis through the use of a language which is conceptually low-level enough to approximate the actual languages used, while omitting many details so that they convey more general messages and appears clean and friendly to, e.g., a C user. This is a reason why we sometimes refer to a statement as an instruction where we mean a similar thing. We call this language Ozpin, whose syntax is shown in Figure 1.5.

For the above reason, Ozpin has only some essential syntax that is required to encode meaningful commands. We also omit some derivable or syntactically similar operators, e.g., unary, bitwise, and logical operators, as long as they are well-understood. We assume usual semantics for the common operations.

A program \( P \) is a list of functions. Each function \( F \) has a header with function name \( fn \) and a list of parameters \( params \), which is a comma-separated list of variable names. An asterisk * is a decorator which indicates that the variable is used as a pointer. We usually omit the entry function’s header in the examples.

---

\[ P \rightarrow F \mid \epsilon \]
\[ F \rightarrow \text{dec fn ( } \mid \text{dec fn ( params ) stmts} \]
\[ \text{params} \rightarrow \text{dec var} | \text{dec var}, \text{params} \]
\[ \text{dec} \rightarrow \star \mid \epsilon \]
\[ \text{stmts} \rightarrow \text{stmt stmts} | \{ \text{stmts} \} | \epsilon \]
\[ \text{stmt} \rightarrow \text{var := read} | \text{var := expr} | \text{assume expr} \]
\[ \text{fn ( ) } | \text{fn ( args ) } | \text{return expr} \]
\[ \text{var := load var} | \text{store var expr} \]
\[ \text{if ( expr } \mid \text{if ( expr } \mid \text{else stmts } | \text{while ( expr } \mid \text{stmts} \]
\[ \text{expr} \rightarrow \text{const} | \text{var} | \text{expr op expr} | \text{fn ( ) } | \text{fn ( args ) } | \ldots \]
\[ \text{op} \rightarrow + | - | * | / | \ldots \]
\[ \rightarrow | < | > | \ldots | \text{and} | \text{or} | \ldots \]
\[ \text{args} \rightarrow \text{expr} | \text{expr}, \text{args} \]

Figure 1.5: Ozpin syntax. \( \text{var} \), \( \text{const} \), and \( \text{fn} \) are the syntactic categories of variable, constant, and function names respectively, which we do not expand.

process. Chapter 3 and Chapter 4 are chapters about two different techniques, one with over-approximation on the calling context (in FGCSE, the calling context can be the context of any code fragment), the other one with over-approximation on the calling details; Chapter 5 is about how to suppress the infeasible summaries from over-approximation; Appendix A is abstracting this process using a tree representation.

We conclude the thesis and discuss future work in Chapter 6.

1.4 Subject Languages Used Throughout the Thesis

We define a simple language that is used by all of our examples in the thesis. This is adapted from the subject language of Cirrus, our FGCSE prototype (see Figure 1.4b). Later implementations (Cirrocumulus and Stratus), however, use LLVM IR (Intermediate Representation) for the purpose of more thorough evaluation. We unify the presentation of examples and related discussions in this thesis through the use of a language which is conceptually low-level enough to approximate the actual languages used, while omitting many details so that they convey more general messages and appears clean and friendly to, e.g., a C user. This is a reason why we sometimes refer to a statement as an instruction where we mean a similar thing. We call this language Ozpin, whose syntax is shown in Figure 1.5.

For the above reason, Ozpin has only some essential syntax that is required to encode meaningful commands. We also omit some derivable or syntactically similar operators, e.g., unary, bitwise, and logical operators, as long as they are well-understood. We assume usual semantics for the common operations.

A program \( P \) is a list of functions. Each function \( F \) has a header with function name \( fn \) and a list of parameters \( params \), which is a comma-separated list of variable names. An asterisk * is a decorator which indicates that the variable is used as a pointer. We usually omit the entry function’s header in the examples.

---

\(^2\)This is except for the instances in which we are discussing real-world programs, e.g., from the evaluation, in which case we use their direct source code instead.

\(^3\)Note that though our SE tools, like many other SE tools such as KLEE, directly deal with such low-level languages, we can use front-end compilers to transform C or C-like languages to them.
Function $F$ also has a list of statements denoted by $stmts$, consisting of zero to many statements denoted by $stmt$. A statement is a simple assignment, a memory operation, a control statement, a function invocation, or a return.

An assignment is either assigning an input value (through $\text{read}$) to variable $\text{var}$, assigning the evaluation result of an expression $\text{expr}$ to a variable $\text{var}$, or an $\text{assume}$. $\text{read}$ is used by programs take to inputs. In examples concerning symbolic execution, $\text{read}$ is where a symbolic variable is introduced. An $\text{assume}$ is equivalent to a conditional loop where the condition is $\neg \text{expr}$ and the loop body is empty. When an $\text{assume}$ is encountered we basically assume some condition for the rest of the execution (we do not consider the loop itself).

Statements include two kinds of memory operations. The memory they work on is simply a mapping from pointer values (addresses) to values. A load operation will “dereference” a pointer variable—load a byte from the cell mapped to by the pointer variable to another variable. It is easy to extend its function to loading multiple bytes, but we do not need them in our examples so we omit this for simplicity. A store operation will update the memory with every cell unchanged except that the cell mapped to by the specified pointer variable $\text{var}$ will have a new value equal to the specified expression $\text{expr}$.

A control statement is either a branching statement or a loop, guarded by an expression $\text{expr}$ as the condition. Depending on whether the expression evaluates to 0 or not, the control statement directs control to different statements. An expression is either a constant, a variable, the result of an operation, or the return value of a call with a list of expressions as arguments. The operators have standard mathematical meanings which we need not repeat.
Chapter 2

Background

This chapter introduces the background of the thesis. We first briefly summarise the necessary background to understanding applications of symbolic execution as it relates to this thesis. Then we revisit the basic concepts of symbolic execution we have seen in Section 1.1, i.e., the basic work pattern and the major challenges. This is followed by a classification of SE applications based on how thorough the exploration is expected to be. Finally, we investigate the existing SE approaches with or without approximation, and in particular, we introduce how they handle the aforementioned challenges.

2.1 Program Verification, Testing and Fuzzing

Program verification, testing, and fuzzing, which is a specific kind of testing, are the general background of symbolic execution. Symbolic execution is used as an automation method of such processes under the white-box assumption, i.e., the source code is assumed known to the verifier or tester.

2.1.1 Program Verification

Verification is to formally prove the correctness of a program. The correctness is judged by whether the program satisfies desirable or stated program properties, such as, being free of storage violation, avoiding overflow, or satisfying given assertions.

Verification overlaps with testing. Testing is a method to find a property violation and produce a witness, which is very helpful in verification, while it can also be used to validate requirements. We use other techniques in conjunction with testing to verify a program, such as abstract interpretation and points-to analysis.

Verification is one aspect of the V&V activities in software development. The other one is validation, which is the process of assuring a software meets the user’s requirements.

2.1.2 Program Testing

Testing is a method to detect bugs or violations of requirements in a program. Unlike program verification, it never proves a program correct. As Dijkstra [48] puts it,

> Program testing can be used to show the presence of bugs, but never to show their absence!

It does not remove the defects found either, which is for the subsequent debugging process. In short, testing warns and reports against bugs.
Testing of a program generally has the following steps. First, we select test cases. This can be achieved by manually analysing the code and systematically choosing them, e.g., equivalence partitioning of input domains and data-flow analysis, or using automated techniques such as fuzzing (see below) and symbolic execution. A set of test cases systematically selected for a program is called a test suite. Then we execute the test cases, to expose errors in the program. Finally, we evaluate the testing results, i.e., checking the outputs against our expectation. Testing also includes code reviewing and inspection, and the evaluation of test suites (we discuss test coverage below), which facilitate the process and achieve better effectiveness. A System Under Test is abbreviated by SUT.

Testing is used in complement with many other techniques during V&V. Compared with static analysis, it avoids false positives as it conducts real execution on programs.

Depending on the automation degree, testing can be manual or automated. The automation of test execution and part of the evaluation process are easy. For the latter we mean automatically checking a program output against a given requirement. However, the automation of test selection and the generation of those requirements are hard. SE plays an important role in test selection.

Finding flaws in large and complex systems manually is often onerous if not impossible. It is also limited by the skills of the person in regard, which are less reliable than a machine in terms of thoroughness. Automation is then desirable. It is superior to manual testing in simplicity, objectivity, and hence the overall effectiveness. However, compared with manual testing, automatic testing has limitations: it is more suitable for generic bug detection, such as program crashes or exceptions, than validating requirements, such as checking the test outputs against the expectation; in particular, symbolic execution can be very expensive for finding deep bugs; symbolic execution relies on a constraint solver to select test values, which makes its individual test cases (assuming we have expected outputs for them) less good at detecting errors like boundary shifts. Although there are efforts towards the support of these [79], we should combine manual and automatic testing to maximise test outcomes [134].

Testing approaches can be divided into three groups depending on the availability of an SUT during the process. They are Black-box Testing, White-box Testing, and Grey-box Testing. Black-box testing assumes that the internals of the system are not available to the tester, only the specifications such as the interface; white-box testing assumes that it is available, including the program’s internal structure and source code; grey-box testing is in the middle, assuming partial or abstract information such as the algorithms used.

Some specific and notable testing problems and techniques are listed below:

**Unit and system testing** A large software system can be considered as an integration of program components, or units. Unit testing is designed for testing these components separately before system testing, which is to test the system as an entity. This approach is more efficient than applying only system testing, a process which is more dynamic (non-deterministic), more expensive, and at which stage debugging is harder.

**Test coverage** The coverage of a test suite measures the test suite’s quality. There are many coverage criteria, such as statement coverage, decision (or branch) coverage, and path coverage, which are the coverage criteria that are relevant to this thesis:
1. Statement coverage is measured by the number of statements covered by a test suite.

2. Decision or branch coverage is measured by the number of branches covered (e.g., an if statement as two branches).

3. Path coverage is measured by the number of path covered.

Statement coverage is subsumed by decision coverage, which is subsumed by path coverage. A criteria subsumes another, means that when the former is met, the latter is met too. Inozemtseva and Holmes [75] showed that higher coverage indicates better thoroughness of a test suite, and hence better effectiveness in error detection. However, it also showed that the higher coverage achieved by increasing the size of the test suite is better correlated with the effectiveness than when it is achieved by increasing each individual test’s coverage. In other words, when the size is fixed, the correlation between coverage and effectiveness drops. Also, different coverage criteria have no significant difference in their capabilities of evaluating the quality of a test suite [75].

**Testability**  The *testability* of a system is a matter of, during testing, how well it can be controlled—how well the inputs can be controlled and fed to the system, and observed—how well the outputs can be assessed. Good developers are expected to produce as testable programs as possible (for some applications the programs are inherently less testable, such as mobile applications that need to handle random events). Testability is also important in symbolic execution or other automated verification processes, where it is additionally a matter of how fast verification can be performed on a system, and how precisely the verification results reflect the system’s correctness. In other words, the efficiency and precision of verification not only depend on the verifier but also the manner in which the target system is constructed. Program transformation and optimisation can change the testability of a program. Harman et al. [72] investigated in *testability transformation* and showed that transformation does not have to preserve the program’s meaning to improve the testability of it. Dong et al. [50] and Cadar [25] studied how the optimisations performed on programs affect their testability in SE. Wagner et al. [132] proposed an specific optimisation for programs to be quickly verified. If we consider testability optimisation as part of symbolic execution, then we can classify it under the category of neither under-approximation nor over-approximation, because some optimisations do not preserve the original program’s meaning as stated above. A systematic classification of SE presented later.

**Regression testing**  Regression testing facilitates the process of testing evolving systems. It can generate test cases that specifically covers code that is affected by a modification or a patch of a program, based on the assumption that the other parts are well-tested by previous testing activities on the older version of the same program. Three major problems under research regarding regression testing is test suite minimisation, test case selection, and test case prioritisation, which are covered in a recent survey by Yoo and Harman [139]. Test suite minimisation is the problem of finding a minimal subset of a test suite such that the test requirements are still satisfied; test case selection is the problem of, also, how to identify a subset of a test suite for a program to test the modified version of the same program; test case prioritisation is the problem of ordering a test suite such that executing its test cases reveals more benefits at an early stage, e.g., more fault detection.
2.1.3 Fuzzing

Fuzz testing or Fuzzing is a software testing technique, where inputs are automatically generated to try the target system, in order to reveal vulnerabilities. As Takanen et al. defined [123], fuzzing is

A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from the modems applications’ tendency to fail due to random input caused by line noise on “fuzzy” telephone lines.

In fact, “fuzz” was taken as the tool’s name authored by the people who discovered this phenomenon [95]. Originally the tool can generate a lot of random inputs based on some heuristics, e.g., it can include both printable and control characters in an input, or put delays between inputs, to crash UNIX utilities. It has since led to a lot of research and the name becomes the label of the tools of this kind.

Fuzzing is automated yet lightweight, compared to symbolic execution. Fuzzing does not assume the availability of source code. In other words, it is able to do black-box testing, but its heuristics might still let it benefit from the source code if given [60, 56, 103]. Tests are randomly generated (under some rules given by the heuristics), so it does not depend on theorem provers, whose invocation is expensive (see Section 2.2). Fuzzing’s disadvantages are its inability to check function correctness and that it could produce too many tests:

- Because test inputs are randomly generated, usually in large numbers, it is hard to produce test oracles accordingly, therefore it is more suitable for crashing programs (finding generic bugs) rather than testing for functional correctness. This is true for symbolic execution as well.
- The test suite is less refined than that of symbolic execution. For example, many tests execute the same path, thus not revealing different program behaviours, or some paths are missed by all tests (which can happen in SE as well when it is incomplete). This also makes managing the test suite harder.

Fuzzing and symbolic execution can work together and the combination outperforms the individual techniques [120]. More about fuzzing techniques can be found in a survey by McNally et al. [94].

2.2 Constraint Solving and Solvers

A constraint is a set of logic formulae, representing a set of requirements or conditions. A constraint is said with respect to a certain theory or a fragment of a theory which define a set of formulae in question. We usually talk about first-order theories. For example, the theory of equality consists of only one nonlogical symbol (=) as the connectives between variables; the theory of integers allows linear arithmetic operations on integers; the theory of arrays allows array reads and writes. A fragment of a theory tries to restrict the discussion to a specific subset of formulae in a theory. The restrictions are with respect to the logical connectives used in the formulae. For example, we usually talk about the quantifier-free fragment of a theory, which excludes the use of quantifiers (\(\exists, \forall\)) in this theory.

\[^{1}\text{White-box fuzzing is a term sometimes used by symbolic-execution-based fuzzers [64, 60, 80, 103].}\]
In symbolic execution the required theories change depending on the programs we analyse. For example, reasoning about assignments requires only the theory of equality; to reason about addition operations and comparisons we may invoke a theory of linear arithmetic; reasoning about arrays and pointers we additionally need a theory of arrays. In SE, constraints are usually quantifier-free, with the exception of, e.g., comparing two arrays. In general, we need a reasoning method for a combination of (fragments of) first-order theories to do SE.

The constraint solving problem is to find satisfying assignments for the variables in given constraints. Depending on the problem types, we use different solving algorithms.

### 2.2.1 Constraint Solving Problems and Algorithms

Constraint solving algorithms have been used in mathematics for centuries, witness Gaussian elimination [57] for linear equations, Fourier-Motzkin elimination [54] for linear inequalities, and many other methods that lay the foundation of constraint solving. After the stored-program computer was invented, it was soon put to use in constraint solving. Dantzig’s simplex algorithm that can be used for linear programming emerged around the same time [42].

Later, constraint solving methods have been developed for many different theories, usually employing a combination of search and propagation. For a simple example of the idea behind constraint propagation, assume we know that $x \in [1, 6]$, that is, $x$ is an integer (say), in the interval $[1, 6]$ and $y \in [0, 3]$. If we additionally know that $x < y$, we can cut down the possible space of solutions by letting the $x < y$ constraint trigger the bounds propagation. The propagator rule is

\[
\begin{align*}
\text{LowerBound}(y) & \leftarrow \text{LowerBound}(x) + 1 \\
\text{UpperBound}(x) & \leftarrow \text{UpperBound}(y) - 1
\end{align*}
\]

So we can tighten the bounds information to $x \in [1, 2]$, $y \in [2, 3]$. Propagation is not complete, in general, as a solver procedure; that is, search remains a critical solver ingredient, but propagation limits search.

An early interest was the Boolean Satisfiability problem (SAT), or the satisfiability problem of propositional formulae, for which SAT solvers were developed. There are many SAT solving algorithms, but they can be generally classified as DPLL-based (DPLL is short for Davis-Putnam-Loveland-Logemann [45, 44]) or stochastic search. DPLL follows the search and propagation idea we sketched above, with propagation being unit propagation, that is, propagation of single bit values, as soon as they have been discovered/fixed.

Although SAT focuses on propositional logic, it is useful because formulae in many other theories can be transformed to propositional formulae by bit-blasting. Bit-blasting breaks down variables of larger domains such as an integer as Booleans, each representing a bit of it. This is especially true for constraints generated for the purpose of program analysis since they are essentially reasoning about fixed-size bit-vectors. However, a problem of bit-blasting is that it potentially creates an astronomically large formula even for a small constraint, e.g., with a multiplication of two variables. Bit-blasting also easily loses information that would be useful in word-level propagation.

Satisfiability Modulo Theories (SMT) is a natural extension of SAT, which aims to solve formulae of a combination of theories. It can apply simplifications and word-level solving
in accordance to the theory in question, such as Gaussian elimination before invoking bit-blasting and a SAT solver. This makes it more powerful and more applicable in automatic theorem proving, code verification, model checking, scheduling [46], etc. SMT-LIB [12] is a well-accepted standard for SMT solvers, specifying the background theories for the solvers and introducing a language to communicate with them.

Applications of SMT solvers in verification Constraint programming is the major motivation to the evolution of constraint solvers. It is a programming paradigm using constraints and recurring mechanism to state and solve problems. Meanwhile constraint solvers play an important role in program verification, where in many cases the tasks can be reduced to problems for which decision procedures exist. So do they in automated program testing, where we deal with typically path constraints, possibly huge in number or length. An SE tool has an underlying SMT solver. In fact, solving path constraints takes up most of the symbolic execution time [19]. STP [55] is the constraint solver deployed in KLEE [26]. Z3 [47] is a comprehensive SMT solver developed and maintained by Microsoft Research. SAGE’s [64] computational usage of Z3 possibly represents the largest usage of any SMT solver (as suggested by Z3’s authors [63]). Another comprehensive solver is CVC4 [13]. Some SE tools support multiple solvers or a portfolio solver [98].

Complexity SAT is NP-complete\(^2\). SMT, depending on the theories in consideration, can be tractable, intractable, or even undecidable. In practice, many verification problems belong to some fragments of these theories in discussion, e.g., they are quantifier-free, therefore they may be decidable and even solved efficiently. An exception worth mentioning is the theory of arrays, which can be used to model computer storage. Formulae of this kind have quantifiers when we need to compare whole arrays, making them undecidable (which is why write concretisation is used, as discussed in Section 2.3.3). In many cases SE queries from a single state are also disjunction-free\(^3\), which makes them easier to solve, but they still stress the solver because of the path explosion problem. This means the methods used for the exploration during SE can be as prominent in deciding an approach’s efficiency as the methods used in solving individual constraints. In certain cases, the number of constraints complicates the problem more than the hardness [19]\(^4\). Therefore while SE’s performance depends on its solver, it also depends on how SE itself uses the solver. Part of this thesis tries to approximate what different approaches do to minimise the use of a solver. A formal language featuring the use of execution trees is introduced in Appendix A.

2.2.2 Incremental Solving

Incremental solving [77] is a particular kind of optimisation in constraint solvers. It lets solvers solve additional constraints quickly, based on the lemmas built from previously solved constraints. For example, when solving two constraints \(c_1 \land \ldots \land c_k\) and \(c_1 \land \ldots \land c_k \land c_{k+1}\), incremental solving is likely faster than using traditional solving, because they have

\(^2\)We assume formulae are in Conjunctive Normal Form (CNF). For Disjunctive Normal Form (DNF) the problem can be solved in linear time. A SAT formula can be transformed to an equisatisfiable formula in CNF in linear time.

\(^3\)Disjunctions are introduced by state merging, summarisation that has implicit merging, bounds checking, etc. In low-level program representation, logical ors and ands are often translated into control flow splits and handled by multiple states.

\(^4\)Different SE architecture utilise solvers differently, e.g., this might be less significant in SE tools that do not distinguish independent constraint components.
some primitive constraints in common and incremental solving can benefit from knowledge gained while solving the common parts.

Incremental solving is a superset of independence checking and solution caching (see query optimisations in Section 2.3.2), because it can still accelerate the solving of constraints that share common variables. This is explained in detail in Section 5.1.3. Note that, in this thesis, we refer to the incremental solving functionality that can be enabled and disabled in a solver as incremental solving; we do not use this term to refer to the query optimisation techniques that are applied prior to the invocation of the solver, although they do provide an extent of incrementality.

Incremental solving requires an indication of what to remember—an indication of repetitive constraints. It also requires similarity between queries, in order to reuse the knowledge from the historical solving activities. For example, consecutively solving two constraints $c_1, c_2$ which do not share any variable will not benefit from incremental solving, because there is nothing to reuse.

In Chapter 5 we discuss the significance of incremental solving in symbolic execution. Existing symbolic execution approaches do not actively deploy incremental solving. Sen et al. implemented their incremental solver and plugged into their concolic execution tool [115]. Liu et al. [89] studied the three types of incremental solving approaches/substitutions and found that no work has been using the incremental constraint solving feature provided by an off-the-shelf solver. Again, we discuss the reason behind this in combination with our review of incremental solving in SE.

**Unsatisfiable cores** An unsatisfiable core (or unsat core) provides a generalised reason of why a constraint is not satisfiable. For example, if $c_1 \land c_2 \land c_3$ is unsatisfiable and we discover that $\{c_1, c_2\}$ is already a conflict without considering $c_3$, then we can say $\{c_1, c_2\}$ is a reason (one of the reasons) why $c_1 \land c_2 \land c_3$ fails. Computing unsat cores is very helpful for speeding up SE. We can see an example in Section 2.3.2 regarding interpolation. A problem in this area is how to compute the minimal unsat cores. A minimal unsat core is an unsat core that when removing any primitive constraint from it, it becomes satisfiable. We are interested in minimal unsat cores because they increase the space of known infeasibility, which is useful for, e.g., path pruning in SE. We compute unsat cores from assumption checking. This is introduced together with our application of incremental solving in SE in Chapter 5.

### 2.3 Symbolic Execution

Program verification and testing are important, yet expensive, activities in software engineering. A significant portion of software development time is spent on these processes [21, 69] to ensure software quality. However, the traditional manual ways of verification and testing are labour-intensive and yet unreliable, because human developers have limited powers and make mistakes, especially when doing repetitive work. Program verification and testing need automation.

King [84] proposed to use symbolic execution as an enhanced testing technique in 1976. It is achieved by using “symbolised” inputs to execute a program. Such a symbolic input represents a class of concrete inputs. Like concrete execution, symbolic inputs participate in the evaluation of expressions, but since a symbolic input does not have a definite value, this creates symbolic expressions that are carried down every feasible execution path. In the
end, symbolic execution constructs a relation between the inputs and the outputs for each feasible path. It can find input-output relations and in many cases determine what it takes to produce a particular output. We have seen an example in Figure 1.1. We refer to the class of SE techniques based on this idea as Static Symbolic Execution (SSE), as opposed to a dynamic variation of it (called DSE) that interleaves concrete execution in SSE, which we introduce in Section 2.3.3.

Under the white-box assumption, a program’s source code is (partially) available to the verification tools (see Section 2.1), which enables symbolic execution. Using symbolic execution, we can prove program properties automatically or generate test suites with higher quality automatically. Having high quality means a test suite finds more faults and preferably using fewer test cases. Symbolic execution approximates this by aiming for a high code coverage. In summary, it has two main advantages—being automatic, which saves human efforts—and thorough within certain theories, which complements the manual methods.

Symbolic execution has been the subject of active research in the last decade, due to the rapid development of hardware and theorem proving algorithms, and the increasing call for automation in software verification. The obstacle of symbolic execution currently is the sophistication of real-world programs. More specifically, the difficulties can be summarised as path explosion and hard-to-reason-about code.

Path explosion Path explosion is an obstacle for SE. The number of potential paths in a program’s execution tree grows exponentially with the number of branching statements, which creates a large search space and implies long constraints to solve (constraint solving is the most time consuming process in symbolic execution [19]). Even the most powerful SE tools can struggle to execute some small programs with a couple of input-dependent loops. Much symbolic execution research investigates solutions to path explosion.

Hard-to-reason-about code This is the problem of reasoning about code that is too complicated (yet not exhibiting error itself), unable to be modelled using supported theories, or unavailable to the SE tool. This kind of code decreases the precision of symbolic execution. We note that achieving precision is against the objective to achieve scalability. For example, reasoning about a hash function can be very expensive while not reasoning about it can lead to unsound or incomplete results. In other words, we cannot consider one while neglecting the other. Because SE’s capability and efficiency of reasoning directly depend on the constraint solvers it uses, these are in part a problem in constraint solving too.

2.3.1 Objectives of SE

In this section we introduce the use cases of symbolic execution. Here, for the interests of the thesis, we classify them by how thorough the SE exploration is expected to be (rather than introducing each application).

Exhaustive Exploration

In exhaustive exploration, an SE tool is expected to thoroughly explore a program, i.e., consider all possible states of a program.

Due to the complexity of SE, we usually specify a time or depth limit. A traditional depth is associated with each symbolic state, defined by roughly the number of conditional statements or similarly, the number of times symbolic execution branches along the path of the
state. The difference is that, a state can branch at other kinds of locations than conditionals, e.g., a division where SE wants to check whether the divisor may be zero.

Some SE tools also allow us to specify a range of values the symbolic variables can choose from, e.g., the length of a symbolic string. This is to be distinguished from specifying a condition on symbolic variables so that SE can explore a specific part of a program, which is an optimisation with approximation, introduced in Section 2.3.3.

Typically, exhaustive exploration is expected in coverage-oriented test generation and program proving.

Selective or Targeted Exploration

In selective or targeted exploration, an SE tool is given directions on what kinds of program states need to be considered and what not. Such expectations (or targets) are typically specified by identifying a part of program for SE, e.g., providing target line numbers or providing target function names.

This is mainly used when a user has a specific goal, such as a line of code suspected to be erroneous, in violation of a program property, missed by a test suite manually generated, or a function that the user thinks needs more intensive exploration.

Compared to exhaustive exploration, this kind of exploration is cheaper and much preferred when such a goal exists. Symbolic execution approaches specialised to solve the problem where they need to reach a goal in a program are also called Directed Symbolic Execution. In directed SE, searcher optimisations are usually used to reach the target code quickly (see Section 2.3.2).

Note that many under-approximation optimisations effectively achieve selective exploration, because they focus the search on a subset of program paths, which they believe covers most of the interesting behaviours of a program. We leave this to Section 2.3.3, while this section mainly discusses selective exploration as a motive to apply SE, rather than a consequence of applying a certain kind of SE.

The Use in Regression Testing, or Incremental SE

A particular interest of using directed symbolic execution is Regression Testing [139] (see Section 2.1.2). In regression testing, a program is only partially modified after the last time a verification tool confirmed its correctness or a test suite was generated for it. Under this assumption, we only need to do verification or generate additional tests for the modified part of the program including the parts of program affected by the change.

There are many ways to approach regression testing. In particular, we can use Incremental Symbolic Execution for such an application. Incremental SE is a series of SE activities that verify the “incremented” parts of a program as its development goes, with the present SE run reusing information from the previous runs. In general, it includes a method to identify modified/affected, or “interesting” code, as well as targeted symbolic execution techniques to specifically verify the thus identified code. The general rationale behind incremental SE is to improve the efficiency by not doing redundant work on code that has been verified previously.

The related work on incremental SE is discussed in Section 2.3.3.
2.3.2 Faithful Optimisations on Symbolic Execution

There are many symbolic execution approaches and each is a combination of techniques to help mitigate the problems of path explosion and hard-to-reason-about code. We summarise them and categorise them by whether and how they create approximation. An SE technique that comes with approximation is either under-approximation, over-approximation, or a mix of both. We focus on how well they handle the aforementioned difficulties.

This section introduces the faithful optimisation techniques. Faithful optimisations do not incur approximations, when given a program that can be captured by certain theories the underlying constraint solvers support. They reject an infeasible symbolic state once there is sufficient information to do so; they do not reject feasible states. Faithful optimisations are intended to keep SE sound and complete.

Searcher Techniques/Heuristics

A searcher plays an important role in SE. A searcher is responsible for the following tasks:

- In coverage-oriented or defect exposing applications, identifying the most promising path to execute next, i.e., finding the path that leads to the most amount of uncovered code.
- In directed SE, identifying the path that can lead to the target.
- Helping symbolic execution avoid corners in a program, such as infinite loops.

These are important tasks that directly decide the efficiency of SE. A searcher’s job can be seen as prioritisation of symbolic states. Searcher techniques aim to improve a searcher’s cleverness when it comes to make such decisions.

The searcher essentially searches in the symbolic execution tree of a program. A symbolic execution tree is typically the result of unrolling the program’s Control Flow Graph (CFG) in a certain way, with the root representing the entry basic block, each node corresponding to a specific basic block, and each conditional creating two children for the current node. A path from the root to a leaf node corresponds to an execution path. A node in a execution tree corresponds to a symbolic state and the node’s depth in the tree is the state’s depth. SE can alter the definition of “conditionals” depending on the verification goal, or approximate the tree (see Appendix A). We can loosely understand a depth as the number of conditional statements on the path from the root to the state’s node.

Most naively, SE tools can search in such a tree using (bounded) Depth-First Search (DFS) or Breadth-First Search (BFS) \([131, 62, 28, 24]\). A big disadvantage of DFS is that it can get stuck in a corner of some symbolic execution trees, such as the ones from a program with infinite loops. A disadvantage of BFS is that it cannot generate many tests at the early stages of symbolic execution, because SE cannot get to the end for most of the paths.

Random search randomly chooses a state to execute, so that it avoids being stuck in a loop like DFS, and it does not have a restriction to expand before deepening like BFS. There are mainly two popular ways of randomly choosing states. One is giving every state the same chance to be selected. If there are \(k\) states, then each state gets a possibility of \(\frac{1}{k}\) to be executed. The problem is that for a loop that produces a lot of states, the states it produced are more likely to be selected than the remaining states, effectively prioritising the loop over the rest of the program. The other randomisation strategy is to give each state a chance proportional to \(2^{-d}\) to be selected, assuming \(d\) is the depth of the state, and it starts from 0.
This randomisation does not prioritise structures that can produce a lot of states, because the sum of their possibilities do not exceed $2^{-d}$, with $d$ being the depth of the entry node of this structure. These approaches are simple and effective, and are chosen (sometimes as default) by many tools [26, 24, 90].

The problem with above algorithms is that they are oblivious to the potentials of the states they choose, e.g., how much code coverage can the states provide. Heuristic search [93, 24, 26, 136, 90, 88, 91] is a solution to this problem. It is important in coverage-oriented applications and especially important in directed SE. Static analysis on the CFG of a program can provide heuristics to searchers, e.g., computing the shortest potential path from a state to a target or uncovered code. Other information collected during the execution can also be used, e.g., the (dis)similarity between path constraints [104]. Directed SE especially relies on search heuristics to reach the specified target. Directed SE has been used to reproduce crashes with limited information [83], reaching potential crashing points [104], reaching changed code in programs [106], etc. [17] We introduce KLEE’s heuristic searcher in Section 4.3.3 and discuss how over-approximation can be an obstacle to deploying certain heuristics.

Some SE tools allow a combination of searchers or heuristics to be run at the same time, such as KLEE’s interleaved searcher [26] that alternates between search algorithms to exploit the strengths of each.

Seeding is also considered a way of prioritising the states, with user guidance. Seeding is to let SE execute using an initial test, which a user believes can lead to the most interesting part of the program. SE can then start the exploration from the initial path, covering many other paths adjacent to it. Seeding can be easily combined with concolic execution (see Section 2.3.3).

Search Acceleration

This refers to techniques that deduce additional information during SE beside traditional path constraints. This information can help SE prune infeasible paths earlier or can be used for prioritisation.

**Interpolation** helps SE deduce infeasibility quickly. Jaffar et al. [76, 78] introduced an interpolation method for program verification. It computes interpolants upon discovery of unsatisfiability. An interpolant captures the generalised “reason” why a logical formula is unsatisfiable, which is also called the precondition of unsatisfiability or unsatisfiable core (see Section 2.2). A most general reason is called a weakest precondition or a minimal unsatisfiable core. An interpolant can be cached for code locations and checked against upon reach. If the current state entails the interpolant, then we know it will fail again to reach the target for which the interpolant is generated, so we do not need to continue executing the state for that target (just terminate the state). Because a perfect weakest precondition is hard to compute, Jaffar et al. approximate it by eliminating less relevant variables or conjuncts using existential quantifiers, unsatisfiable cores, and heuristics.

Loops with dependency on symbolic inputs are capable of producing a large number of symbolic states and are part of the reason for path explosion. Godefroid and Luchaup [65] used loop invariants to accelerate the exploration in loops. They made the observation that an induction variable (IV), a variable that is incremented or decremented by a constant amount in each iteration of a loop, is helpful in inferring constraints for locations that require many iterations to reach. They can infer IVs for a loop by observing concrete execution instances (using concolic execution) that go through it, i.e., observing how variables change in a loop.
Similarly they identify loop guards that are IV-dependent. Loop guards are conditionals where the execution of a program decides whether to execute inside or outside the loop. Being IV-dependent for a guard means that it is of a specific type of Boolean expression, and it changes the same amount towards a flip of its value in each iteration. With this information Godefroid and Luchaup can create a concise constraint that corresponds to the execution of the loop any number of times, and further compute a loop summary (for summarisation, see CSE in Section 2.3.4), basically a constraint that captures a set of ways to execute the loop as well as the execution outcome. Finally they can obtain the critical conditions to reach some code locations which might not be obvious to traditional SE, at an early stage of loop exploration. Assuming the base SE’s soundness and completeness, the exploration using these conditions is sound and complete, except for the cases where there are variables modified in a loop that are not IVs or the guards are not IV-dependent. Earlier usage of invariant generation or a similar idea in symbolic execution includes Păsăreanu and Visser [100], Saxena et al. [113], and Obdržálek and Trtík [96]. Xiao et al. [135] carried out studies specifically on the problems regarding loops in symbolic execution, surveyed the existing solutions to these problems, and finally proposed guidelines for solving them.

**Remarks**  The two types of techniques in this section are highly relevant to our research. In Chapter 5, we propose another usage of unsatisfiable cores similar to interpolation, while we use it as a correction method for over-approximation. We integrated this feature provided by another off-the-shelf SMT solver (see Section 2.2) into our SE tool, and compared with traditional solving as well as incremental solving. Using loop invariant are relevant to our work in the sense that it is related to loop summarisation. Summarisation is introduced later with CSE in Section 2.3.4. Here we note that invariant generation relies on guessing induction variables and has a stricter assumption on a loop’s constitution, such as the dependency between variables and conditionals.

**State Merging**

State merging or path merging refer to a kind of technique that considers multiple paths collectively, by putting their path constraints in disjunction. State merging slows down the exponential growth of the number of cases to be considered by SE. Note that merging does not decrease the space of states, and solving a constraint with disjunctions of many different cases is not necessarily cheaper than solving them individually (consider the worst case: they are all unsatisfiable). Sometimes merging decreases overall performance of SE [70]. Kuznetsov et al. [86] used some heuristics based on the number of shared variables to choose states for merging. Another problem is that path merging techniques that require the exploration of all paths leading to a merge point before merging, complicates or prevents the searcher prioritisation, for which Kuznetsov et al. [86] proposed to relax this restriction. Sen et al. [116] introduced another improvement to state merging via value summaries, which is an alternative to the auxiliary variables used by the other merging approaches in creating disjunctive logical expressions for two states. They showed that this representation is more compact and can let SE proceed without fully supporting the theories of logic formulae from all the merged states.

State merging is a potential solution to path explosion. It is essentially an attempt to downsize the execution tree of SE on a program to a size in a linear relation with the program’s size on the premises that it does not trivialise the problem of SE. To see why, readers
2.3. Symbolic Execution

can also refer to merging in Appendix A.4.2. Again, this in return makes constraints harder to solve. Despite this, it has improved the scalability of SE significantly as reported in the above studies.

Many Compositional Symbolic Execution tools merge states. We discuss such tools in Section 2.3.4. Merging, as one major advantage of SSE, can be used in alternation with DSE [5]. In Section 2.3.3 we discuss DSE and its inherent characteristics that prevent or hamper state merging together with concretisation.

A relevant topic to merging is program and path slicing [81]. In SE, this is to use a criterion to identify certain statements covered by a path, which are collectively a slice of the path. Qi et al. [105] proposed to use a slicing criterion based on the symbolic outputs of a program. They identify statements that the program output depends on in a path, which is called a relevant slice, and compute a path condition over the relevant slice. Essentially, the constraints not from the statements in the relevant slice are dropped, and this condition represents a set of paths (with the same relevant slice condition). Strictly speaking, such a condition is an over-approximation of the union of the path conditions it represents due to the dropped constraints. Eventually, this method computes one test for each different symbolic output (the output domain, however, is not an over-approximation), reducing the number of tests as well as the time to do constraint solving. This method also has potential in CSE, e.g., it can be used to reduce the number of summaries. A similar approach is Symbolic Program Decomposition [112] which also groups program paths during SE based on data dependencies.

Query Optimisations

A general introduction to constraint solving is in Section 2.2. Here, we discuss the constraint solving techniques that are closely related to SE, which are mostly query optimisations, that is, optimising the queries sent to the solver to relieve pressure on the solver.

For whitebox fuzzing, the art of constraint generation is as important as the art of constraint solving. [19]

There are many query optimisations adopted by SE:

- Rewriting and simplification [26, 19, 20]: these include using a different but logically equivalent expression so that the solving is faster, eliminating intermediate variables or redundant expressions, and deducing implications.

- Dividing and caching [26, 130, 19, 82]: these include dividing a set (conjunction) of constraints into independent sets based on the sets of variables they use, and caching solutions to the (independent) constraints.

- Counterexample caching in KLEE [26]: KLEE caches solutions and reuses them. Unlike the above, it can reuse a solution from a constraint to solve another constraint. This is based on three observations about any constraints $c_1, c_2$ on symbolic input $\vec{X}$ and an assignment $\vec{X} = \vec{v}$: if $c_1$ is unsatisfiable, so is $c_1 \land c_2$; if $c_1 \land c_2$ is satisfied by $\vec{v}$, so is $c_1$; if $c_1$ is satisfied by $\vec{v}$, it is possible that $c_1 \land c_2$ is too.

- Other heuristics [19]: for example, limiting the number of flips on a constraint. Note that this particular approach is unsound.

We discuss the use of a constraint solving technique called incremental solving in SE in Section 2.2.2, which has possibly been overlooked by existing approaches [89].
Parallelisation

Another large research area is concurrent symbolic execution algorithms. This kind of technique utilises the hardware resources available today to scale SE [37, 119, 22, 108, 32]. There are many ways of distributing symbolic execution tasks onto different workers. The problem is how to distribute them evenly, avoid redundant work, and ensure independency. The possible ways are

- Partitioning the input space by preconditioning. For example, specifying a disjoint set of constraints on input length. The problem with this approach is that the workload may be highly unbalanced because the cost to explore the different partitions are unknown to static analysis. Staats and Păsăreanu [119] proposed a heuristic to generate better disjoint constraints by inspecting an initial set of path constraints. The initial path constraints are computed from a run of shallow-depth SE prior to the parallel SE.

- Partitioning the execution tree dynamically [22, 32]. This kind of approach assigns subtrees to workers and automatically balances the workload, i.e., assigning or transferring jobs to new or less loaded workers. This method yields better balance than static partitioning.

- Parallelisation at the solver level [108]. This approach parallelises the constraint solving in SE. It also has to deal with the above problems. It can be used in combination with the above methods.

2.3.3 Under-Approximation

In contrast to faithful optimisations, we have under-approximation. Techniques in this category under-approximate the state space for a program. Basic under-approximation methods have already been discussed before, including using time, depth limits, or limits on the symbolic variables. Below is a summary of more advanced techniques.

Concretisation Including Concolic Execution

Concretisation is the process of making a symbolic variable concrete, using a value of a satisfying assignment. This has many use cases:

- Simplifying hard constraints, such as those involving a symbolic index in array updates. MAYHEM [29] and Driller [120] use write concretisation to simplify memory write operations involving symbolic pointers, because these operations are hard to solve. Concretising a symbolic pointer makes it a concrete one, which reduces the number of possibilities the solver has to check during constraint solving.

- Making external calls. An external call is a call to an external function, which is “external” because either we do not have access to the function’s source code or we do not have the desire to symbolically execute the function’s code due to it being already verified, complicated, or hard to reason about. A concrete external call can sometimes be made\(^5\) to obtain partial information about the call. This is a powerful technique that makes SE able to reason (partially) about many kinds of environmental interaction.

\(^5\)Sometimes, a concrete call cannot be made at verification time, e.g., communicating to a server while we are verifying a client offline.
• Creating a concrete test input, which plays an important role in the concolic execution. This is a kind of symbolic execution that makes alternation between symbolic and concrete execution, in which process it uses the concrete input to direct the next round of symbolic execution. We later discuss this in detail.

The third point above is not necessarily an approximation (it is as sound and complete as traditional static symbolic execution provided they are equally capable of reasoning in any given theory), but it is closely related to the other applications of concretisation, i.e., constraint simplification and external calls, which usually are approximations. So we place them together under this category. We also refer to sound concretisation only, as unsound concretisation [59] is specifically discussed later with higher-order test generation (we also take similar definitions of soundness and completeness from this work [59]).

EGT [27] and DART [62] first proposed to use concrete inputs to direct symbolic execution at around the same time. This idea became popular and was adopted and complemented by later work. EXE [28] introduced additional support such as efficient evaluation of symbolic pointer expressions, while SAGE [64] additionally stressed its generational search strategy and its heuristics (see Section 2.3.2), which helped it detect security-related bugs in Microsoft applications [19]. SE based on a combination or alternation of symbolic and concrete execution is called concolic (concrete plus symbolic) execution or Dynamic Symbolic Execution (DSE) in the literature.

**Concolic execution or DSE work pattern** Typically, DSE executes a program path by path. Unlike traditional static symbolic execution, a concrete execution phase is interleaved between paths. Initially, a random input (or a seed input specified in some other way) concretely executes the program, while the symbolic execution engine shadows this process, collecting information about the changes in the program state. At the end of this execution, a path constraint is constructed, and potential branching points and branching conditions are recorded. A prioritisation procedure can then be invoked to rank all potential adjacent paths, and choose one of them as the next to explore. Once a path is selected, DSE generates a test that is expected to execute this particular path. In the case when a test does not execute the expected path, we say divergence [64] happens. (Divergence is the result of imprecision in SE or its constraint solver, which can make the verification unsound, incomplete or both.) DSE cycles until all paths in a program have been explored. Note that beside actually running a test with a real machine, the concrete execution phase can also be achieved by statically evaluating all expressions and conditions along the path. Since this involves only concrete values, its cost is marginal compare to that of symbolic execution.

The advantage of DSE compared to static SE can be generally summarised as being

• More efficient: part of the decision making during the exploration does not involve symbolic variables.

• More robust: behaviours of many hard-to-reason-about statements can be captured via concrete execution.

• False-positive-free: even when there is divergence, an error confirmed by a concrete execution cannot be a false positive.
Since DSE’s emergence, it has attracted attention from researchers and found large-scale applications in industry. Beside the previously mentioned work, the evolution of DSE witnessed CUTE and jCUTE [115, 114], CREST [24], Pex [126], S²E [33], and MAYHEM [29], to name a few. David et al. [43] studied and compared the concretisation and symbolisation policies regarding SE tools’ efficiency. Recent surveys [7, 31] of DSE may be of interests to some readers.

KLEE [26] is a symbolic execution tool closely related with our research. It makes use of concrete execution in many ways:

- It is able to evaluate a statement symbolically or concretely. Concrete execution can be seen as a special case of symbolic execution—when the variables required by the evaluation of a statement are all concrete, KLEE executes it like a real machine would do. This makes KLEE capable of executing concretised test inputs.

- On the above basis, it simplifies the execution using concretisation. For example, it concretises floating point operations, which are not supported (but is supported in its extensions [39, 124]); when there is a call to an external function, for which the source code is not available, KLEE concretises the symbolic arguments, if there are any, and makes an actual call to the external function with these arguments.

- KLEE caches concrete solutions to constraints and uses them to speed up constraint solving and direct execution. This is part of the counterexample caching feature of KLEE. One of the hunches behind counterexample caching is that a concrete solution can sometimes satisfy a stricter constraint than what it has satisfied. For example, after deciding that the valuation \((x \mapsto 1, y \mapsto -2)\) satisfies \(x > 0 \land x + y < 0\), it can cache it and check it on \(x > 0 \land x + y < 0 \land y \mod 2 = 0\). Checking a valuation against a constraint is much cheaper than solving the constraint. Moreover, even if we did not cache a good solution for the current constraint, e.g., \((x \mapsto 1, y \mapsto -3)\) is not satisfying the above constraint, we can still use it as a solution to \(x > 0 \land x + y < 0 \land y \mod 2 \neq 0\), which is also needed to be checked during SE (imagine we are at a conditional statement with if \((y \mod 2)\)). That is, a concrete solution can always find a path in a program. Essentially, KLEE can execute a concrete solution like a real machine executes a concrete test input, with the inherent advantages we discussed above.

Incorporating these techniques, KLEE is able to find bugs in real programs efficiently. For example, it has found bugs in Coreutils that have been missed by thorough testing during its development over a period of more than 15 years [26].

Higher-Order Test Generation

Godefroid [59] proposed higher-order test generation, which is another closely related work to our research. Higher-order means the constraints contain higher-order logic, introduced by uninterpreted functions (see Section 2.2). Uninterpreted functions can be used to represent a function that has not been or cannot be reasoned about by symbolic execution. Their motivation is to use uninterpreted functions as an alternative to unsound concretisation as used in DART [62] which produces divergence, and to extend sound concretisation.

Unsound concretisation does not record the implications of concretisation as constraints. For example, in the code in Figure 2.1, suppose function \texttt{hash} is too expensive to symbolically execute, too hard for symbolic execution to interpret or reason about, or unavailable for
2.3. Symbolic Execution

If \( x = \text{hash}(y) \) \{// hash(42) = 567
\)

\[
\text{if } (\text{y=10}) \quad \text{error()}
\]

Otherwise \{...

Figure 2.1: An example illustrating unsound concretisation.

analysis. Using concretisation described earlier, we concretise \( y \) as 42 and we can partially reason about it. Suppose we get \( \text{hash}(42) = 567 \). Then we know \( x \) has to be 567 to go inside the conditional. However, the concretisation is not considered as a constraint, i.e., we do not require \( y = 42 \). This becomes a problem when we want to reach the error in the code, where we compute \( x = 567 \land y = 10 \) as the path constraint, which cannot guarantee the reach, because \( \text{hash}(10) \) might not be 567. In this case, it is a false alarm as \( \text{hash}(10) \) is 89. In other words, concrete execution can diverge from the prediction with unsound concretisation. (Note that this unsoundness is not with respect to concrete execution, which is always a sound analysis no matter there is divergence or not.) On the other hand, sound concretisation makes \( y = 42 \) a part of the path constraint, and thus guarantees the soundness of the analysis.

As we discussed earlier, sound concretisation under-approximates the set of feasible states. This makes it miss some program behaviours, e.g., it will conclude that the error line is not reachable in the above example, which might be a false claim. In this case, \( (x \mapsto 89, y \mapsto 10) \) can reach the error. Unsound concretisation is an approximation, making both some feasible states infeasible, e.g., \( (x \mapsto 567, y \mapsto 10, pc \mapsto 7) \), and some infeasible states feasible, e.g., \( (x \mapsto 567, y \mapsto 10, pc \mapsto 4) \). Here, \( pc \) is the program counter. Sound concretisation is comparably harder to implement [59].

Uninterpreted functions improve this situation by introducing symbols for functions. In the above example, we can let \( \text{hash} \) take part in constraint solving as a symbol \( h \). \( h \) can be given constraints as well, such as \( h(42) = 567 \). This can be obtained from a concrete execution of the function with input 42 and an observation of the output (567), which process is called sampling. This requires \( \text{hash} \) to evaluate to 567 when given 42 and assumes that it remain consistent, i.e., always behaving like this. Higher-order test generation [59] produces tests from validity proofs. Essentially, this means the uninterpreted functions are universally quantified, with the their samples imposed as an antecedent to a path constraint. For example, with the sample \( h(42) = 567 \), to enter the true branch of line 1 in Figure 2.1 is to show the validity of

\[
\exists x, y \ (h(42) = 567 \Rightarrow x = h(y))
\]

which implicitly universally quantifies \( h \). The test generated is \( (x \mapsto 567, y \mapsto 42) \).

Given enough samples, such an approach can efficiently reason about expensive functions, but it is an under-approximation, e.g., when no samples are given, no tests can be generated for the path mentioned above. The study concluded that higher-order test generation using this technique is more powerful than sound concretisation [59].

Other usages of uninterpreted functions can be seen in related work [1, 66, 11]. In particular, we discuss an application in CSE [1] in Section 2.3.4, where uninterpreted functions are existentially quantified, making the usage an over-approximation.
Remarks. Higher-order SE and the usage of uninterpreted functions are highly relevant topics to our research. In Chapter 4, motivated by three observations, we novelly propose to make a more general kind of function uninterpreted, e.g., user functions, in order to differentiate the roles of code and facilitate partial proving of programs. We do not directly use uninterpreted functions in our presentation of constraints, but achieve similar capability via other methods.

**Preconditioned Symbolic Execution**

Preconditioned SE imposes preconditions on symbolic inputs, thus the name. This is more than a simplification for the search. Avgeronos et al. [4] proposed four kinds of preconditions for efficiently reaching certain types of exploits. Specifically, the preconditions can be or require

- **Empty:** no precondition is imposed.
- **Inputs to have known length:** setting a maximum length for inputs. This length can be automatically computed using static analysis.
- **Inputs to have known prefix:** setting a prefix for inputs, e.g., a file header.
- **Inputs to be concrete:** effectively concretisation for the examination of test cases (that have crashed a program, for instance).

**Aggressive Pruning**

This includes pruning paths that are believed to be less important than others. A simple example is the execution of infinite loops. Symbolic execution tools need to stop unrolling the loop at some point when additional paths from are no longer beneficial, otherwise it might get trapped in the infinite loop.

In theory, symbolic execution achieves full path coverage on programs if it terminates. However, path coverage is often too strong as a criterion and too expensive to achieve. In practice, we use lesser criteria to measure its performance, such as statement coverage. This is backed up by a study [75] showing that the different coverage criteria are not significantly different in their capability of measuring the effectiveness of a test suite, while the coverage criteria do say its quality (or lack of quality in a comparison) from different perspectives.

Note that even though sometimes our goal is statement coverage, we still need to visit every feasible path we can find during symbolic execution, otherwise we might miss reachable statements. Consider the example in Figure 2.2. A greedy statement covering approach will choose to explore the true branch of the first if to cover line 3 and ignore the false branch because it does not gives any extra statement coverage. However, it will later miss line 5. Below we introduce techniques that do such aggressive pruning systematically so that the paths missed out are not as important as the paths explored.

Boonstoppel et al. [18] used a technique call read-write set (RWset) to identify redundancy during SE. RWset discovers paths that cannot lead to new behaviours, by using a combination of liveness analysis and constraint matching. If two states have constraints that are equivalent when projected onto live variables then one of them can be discarded, as they describe identical program behaviour. This work was later extended by Bugrara and Engler [23].
2.3. Symbolic Execution

Anand et al. [3] proposed subsumption checking for symbolic states that discovers certain relations between them to eliminate redundancy. This is due to an abstraction of states. For example, a state that unrolled a loop with repeated actions three times and a state that unrolled four times are a match, and can be seen by summarising the previous two iterations (nodes) as one in the latter state. Then, with a precise enough abstraction, we know if the former state does not violate a property, neither will the latter (similarly for the onward states), in which case we can prune the latter state.

Yi et al. [138] introduced a redundant path elimination method in SE. This is called post-conditioned SE. Its aim is to achieve statement coverage while systematically pruning feasible paths that cannot lead to new statements by detecting common suffixes of paths. If a path’s possible suffixes have all been explored, it becomes redundant with regard to statement coverage.

Incremental SE

As discussed in Section 2.3.1, incremental SE searches in a targeted space of interest, or searches for such a target (using directed SE, discussed in Section 2.3.1), making it an under-approximation compared to an exhaustive search.

Person et al. [102] presented Directed Incremental Symbolic Execution (DiSE), an incremental SE implementation for Java programs, based on Symbolic PathFinder (SPF) [2]. DiSE statically analyses the CFGs of two versions of a program, then computes locations affected by the changes in the new version, before it applies directed SE to the changed/affected code. A similar incremental SE approach that computes the differences between versions of a program statically is KATCH [91].

Yang et al. [137] proposed Memoise which implements incremental SE differently. It uses a trie-based approach, which means to store symbolic execution choices at branching points of a program and reuse them for subsequent SE runs on the same or modified program. This makes Memoise not only capable of doing regression testing but also of a more cost-effective exhaustive exploration using iterative deepening, and directed exploration using the information from a trie. A similar incremental SE approach that memoises results of previous SE runs to speed up the current run is staged symbolic execution [117]. Shadow symbolic execution [99], on the other hand, runs a single SE instance on two versions of programs and verifies the changed part of the program.

ZESTI [92] is an incremental SE tool that reuses test suites generated through previous SE runs or other test generation means, for gradual improvement of the suites.

Other augmentations to SE with incrementality include modular demand-driven analysis [128], the static dependence analysis in DGSE [133], the use of annotations [36], and must-summary checking [63] (specifically for compositional SE, discussed in Section 2.3.4).

A related application is to use symbolic execution to differentiate (or show the equivalence between, as is the case in regression verification [67]) two versions of programs [101, 109, ...]
and further use the results to do refactoring assurance, change characterisation, or test suite evolution, etc. [101]

2.3.4 Over-Approximation

In the above we have discussed under-approximation techniques, which essentially try to focus on (incomplete yet) the important subsets of the search space and apply SE there. Over-approximation is another way of altering the search space in SE. As we define in Section 1.1.2, techniques under this category over-approximate or temporarily over-approximate the state space of a program. Even though this would appear to increase the amount of work for an SE tool to do, its ultimate goal is still to improve the efficiency, as we can see in the following summary.

(Bottom-Up) Compositional SE

Compositional Symbolic Execution (CSE) is a type of symbolic execution, whose distinguishing feature is an additional process called summarisation. Summarisation extracts summaries from a selected part of a program. Summaries effectively contain the information about symbolically executing the corresponding program part. As a result, a CSE tool need not execute this part of the program, it only needs to use the summaries when it is encountered.

The idea behind CSE is that some particular code fragments in a program are frequently encountered, e.g., a function or a loop body. Although they are visited under different contexts, the set of constraints they can produce is always the same. This can be exploited by using a method to obtain the set of constraints prior to the execution of a code fragment, so that we can reuse this set to propagate the states instead of executing the code repeatedly. This method, which is therefore given name “summarisation”, together with the reusing processes, should be cheaper compared to the repetitive execution.

Existing summarisation approaches summarise functions, while essentially a “function” can denote any program part to compute summaries from [58]. We might use “function” to generally represent any code fragment in this section, while in Chapter 3, we explicitly talk about “code fragments” because the finer granularity is a feature of the ideas presented there.

Path and function summaries A summary consists of path summaries from different paths in a function. Suppose a path is denoted by $R$. A path summary can take the form $\phi_R = pre_R \land post_R$, which consists of a precondition and a postcondition. A precondition is a series of constraints on the function inputs that, when satisfied, guarantees that the execution with these inputs will follow path $R$, provided that the summarisation procedure is sound. In other words, a precondition is (as strong as) a path constraint of $R$. A postcondition is a logical description of the function outputs when it is executed along path $R$. After systematically visiting many paths in a function, CSE can compute a comprehensive function summary $\phi = \bigvee_R \phi_R$. If we can obtain a summary consisting of a sufficient number of path summaries, then when symbolic execution encounters a call to this function, it can check if the current symbolic state satisfies a $pre_R$ and if so, update the state with the corresponding $post_R$ directly instead of executing the function. How many paths can be obtained, or how weak $\bigvee_R pre_R$ is, directly depends on the context that is assumed during the computation of

\[6, 16, 53\]
2.3. Symbolic Execution

<table>
<thead>
<tr>
<th></th>
<th>Top-down</th>
<th>Bottom-up</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td>Sound</td>
<td>More reusable</td>
</tr>
<tr>
<td></td>
<td>Can exploit the context</td>
<td>Once and for all</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>Limited reusing</td>
<td>Too many summaries</td>
</tr>
<tr>
<td></td>
<td>Recomputation</td>
<td></td>
</tr>
</tbody>
</table>

There are many ways to do summarisation, which can be categorised as top-down or bottom-up, hence we also have top-down or bottom-up CSE that use the corresponding summarisation techniques.

**Top-down summarisation**  Top-down summarisation computes summaries only when it sees a function call during symbolic execution, and it computes the summaries with respect to a certain calling context. Simply speaking, when a symbolic state reaches a function call, it will focus the exploration on the function by repeatedly backtracking in it, as opposed to executing what is after the call. Each time it reaches a function exit it will obtain a summary, computed from the path constraint. Because this constraint is based on a calling context, so is the summary. Eventually, symbolic execution will compute many summaries for this call and it can then obtain a comprehensive function summary.

**Bottom-up summarisation**  A bottom-up summarisation is a local symbolic execution on the functions to be summarised without any context information. Similarly, it also gets a summary at the end of each function path, which does not have dependency on a context. After the summarisation, SE can symbolically execute the program globally or compute another higher-level summary, reusing the previously computed summaries.

The advantage and disadvantage of using each type of summarisation is shown in Table 2.1. The most significant difference is that because the computation involves context constraints, top-down summarisation always produces feasible summaries, which means they represent execution that will happen for a function when the program are given certain inputs. On the other hand, bottom-up summarisation can generate a lot of summaries including infeasible ones. Symbolic execution might waste time on attempting these infeasible summaries, i.e., trying to go into the paths represented by them but only to find that these paths are infeasible after checking the preconditions. Because top-down summarisation has the context information, it can use concrete context information to partially reason about statements falling out of the supported theories.

On the other hand, top-down summarisation is not as reusable as the bottom-up kind, because the summaries are specific to a context. They are reusable when a similar context
is encountered. Similarly, we have to compute summaries again if symbolic execution encounters a very different context that makes the same call. On the other hand, bottom-up summarisation does not have such a concern.

Part of our work is inspired by the rich literature on top-down CSE. Although the most basic top-down CSE is not over-approximation, we introduce them here, as the counterpart of bottom-up CSE, which is discussed later.

The use of summarisation and the fundamental idea of CSE were first proposed as a potential solution to the path explosion problem by Godefroid and Klarlund [61]. Later the idea was solidified and implemented in SMART [58] as an augmentation to dynamic test generation [62] (see Section 2.3.3). SMART uses top-down summarisation, as introduced earlier, which computes function summaries on demand. A top-down approach combined with concolic testing gives SMART an advantage apart from what we have discussed above: it can reason about more types of statements than a bottom-up approach. Let us use an example from the SMART paper [58] here. Suppose some code to be summarised contains a conditional with \( \text{if} (\text{hash}(x) = 100) \), which is hard to reason about and to satisfy without any information on \( x \). However, with some context information that is taken into consideration by top-down summarisation, it is possible that this constraint is easily satisfied. On the other hand, compared to a bottom-up approach, SMART’s summaries have less reusability, because they are context reliant. When the summaries do not match the calling context, a new summary needs to be computed. SMART does not over-approximate during the above process, and it is as sound and complete as its underlying DSE tool, DART [62].

Many later applications of CSE adopted the concepts and methodology from SMART. Anand et al. [1] used CSE in a directed SE problem, i.e., to find a path leading to a specified target (see Section 2.3.1). Their summarisation computes summaries lazily, in contrast to the full computation in SMART [58], which makes the function summaries incomplete at some stages but reduces the cost of summarisation. The reason why they can do this is that when we are only searching for a target, some paths do not need to be explored and similarly some summaries do not need to be computed. As a result, symbolic execution can determine the reachability of a target earlier. Another key contribution is the use of an uninterpreted function as an over-approximation of a function that is not fully summarised. Uninterpreted functions allow symbolic execution to know earlier whether a target is unreachable.

Based on the same idea, Christakis and Godefroid [34] presented IC-Cut which has a more selective way of summarising functions, as the previous work did not consider how a strategy of summarisation can affect the overall performance. IC-Cut uses the function interface and looks at how constrained the function arguments are to determine whether or not to summarise a function. The evaluation showed that there is good reason to summarise the “low-complexity” functions, i.e., with unconstrained symbolic inputs and at most one output, which is inspiring for us.

Another improvement in CSE is the granularity. Avgerinos et al. [5] presented Veritesting which is capable of performing SSE on specific code fragments in the middle of DSE. This is similar to a top-down summarisation process, with selective merging to reduce path explosion. A difference is that they do not use memoisation on the summaries. Another highlight of veritesting is the transition between SSE and DSE: DSE is used to handle hard-to-reason-about code such as system calls, and loops that are expensive to unroll. DSE is resumed after a transition point identified by using a recovered CFG.

There are also applications of bottom-up summarisation. Qiu et al. [107] introduced a
CSE approach based on memoisation trees. Summaries are recorded in this data structure which memoises the choices taken for each feasible path during the summarisation process. Rojas and Păsăreanu [111] combined CSE with partial evaluation to further improve its scalability. Similar to the previous function summarisation techniques, these two CSE adaptations summarise methods (for Java programs). Ognawala et al. [97] introduced another bottom-up compositional analysis based on KLEE. Their approach first looks for vulnerabilities in isolated functions, then it performs compositional analysis to confirm what is found is the previous process. Ognawala et al. also presented a ranking method for the vulnerabilities according to the properties of the functions where they are located. The ranking method takes into account the functions’ depth, degree of exposure to the environment, etc. As a result it evaluates the severity of the vulnerabilities so that a user can examine them most efficiently. Strejček and Trtík [121] introduced a slightly different loop summarisation approach that over-approximates. Unlike the above techniques, this over-approximation leads to unsoundness (in our terminology, i.e., falsely claiming that a target is reachable).

The work on CSE changed the landscape of SE research. Because CSE potentially introduces over-approximation, Vanoverbergh and Piessens [129] carried out studies about the additional conditions on a CSE approach for certain verification criteria to hold, namely precision and progression, which are equivalent to soundness and completeness in our discussion.

Remarks We conclude the literature review on CSE here. CSE, as an solution to the redundancies in symbolic execution, has attracted much attention from researchers. However, several limitations have not been addressed. First, existing approaches summarise functions instead of arbitrary code fragments, not fully realising CSE’s potential. Specifically, there are loop summarisation in combination with invariant generation [65], and veritesting [5] that interleaves DSE and SSE, where SSE can be used on code fragments. Secondly, existing approaches discussed the functions suitable for summarisation, but not discussing or comparing different strategies in choosing summarisation targets such as functions and code fragments, and not discussing why these strategies affect the efficiency of CSE. This is especially important in bottom-up summarisation, which is an over-approximation. Unlike bottom-up summarisation, a top-down approach can choose an eager summarisation strategy, because it does not have to deal with infeasible paths from the over-approximation of the context information. An example is veritesting [5], where SSE is enabled starting from any branching statement and ending at a transition point such as an external call. In Chapter 3, we present our suggestion for generalisation and improvement.

SE for Program Parts

A research area that is closely related to bottom-up CSE is program partitioning. It facilitates the program verification and testing procedure by considering a program as a collection of independent units or components. Since symbolic execution is a combinatorial search problem in program components, this is especially useful because it dramatically reduces the complexity of the tasks. The drawbacks of partitioning include false positives due to the ignorance of the relation between program components. This is said with respect to an error-detection task. A false positive is to conclude that an error exists while it actually does not. On the other hand, it can prove program components correct faster than the traditional approaches, if they really are correct.
Here we discuss the partition and verification techniques specifically used in symbolic execution. Bottom-up CSE, incorporates the process of summarisation, which does not depend on a context and can be considered as symbolic execution on certain parts of a program. A principle of program partitioning is to minimise the number of false positives, while a summarisation process also needs to minimise the number of (infeasible) summaries.

Under-constrained symbolic execution [52, 110] works on specific functions directly without a context. Ramos and Engler [110] proposed to use it for the verification of function patches (see incremental SE in Section 2.3.1), since its advantages lie in the verification of small pieces of code. Another potential use is in combination with checkers of specific behaviours, such as memory leaks, so as to verify deep parts of a program quickly. Ramos and Engler proposed to use automatic heuristics and preconditions from users to suppress false alarms.

Tomb et al. [127] proposed variably interprocedural analysis in SE. Compared to whole-program analysis like standard SE, or intraprocedural analysis where invocations are approximated, this approach allows the specification of a call depth for the analysis of each method such that the analysis can inspect a number of invocations originated from the method up to the depth. This method is especially useful for, e.g., methods that use accessor method to get field values. Compared to under-constrained SE, where only path prefixes are skipped, this approach is theoretically more cost-efficient since it also abstracts away deep invocations that exceed the call depth—one of the conclusions of the study is that a larger depth, although helpful in removing false positives, does not noticeably increase the error detection rate of the analysis and is very expensive.

Chakrabarti and Godefroid [30] discussed program partitioning for symbolic execution. What is especially interesting to us, is the discussion about highly-intertwined functions and the heuristics to identify them. Some functions have strong relations between them and symbolically executing them in isolation creates a lot of over-approximation, similar to how bottom-up summarisation creates too many infeasible summaries. Therefore, putting them in the same partition is crucial to the efficiency of SE. The heuristics can be generally summarised as follows: two functions belong to the same partition (i.e., they are highly dependent) if one calls the other and the callee is not popular, e.g., it is only called by this caller; the functions that call similar functions are more related than the functions that call disjoint sets of functions. The experiment showed that this partitioning method can achieve better coverage-false-alarm ratio. Similar discussions about the applications of over-approximation techniques in the same regard can be found in multiple places in this thesis.

**Backward Symbolic Execution (BSE)**

Traditional SE tools mimic real program execution, i.e., they start from the program entry and end at the exit(s) or the detected error(s). On the other hand, backward [90, 125, 49] or bidirectional [10] SE tools start from the supposed “goal” and search for the route(s) to the entry. From static analysis point of view, there is no need to follow the specific order defined by the execution of the program: as long as the purpose of generating path constraints (and eventually generating tests) is fulfilled, we should be able to use whatever approach is more efficient or applicable.

BSE is a popular approach for directed exploration (see directed SE in Section 2.3.1), where a target is assumed available prior to SE. Ma et al. [90] achieved BSE using function call chain tracing. This is a function-wise backward reasoning approach. Within a function,
forward symbolic execution is used, but after that, function summaries are used to construct interprocedural path constraints in a backward manner. Thummalapenta et al. [125] introduced method sequence generation to test object-oriented programs, which implements a similar back-tracing idea for method call chains instead. Dinges and Agha [49] introduced symcretic execution, which is the earliest attempt of combining concrete execution with backward reasoning in SE. They use forward heuristic search when BSE cannot reason about particular parts of the code. The applications of forward heuristics in directed symbolic execution are discussed in Section 2.3.2 regarding searcher techniques.

BSE is an over-approximation because it cannot guarantee that at all stages, each symbolic state it considers is feasible, due to the order of collecting information. The symbolic execution trees of traditional and SE and BSE are very different, even for function/method back-tracing. They are the results of unrolling a program’s CFG in different ways.

Over the past decade, the combination of symbolic execution and concrete execution has made forward reasoning more efficient [62, 64, 126, 114, 80, 5]. However, BSE is inherently incompatible with concolic execution, because on the one hand, concrete execution does not go backwards, and on the other hand, a backward trace can terminate anywhere prematurely, i.e., at a conditional or non-conditional, before reaching the entry. Recall that using concrete values as search guidance in SE is in part based on the fact that no matter what values a state has, it can always find a path to go forward, but in backward reasoning, some states with certain concrete values cannot go backwards when encountering some statements. For example, all states \((x \mapsto _)\) can be propagated forwards at statement \(x := 1\), while only \((x \mapsto 1)\) can be propagated backwards at this statement and the others have to terminate, because they should not have existed at the place where an assignment has made \(x \gets 1\). This does not mean they cannot be used in alternation or in complement with each other, but it limits the usage of concretising in BSE and makes backward reasoning more vulnerable to scalability problems than forward reasoning.

Baluda et al. [10] proposed bidirectional SE, incorporating both directions of reasoning and utilising the strengths from both of them. They also proposed a method to decide, for each symbolic execution step, which direction should the analysis proceed in to increase scalability.

Program Abstraction

Program proving are commonly achieved by program abstraction. Here we introduce some notable program proving approaches in model checking. SLAM [8, 9] can do model checking on programs through abstraction and iterative refinement. Abstraction is to over-approximate program executions, essentially defining the state space of consideration; refinement is to check certain executions in it and tell if they are erroneous—if so, it computes a witness, otherwise it narrows down the space by preconditioning. This paradigm is referred to as the CEGAR [38] framework. BLAST [74, 73] uses lazy abstraction to speed up the above process. Lazy abstraction refines the abstract model of a program on demand and avoid repeated work on the same part of the model. Many SE approaches [68, 14, 66, 15] adopt a similar method when it comes to proving.
2.3.5 Hybrid Approaches

Some techniques belong to neither of the above categories, as they use both under-approximation and over-approximation. This may happen as a side effect of using an idealised program model (say, assuming proper integers instead of fixed-width integers, ignoring non-linear expressions or floating point numbers, using logical memory models [8], etc.). This may be combined with unsound concretisation [59], and some non-semantic-preserving program testability optimisations (Section 2.1.2). Moreover, symbolic execution is dealing with sophisticated programs in practice (and it can be engineering intensive). Many mature SE tools are a combination of the above algorithms and methods, as well as other related techniques such as constraint solving and fuzzing. We briefly discuss related work that feature a combination of traditional approaches.

SYNERGY [68] and DASH [14] combined program testing and verification, using them to complement each other. That is, they used proofs of a program to guide testing to erroneous program parts (or avoid parts proved safe), and concrete execution to refine the results of abstraction. During the compound analysis, they maintain a data structure for executions known to be feasible, and a separate data structure that is an abstraction of potential executions, with the intention to grow the former and refine the latter. DASH is an improvement over SYNERGY with a new refinement technique, support for pointers, and interprocedural analysis. Christakis et al. [35] introduced an improvement to the integration of verification and testing. SMASH [66] also combined under-approximation and over-approximation, more specifically, may analysis and must analysis. May analysis computes may information which is true for all executions of a program, in this case, bottom-up summarisation; must analysis conversely computes must information that must be true for a particular execution, in this case, top-down summarisation and concretisation. SMASH applies these two techniques in alternation, bearing the idea in mind that may analysis can easily rule out a set of executions irrelevant to the analysis objective, as we see in Section 1.1.2, and must analysis efficiently reason about complex executions, as we see in Section 2.3.3 about concretisation.

Veritest [5] allows alternation between DSE and the traditional SSE, based on the idea that they are good at analysing different kinds of code: DSE is good at hard-to-reason-about code, e.g., system calls, as discussed earlier; static SE can perform analysis on multiple paths at a time and merge them so as to avoid path explosion (see state merging in Section 2.3.2). Veritest, implemented as MergePoint, has a method to identify transition points where it switches between the two approaches, to achieve better performance than using them separately. MergePoint has found over 10,000 unique bugs in Debian programs.

Bidirectional symbolic execution [10] proposed to combine forward reasoning and backward reasoning to increase the coverage of SE. This is motivated by the fact that the two approaches collect reachability information from different code location: forward reasoning has the context information to confirm reachable locations or identify locations not reached yet; backward reasoning can quickly collect the reachability conditions of those unreached locations.

S2E [33] provides a platform for building program analysis tools. In order to build tools suitable for different applications, it support a number of execution consistency models, which admit different sets of program paths based on tailored assumptions and effectively decide what kind of approximation is performed by a tool. It increases the scalability of tools on top. Driller [120], for instance, uses a model from this work.
Chapter 3

Compositional Symbolic Execution and an Improvement


In this chapter, we look at an application of over-approximation in symbolic execution—Compositional Symbolic Execution (CSE). CSE has been proposed as a way to increase the efficiency of symbolic execution. Essentially, when a function is symbolically executed, a summary of the path that was executed is stored. This summary records the precondition and postcondition of the path, and on subsequent calls that satisfy that precondition, the corresponding postcondition can be returned instead of executing the function again. The process of symbolically executing a function to obtain summaries is called summarisation. Summarisation usually has to over-approximate the calling context of the function being summarised to make the resulting summaries reusable. In this chapter, we study this particular application of over-approximation in symbolic execution.

In particular, we discover that there is a shortcoming using functions as the unit of summarisation, which is that it leaves the symbolic execution tool at the mercy of a program designer, essentially resulting in an arbitrary summarisation strategy. In this chapter, we propose the use of fine-grained summaries, in which blocks within functions are summarised. At such a fine-grained level, symbolic execution of a path effectively becomes the concatenation of the summaries along that path. Fine-grained CSE is a complement and generalisation of CSE. More importantly, we explain why summarisation should not be applied blindly, and then propose three summarisation strategies. Using our compositional symbolic execution prototype Cirrus, we perform a preliminary experimental evaluation, demonstrating that they can improve the speed of symbolic execution by reducing the number of calls sent to the underlying constraint solver.

3.1 Background

We first revisit CSE and discuss the role of “direction” in symbolic execution, as it also concerns one of the features of fine-grained CSE—it being compatible with both forward and backward reasoning.
3.1.1 Introduction to Compositional Symbolic Execution

Compositional Symbolic Execution (CSE)\cite{58,1,34,107,97} is one strategy to help alleviate path explosion. CSE reuses previously generated path constraints to reduce the pressure on path search, constraint generation and most importantly, constraint solving. It can intelligently eliminate redundant constraint solving, and can be embedded into other types of symbolic execution (dynamic, for instance).

In Section 2.3.4, we have seen the idea of CSE: if we can obtain a summary for a function path consisting of precondition (in this thesis, entry condition) and a postcondition, then when symbolic execution encounters a call to this function, it can check if the current symbolic state satisfies the precondition and if so, update the state with the corresponding postcondition directly instead of executing the function path; if we have systematically collected a number of summaries for the paths in the function, then we can frequently reuses the summaries at the calls to the function, thus saving a lot of time in SE.

An approximation of CSE’s cost using execution tree is discussed in Appendix A.5.1.

Ways of Computing Summaries

We call the process of computing summaries summarisation. There are generally two types of summarisation: top-down and bottom-up summarisation. We have introduced them in Section 2.3.4. Top-down summarisation assumes a specific calling context during the computation of summaries, while bottom-up summarisation assumes the most general context—without additional information provided specifically to the summarisation module, it is usually True. Therefore, bottom-up summarisation is an over-approximation and top-down summarisation is not. The advantages of bottom-up summarisation are that the summaries are more reusable and it does not requires repeated summary computation, but the disadvantages are that there are possibly too many summaries due to lack of context and that it cannot take advantage of concolic execution during summarisation like the top-down approach can.

Our research mainly focuses on over-approximation techniques. We assumes bottom-up summarisation in the discussion of CSE in this thesis.

Merging the Paths in a Summarised Function

Merging symbolic execution is a method to counter path explosion (see Section 2.3.2). It makes SE unable to distinguish the different paths in a certain piece of code. Specifically, some summarisation techniques\cite{58} merge the paths in a function. As a result, the guarantee to fully cover the paths in the function is sacrificed to obtain efficiency. This is also based on the idea that some functions are not important enough to be fully explored.

In our work, we choose a different approach, i.e., we still enumerate the different paths in the summarised code. First, our approach summarises arbitrary code blocks, and they might not be very different in their importance than their context. Secondly, we are also interested in how much more efficient CSE is than traditional SE without such loss of path coverage. In the example in Figure 3.1, we can still see the potential of efficiency gain without path merging. We are specifically interested to see the effect of this particular property of CSE, even though it makes our tool Cirrus less scalable in practice.
3.1.2 Direction of Symbolic Execution

Static symbolic execution can be categorised as forward reasoning or backward reasoning, according to the manner in which it searches for feasible paths and collects constraint information. In general, backward reasoning is not reacting as quickly to an infeasible path as forward reasoning. Forward reasoning makes decisions to discard a branch choice by looking at the conditions at the beginning of the branch, while backward reasoning explores in a branch before it reaches the condition that tells its feasibility. Moreover, some useful information is not made available to a backward approach in time, e.g., initialisation of variables.

On the other hand, when doing directed symbolic execution—symbolic execution towards a specific statement—the conditions near the search target are not made available in time to a forward approach. Two optimisations can be applied in this situation to speed up symbolic execution—heuristic search and Backward Symbolic Execution (BSE) based on backward reasoning. A backward approach could be superior to a forward one in certain cases because it starts with critical reachability information. They can also be combined together. We have discussed the related work in Section 2.3.4.

3.2 Motivation

In this chapter, we investigate the use of fine-grained summaries in symbolic execution, that is, the use of summarising blocks of code within functions. Summaries are generated by collecting condition pairs, which describe the entry condition (the weakest precondition) of a code fragment, and the postcondition achieved under that weakest precondition. Summarising at a finer granularity provides more opportunities for reusage of summaries, thus our hypothesis is that fine-grained summaries will reduce the number of calls to a constraint solver. If summaries are produced for all blocks, path constraints can be completely built by summaries. The result is that the execution can be forwards or backwards.

3.2.1 Improving Efficiency

Summaries are used for fast referencing of generated constraints, saving time when exploring identical subtrees. Consider the program snapshot in Figure 3.1 and its loop body Control Flow Graph (CFG) in Figure 3.2. Conventional symbolic execution unrolls the loop into consecutive copies of the loop body (from line 3 to line 10) guarded by the loop condition, sharing the same subtree structure. In the subtree structure, there are in total four possible routes (one of which is infeasible) and three forking points generated from the nested if statements. Hence six constraint solver calls (two for each fork) are made each time the search deepens into a loop body copy, with one call known to return failure each time.

A useful summarisation in this case spans line 3 to line 10. This summarisation focuses on the corresponding subtree and summarises it into three subpath constraints (the summaries), discarding the infeasible subpath. Each time the search encounters the loop body, these summaries can be concatenated to form new path constraints. After three subpath constraints are cached as summaries, only the blue calls (call A, B, and C in Figure 3.2) are needed for each search in the loop body. Consequently, each search deepening now takes only three solver calls, with comparably little cost (if the search needs to unroll this loop many times) to perform summarisation before hand. It is possible that a better summarisation spans line 2 to line 10, that is, it may be better to also take the loop condition into consideration. Part of
Chapter 3. Compositional Symbolic Execution and an Improvement

1. \( x := \text{read} \)
2. while \((x > 0)\) {
3. \( a := \text{load} sp + x \)
4. if \((a != 0)\) {
5. \( b := \text{load} bp + x \)
6. if \((b = a)\) {
7. if \((b = 0)\) {
8. \( \ldots // \text{dead code} \)
9. } \}
10. \( x := x - 1 \)
11. }

**Figure 3.1:** A motivating example for fine-grained summaries.

![Control flow graph](image)

**Figure 3.2:** Control flow graph (simplified) of the loop body in Figure 3.1. We mark the locations where constraint solver calls are made before and after using summarisation.

our research is to observe the effectiveness of the fine-grained summarisation on arbitrary programs.

### 3.2.2 Combination with Backward Symbolic Execution

While having potentials in directed SE as discussed in Section 3.1.2 and Section 2.3.4, BSE cannot use any concolic execution techniques to avoid path explosion, unlike many forward reasoning approaches, which makes it less scalable. However, with the help of fine-grained summarisation, this situation can be improved.

Our summarisation technique can be flexibly combined with both reasoning methods. The order of reasoning about the instructions in a program depends on the order of summary concatenation. In the explanation of our method, we assume the use of forward reasoning unless we specify the direction. We note that the discussion is similarly applicable to backward reasoning.

### 3.2.3 Contributions

We propose the use of fine-grained summaries in symbolic execution. We call this idea FGCSE and we have an preliminary implementation Cirrus based on it (with a backward reasoning manner but not limited to this specific direction of reasoning). The major innovations are as follows:

1. Compositional symbolic execution based on a “fine-grained” summarisation concept. Fine-grained summarisation complements function-level summarisation and generalises loop summarisation.
3.3. Fine-Grained Compositional Symbolic Execution

2. Awareness of program characteristics that affect the effectiveness of fine-grained summaries, as well as three specific summarisation strategies based on our analysis and discussion in this chapter.

3. Combination of CSE and BSE, with evidence showing that the fine-grained summarisation can improve the path explosion resistance of BSE in Section 3.6.

We have performed a preliminary evaluation of Cirrus on eight small but non-trivial programs, measuring execution time and the number of calls made to the underlying constraint solver. The results demonstrate that fine-grained summarisation can be used to improve execution time due to a reduced number of calls made to the solver, but that its effect is dependent on the properties of the program under test. Further, it demonstrates that the summarisation itself has a relatively small cost if a suitable strategy is applied.

3.3 Fine-Grained Compositional Symbolic Execution

In this section, we introduce our generalisation of CSE based on the idea of summarising code fragments. We first introduce the necessary background for the section. We then explain how we produce condition pairs, and how to perform concatenation and de-concatenation operations on them and eventually construct path constraints.

3.3.1 Variable Versioning

Cirrus works on programs written in a C-like language. However, input programs are first translated to an Intermediate Representation (IR) typical of three-address code or assembly languages.

Compilers often produce three-address code in Static Single Assignment (SSA) form [41]. We do not assume that IR programs come in this form (indeed our compiler to IR does not produce SSA). However, during symbolic execution, we attach version numbers to all the variables. Such as

\[ x_{i+1} := x_i + y_j \]

The subscripts indicate the variables’ “versions”. This allows distinction between multiple occurrences of the same variable, and identifying modifications on a variable.

Standard SE does not need versioned variables because only the current value of each variable matters in a symbolic state. In Cirrus, we use versioned variables so that summaries can be reused under different contexts. More specifically, we need to record the relation between oldest versions of variables and the newest versions of variables. For the same reason, the version numbers in summaries are decided in symbolic execution time rather than compile time.

This is achieved by alignment. Using the previous example, when building the constraint of a path, we align \( x_i \) to the latest version of \( x \) in the current state, and \( y_j \) the latest version of \( y \). The state is then updated so that \( x_{i+1} \) is then the newest version of \( x \).

More generally, in a path \( R; I \) (path \( R \) followed by instruction \( I \)), we align every variable in \( I \) to be consistent with \( R \). Versioning a variable (named) \( x \) in \( I \) is by making the call

\[
\text{VERSION}(R.History, x, x.Mod, \text{True})
\]
The **VERSION** algorithm is listed below as Algorithm 3.1. Here, \( R.\text{History} \) records all the variables and their associated versions in \( R \); \( \text{MAXVER}(R.\text{History}, x) \) and \( \text{MINVER}(R.\text{History}, x) \) retrieve the maximum and minimum version of a variable (named) \( x \) in \( R \) respectively. If there is not a single version of it recorded, then they return 0. \( x.\text{Mod} \) represents whether this variable is modified in \( I \). Using the above example, on the left-hand side \( x_{i+1}.\text{Mod} = 1 \) and on the right-hand side \( x_i.\text{Mod} = 0, y_j.\text{Mod} = 0 \). \( \text{IsPost} = \text{True} \) indicates this instruction comes after \( R \) in execution, i.e., this is forward reasoning and we can do backward reasoning in a similar way by passing in a False. This call returns a number greater than the historical maximum by 1 if the variable is modified, and 0 otherwise. For example, if \( \text{MAXVER}(R.\text{History}, x) = 5 \) and \( \text{MAXVER}(R.\text{History}, y) = 30 \), then \( I \) in a path \( R; I \) will be aligned:

\[
x_1 := x_0 + y_0 \quad \rightarrow \quad x_6 := x_5 + y_{30}
\]

**Algorithm 3.1 Variable versioning.**

```
procedure VERSION(History, VName, Mod, IsPost)
    if IsPost then
        Return MAXVER(History, VName) + Mod
    else
        Return MINVER(History, VName) - Mod
    end if
end procedure
```

In Cirrus, we need to align not only variables from a single instruction, but also a summary, representing a sequence of instruction that has already been aligned. Assuming the same \( R \) as the above (no occurrence of \( z \)), we align as follows for the two instructions:

\[
x_1 := x_0 + y_0 \quad \rightarrow \quad x_6 := x_5 + y_{30}
\]

\[
x_2 := x_1 + z_0 \quad \rightarrow \quad x_7 := x_6 + z_0
\]

The alignment of \( x_2 \) is achieved by passing 2 to \( \text{Mod} \). More generally, to align a variable \( x_i \) in \( R' \) to \( R \) in path \( R; R' \), we call

\[
\text{VERSION}(R.\text{History}, x, x_i.\text{Mod}, \text{True})
\]

where

\[
x_i.\text{Mod} = i - \text{MINVER}(R'.\text{History}, x)
\]

Note that here we require a summary (corresponding to \( R' \)) to have its own version record for all variables in itself.

After aligning \( I \) (or \( R' \)) like the above, the maximum or minimum versions of certain variables change. we need to update the version record of the \( R; I \) (or \( R; R' \)) accordingly.

In addition to regular variables, we version the memory as well. We treat it as a single array variable. The versioning of memory is similar except that we consider a store statement as a modification to the memory. We introduce our memory model in Section 5.3.1 which is more maturely implemented in Cirrocumulus, our later FGCS tool. Here, we note that using a single array in memory modelling is relatively expensive.

By allowing variable versioning and aligning, we can store summaries in the form of reusable constraints, as opposed control flow choices. This saves us constraint generation time but introduces longer constraints (albeit they are not more logically complicated). On
the other hand, each time the search deepens we add new constraints and each time it backtracks we can simply throw away the constraints on the current branch. In these processes the most parts of the path constraint never change. A small benefit is that we do not need to maintain copies of partial path constraints (similarly, states) for the purpose of backtracking (assuming DFS). More importantly, this similarity maintained among queries is required by incremental solving. The usage of incremental solving in FG CSE is introduced in Chapter 5.

3.3.2 Entry Condition and Postcondition

CSE generates path constraints by putting together constraints of the path components. In our approach, the smallest component of a path that we can collect constraint information from is a single instruction. The idea of summarisation is to use entry conditions and postconditions to describe any instruction $I$ or sequence of instructions $R = I_1; \ldots; I_k$ with line numbers $l_1, \ldots, l_k$ respectively. Formally, the sequence $R = I_1; \ldots; I_k$ corresponds to $\{S_1, \ldots, S_k \mid pc(S_1) = l_1 \land \ldots \land pc(S_k) = l_k\}$, where $S_1, \ldots, S_k$ is call the execution given by the sequence of concrete states. Here, $pc()$ gives the program counter of a state.

Definition 2 (Summary and condition pair) A summary on an executable instruction sequence $R$ provides the conditions under which $R$ is executed from the beginning to the end without a control flow escape, as well as the consequences of successfully executing $R$. It can be represented by a condition pair:

$$\text{CP}(R) = \left\{ \begin{array}{l} \text{EC}(R) \\ \text{PC}(R) \end{array} \right\}$$

The entry condition $\text{EC}(R)$ is the weakest proposition under which $R$ is executed. The postcondition $\text{PC}(R)$ is the strongest proposition about the state that results from execution of $R$ under its entry condition.

Hence if $R$ consists only of a basic block of instruction(s), such as assignment(s), then $\text{EC}(R) = \text{True}$, indicating that $R$ will be executed to its end, no matter what. The entry condition for a sequence is only constrained by conditional jumps (such as branch and loop guards). By definition, $R$ corresponds to a unique program path. So we can say $\text{EC}(R)$ is the path constraint. Note that $\text{EC}(R)$ is always logically implied by $\text{PC}(R)$, assuming the variables are named with consistent version numbers. In other words, in a condition pair, the postcondition is always at least as strict as the entry condition. This is more clearly seen when we introduce concatenation.

For each individual instruction other than a conditional branch, its condition pair is generated before summarisation. This is shown in Figure 3.3. This table does not show the condition pairs for function calls and returns, because they are not the focuses of this chapter. Cirrus does not consider interprocedural summarisation, but it can be achieved in a similar way without fundamental difficulty. Note that interprocedural summarisation could result in an explosion of summaries. (In Chapter 5 we introduce Cirrocumulus which is based on FGCSE and can perform interprocedural analysis, but not interprocedural summarisation.)

A conditional branching instruction has no condition pair, but gets one during concatenation. For example, if $I$ is the instruction if $(b) R_1$ else $R_2$ then $I$ will get one of two condition pairs the moment it gets concatenated, according to the sequence of code it is concatenated

\footnote{The notation $R$ is also used for paths in execution trees in Appendix A, because a path also corresponds to a set of executions.}
Chapter 3. Compositional Symbolic Execution and an Improvement

<table>
<thead>
<tr>
<th>$I$</th>
<th>$CP(I)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{read}$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = \text{True}$</td>
</tr>
<tr>
<td>$x := y$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = (x_1 = y_0)$</td>
</tr>
<tr>
<td>$x := \text{load } y$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = (x_1 = \text{mem}_0(y_0))$</td>
</tr>
<tr>
<td>store $y \rightarrow x$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = \left(\text{mem}_1 = \begin{cases} y_0 \rightarrow x_0 \ \text{Other} \rightarrow \text{mem}_0(\text{Other}) \end{cases}\right)$</td>
</tr>
<tr>
<td>$x := y \text{ op } z$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = (x_1 = y_0 \text{ op } z_0)$</td>
</tr>
<tr>
<td>control $^4$</td>
<td>$EC(I) = \text{True}$</td>
</tr>
<tr>
<td></td>
<td>$PC(I) = \text{True}$</td>
</tr>
</tbody>
</table>

This marks symbolic variables, but they are not necessary to be reflected in the condition pairs.

We assume three-address code, i.e., expressions are broken down. In this case, $y$ is a variable or a constant.

A particular kind of constraint (array select and store) available in most SMT solvers can be used to express memory operations.

control $^4$ represents a conditional. Its condition pair is constructed upon concatenation.

Figure 3.3: Generating a condition pair for each individual instruction.

with. For simplicity, let us assume a condition in a conditional always gets a Boolean variable to represent its evaluation outcome, in this case $b$. The resulting condition pair will look like one of these:

$$CP(I; R_1) = \left\{ \begin{array}{l} b_0 = 1 \\ b_0 = 1 \end{array} \right\} \otimes CP(R_1)$$

$$CP(I; R_2) = \left\{ \begin{array}{l} b_0 = 0 \\ b_0 = 0 \end{array} \right\} \otimes CP(R_2)$$

The $\otimes$ symbol is the concatenation operator, described in Section 3.3.3. Again, the versioning process will apply before further operations. In this case, there is not a preceding path before if, so $b$ gets a version 0. The other conditional branching instruction will follow a similar rule.

For handling larger parts of the code, we first have to define the concept of a code fragment:

**Definition 3** (Code fragment) A code fragment $f$ is a connected portion of a program $P$, which has a CFG $G_f$ that is a subgraph of the program’s CFG $G_p$, where $G_f$ has entry nodes $\text{Ent}_N_i$ ($i \in 1 \ldots n$) and exit nodes $\text{Ex}_N_j$ ($j \in 1 \ldots m$). Each path in this fragment—a subpath of the program—starts with a node in $\text{Ent}_N_i$ and ends with a node in $\text{Ex}_N_j$. A code fragment contains one or more executable instruction sequences, each of which implements one of the fragment’s paths.

An instruction sequence is denoted by $R$ as we have seen earlier. Note that our FGCSE tool Cirrus described in Section 3.5 assumes each code fragment has only one entry node, i.e., we have $n = 1$.

As an example, in Figure 3.4 is an if-then-else statement. The code fragment has two paths, one for each branch. The true branch has the instruction sequence $R_t = 1, 2, 3, 4, \ldots$, and the false branch $R_f = 1, 2, 6, 7, \ldots$ (we use the line numbers to refer to the instructions).
3.3. Fine-Grained Compositional Symbolic Execution

```
x := read
if (x > 0) {
  x := 42
  y := x
} else {
  x := 24
  y := x
}
```

**Figure 3.4:** Example code fragment.

The condition pairs of the two paths will be:

\[
EC(R_t) = (x_1 > 0)
\]

\[
PC(R_t) = (x_1 > 0 \land x_2 = 42 \land y_1 = x_2)
\]

\[
EC(R_f) = (x_1 \leq 0)
\]

\[
PC(R_f) = (x_1 \leq 0 \land x_2 = 24 \land y_1 = x_2)
\]

Note that the latest version of all variables before this code fragment is 0, meaning that the condition pairs’ variables’ base version is all 0. Each time a variable is assigned a value, its version is incremented by 1 (see Algorithm 3.1). The derivation of such condition pairs from individual condition pairs uses the concatenation operation.

### 3.3.3 Concatenation

Concatenation is an operation on two condition pairs, each of which could be concatenated condition pairs themselves. Systematic concatenation allows us to construct complex condition pairs from individual condition pairs, and eventually obtaining the path constraint for a given program path. The concatenation is denoted by:

\[
CP(R_1; R_2) = \left\{ \begin{array}{l}
EC(R_1; R_2) \\
PC(R_1; R_2)
\end{array} \right\} = CP(R_1) \otimes CP(R_2)
\]

\(R_1; R_2\) represents the sequence of \(R_1\) followed by \(R_2\).

The entry condition should represent the requirement on variables that the execution does not diverge from the path of \(R_1\) followed by the path of \(R_2\), which is

\[
EC(R_1) \land \forall V (PC(R_1) \Rightarrow EC(R_2))
\]

where \(V = \text{vars}(PC(R_1)) \setminus \text{vars}(EC(R_2))\) and \(\text{vars}(c)\) takes the set of variables of a logic formula \(c\). So \(V\) is a subset of the variables that are not symbolic inputs to \(R_2\). \(PC(R_1) \Rightarrow EC(R_2)\) is the weakest precondition such that given \(EC(R_1)\) and its implication, which is \(PC(R_1)\) (this means, e.g., suppose \(R_1\) is \(x := 1\) then \(EC(R_1) = \text{True}\) implies \(PC(R_1) = (x = 1)\) in regard to the execution of \(R_1\)), the execution will follow \(R_2\). Here, \(\forall V\) is to hide the variables in \(V\) and the implication of the execution of \(R_1\)\(^2\). Without hiding them, we have

\[
EC(R_1; R_2) = PC(R_1) \land EC(R_2)
\]

\(^2\)\(\forall V (PC(R_1) \Rightarrow EC(R_2)) \equiv (\exists V PC(R_1)) \Rightarrow EC(R_2)\)
Note that $PC(R_1) \Rightarrow EC(R_1)$ always holds. This is logically equivalent to the previous presentation regarding the symbolic inputs to $R_1; R_2$, therefore SE is still sound and complete. We use this form in Cirrus, because it can be deduced easily.

The postcondition after concatenation should represent the variable relations after executing $R_1, R_2$ in order, we have:

$$PC(R_1; R_2) = PC(R_1) \land PC(R_2)$$

We give some more examples of concatenation, highlighting the actually computed form of $CP(R_1; R_2)$ and the ideal form of it:

<table>
<thead>
<tr>
<th>Computed</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>${x = 1 }$ $\otimes$ ${x = 1}$</td>
<td>${x = 1}$ $\equiv$ ${x = 1}$</td>
</tr>
<tr>
<td>${y &lt; 0} \otimes {y &lt; 0}$</td>
<td>${x = 1 \land y &lt; 0} \equiv {y &lt; 0}$</td>
</tr>
<tr>
<td>${y &lt; x} \otimes {y &lt; x}$</td>
<td>${x = 1 \land y &lt; x} \equiv {y &lt; 1}$</td>
</tr>
</tbody>
</table>

We can claim the equivalence between the computed form and the ideal form in the above examples because the variable $x$ is not relevant to the conditions that matter (which is True for the first example, and which are on $y$ for the last two), and $x = 1$ is an inevitable consequence of executing $R_1$. There is no other possibility of $x$, so the entry conditions have the same truth table regarding the symbolic inputs. Using this computed form, we have longer constraints, but since the additional constraints are only intermediate, they are very easy for the solver to reduce and solve. Whichever form we use, $EC(R_1; R_2)$ still has the intended meaning, i.e., it is still the weakest condition for any inputs to the code fragment to execute a sequence of instruction without diverging from the path $R_1; R_2$. Here, divergence means the real execution (of a test from SE) is different from the execution predicted by static analysis.

Similarly, before concatenation, variables need to be aligned according to the direction of reasoning to preserve consistency. Specifically, in forward reasoning, variables in $CP(R_2)$ is aligned with respect to $PC(R_1)$; in backward reasoning, variables in $CP(R_1)$ is aligned with respect to $PC(R_2)$. Variable versioning can be seen in Algorithm 3.1.

Concatenation has another useful property: it preserves the variable names and relations in between for the old constraints. This makes it friendly to incremental solving. The newly concatenated part of the entry condition, in this case, $(PC(R_1) \setminus EC(R_1)) \land EC(R_2)$ (the set of constraints in $PC(R_1)$ minus that of $EC(R_1)$ plus that of $EC(R_2)$) can be simply pushed to the incremental solver after proper versioning. Integration of incremental solving into CSE is discussed in Chapter 5.

### 3.3.4 De-Concatenation

De-concatenation is the inverse operation of concatenation. We use de-concatenation to remove newly concatenated condition pairs from a bigger condition pair so that the search can backtrack to a previous branching point, and so we do not have to cache summaries of all subpaths.
3.3. Fine-Grained Compositional Symbolic Execution

The de-concatenation process of $CP(R_1; R_2)$ process is simply cutting off $(PC(R_1) \setminus EC(R_1)) \land EC(R_2)$ from $EC(R_1; R_2)$ and $PC(R_2)$ from $PC(R_1; R_2)$ which is easily achieved by indexing the origin of each component of the condition pair.

Concatenation and de-concatenation on a condition pair is space-conservative, and useful in incremental solving techniques, as mentioned before.

### 3.3.5 Composing Path Constraints

A path constraint is simply the concatenation of the condition pairs from all code fragments in that path. Essentially our CSE approach is a process of finding the possible sequences of instructions, concatenating the condition pairs to achieve the path constraint, and solving this path constraint.

Figure 3.5 shows how two path constraints can be built for the code fragment in Figure 3.4.

Each step contains two operations: version number updating and concatenation, using the rules previously defined. The conditional statement produces an indefinite condition pair. Symbolic execution marks its location as a backtracking point. When going through this point, the symbolic execution will generate a condition pair in accordance with the selected branch.

The example assumes summaries are at the instruction level. If there are summaries of larger blocks available (we record mappings to indicate from where to where summaries are available) in the current path, symbolic execution will use the summaries instead of instruction condition pairs, skipping what is between the starting point and the end point(s) of the summaries. This is how the summarisation mode of symbolic execution avoids redundant solver calls. Again, summaries are essentially cached condition pairs.

### 3.3.6 The Loss of Context

A drawback of bottom-up CSE is the loss of context. Using the same example as Figure 3.2, suppose call 2 is always unsatisfiable for some reason (maybe there are other conditions constraining the array located at $ap$ outside the loop), of which summarisation would not know because summarisation is done locally. Every time the search enters the loop body,
three summaries will be used while only one of them is of value (corresponding to call A),
the other two are infeasible. In Figure 3.6, we denote the calls that will return “unsatisfiable”
by orange text (call 2 to 6, B, and C). In this case, CSE makes two such calls (call B and C)
while Traditional SE discover this using a single call (call 2).

This is an inherent drawback of using over-approximation, which leads to the discussion
in the next section. We introduce another solution to this using assumptions in Section 5.2.1.

3.4 The Problems of Using Summaries

Summaries are the building blocks of path constraints in compositional symbolic execution.
Existing (function-level) summarisation techniques (see CSE in Section 2.3.4) choose func-
tions as the targets of summarisation, which are affected by design choices made by a devel-
oper. In other words, treating function as the smallest unit of summarisation and not being
selective about the functions to summarise have not fully realise CSE’s potential. Partial
loop summarisation [65], on the other hand, generates summaries based on loop invariant
detection which is sound on certain types of loops, and is still not targeting and being se-
lective on arbitrary code fragments. For top-down summarisation approaches, a greedy
summarisation strategy might be feasible, but for bottom-up summarisation which has over-
approximation, this greatly affects the efficiency.

Our target of summarisation can range from one single instruction to multiple basic
blocks, including entire program paths. This calls for a strategy of selecting targets. We
describe three strategies of choosing summarisation targets later in this section. Before that,
we discuss the principles of identifying desirable blocks, which also clarifies why summar-
sation on a large code fragment; e.g., at a function level, is not always desirable.

3.4.1 Program Characteristics and Summarisation

The major cost of symbolic execution comes from constraint solving. One of the benefits of
using summaries is the elimination of redundant solver calls. However, summaries come
with certain costs, including the sacrifice of some of the decision making points and the cost
of computing and storing summaries themselves.

In the example in Figure 3.6, the decision making points in summarisation mode are post-
poned to the end of the subpaths. More specifically, this inefficiency exposes two problems
of summarisation:

1. Latency in decision making: if we directly make call 2 (not using summaries), the
unsatisfiability would have been checked earlier.
3.4. The Problems of Using Summaries

2. Sensibility to target’s dependency: the perfect code fragment to be summarised should not have too many variables constrained elsewhere.

For this reason, long summarisation, such as function-level summarisation, are more risky than the shorter, code-fragment summarisation: there could be more infeasible summaries.

We now discuss features of desirable candidate code fragments for summarisation.

Number of Infeasible Subpaths

A subpath is infeasible when its path constraint is unsatisfiable. A naive path search will try such a path as long as it is topologically possible until the constraint solver stops it from doing so. In other words, an infeasible subpath will consume solver calls. Eliminating these from consideration with cheaper methods can reduce the number of calls sent to a solver.

Fine-grained summarisation can identify a subset of the infeasible subpaths in the summarised code. When a subpath is shown infeasible, i.e., we have generated an unsatisfiable entry condition for this subpath, its condition pair will be not be cached, so throughout the remaining symbolic execution, the infeasible subpath will no longer be considered.

In summarisation, the number of infeasible subpaths detected in code fragments indicates how promising the results summaries are in increasing performance. This can be obtained in summarisation time. The remaining infeasible subpaths that are not detected, i.e., only known with a certain context, are potential “threats” to summarisation. We discuss this problem with the next code characteristic.

Independent Code

A code fragment should have lower priority in summarisation if it has a strong dependency on other parts of the program. Strong dependency means that the variables in this code fragment are partly constrained elsewhere. In this case, the summarisation procedure is unlikely to discover infeasible subpaths due to the lack of constraint information. More importantly, the symbolic execution runs a higher risk of generating long summaries that might be infeasible.

We explain in more details. Our summarisation is a local analysis on a target code fragment, and does not take into consideration any surrounding blocks, so as to allow the resulting summaries to fit into any context. Therefore, a summary might have weak constraints that are only known to be unsatisfiable when provided the surrounding context. This means the summarisation can produce more summaries (for each potential subpaths) than there actually needs to be. For example, a variable may be initialised to a particular value when declared. If a subsequent block that does not has the initialisation information is summarised, the summary will over-approximate the set of possible executions. The infeasible summaries could be added during path search and remain undetected until the initialisation information is given. The longer the summarisation target is, the more such summaries there will be and the more constraint solving time they will take.

So another intuition for summarisation would be to avoid breaking a functional unit apart for summarisation; instead, independent code fragments should be summarised. For example, loops in drivers listening to user actions and performing independent tasks in each of their iterations [18] are good targets for summarisation. A function that does not depend on global variables and does not produce side effects is better than those that do.
In Section 5.2.1, we introduce a method to correct the lack of context based on a constraint solving feature assumption checking.

Minimum Number of Calls Ratio

Combining the two concerns, we summarise a code fragment when we can expect to discover more infeasible subpaths during summarisation and fewer infeasible summaries after summarisation. If the code fragment is rather independent, that is, the satisfiability of the subpaths is not depending on information given outside the fragment, then summarisation is preferable. The more frequently it is visited, the more effective the summaries will be.

We use the number of resulting summaries, normalised by the number of solver calls to produce them, which in a way reflects the relative number of infeasible subpaths found in summarisation, to indicate how promising the summarisation result is. The ratio

\[ \text{CallRatio} = \frac{\text{Number of all resulting summaries}}{\text{Number of calls to summarise the code fragment}} \]

expresses the efficiency of the summarisation on a code fragment. The efficiency is higher when \( \text{CallRatio} \) is smaller. In Figure 3.2, during summarisation (local symbolic execution in the loop body), 6 calls are made in advance to create 3 subpath summaries for symbolic execution. The summarisation would expect the symbolic execution to make use of those 3 subpath summaries instead of 6 calls every time in the search, hence \( \text{CallRatio} = \frac{1}{2} \). Actual efficiency could be different due to a possible dependency problem, as we explain using Figure 3.6, in which case symbolic execution without summarisation can use only 2 calls to explore the code instead of the predicted 6.

Nevertheless, we can use this ratio as an estimate of SE’s potential efficiency gain via summarisation. \( \text{CallRatio} \) reflects whether a summarisation strategy has considered the program characteristics. If a code fragment has a large portion of its paths infeasible and they can be detected locally, \( \text{CallRatio} \) will be small. Otherwise, it can be close to 1, meaning summaries from this fragment are not so valuable. Similarly, if the code fragment is dependent on the context then \( \text{CallRatio} \) will also be smaller compared to when it is not. We set a threshold on \( \text{CallRatio} \) and check against it each time we finish summarisation on one code fragment to prevent weak or useless results (implementation explained in Section 3.5.1). A small \( \text{CallRatio} \) can guarantee increase in symbolic execution speed in most cases, witness the results in Section 3.6.

3.4.2 Strategies for Summarisation

In this section, we present three heuristics for finding good summarisation targets. The three heuristics attempt to identify repeatedly appearing code fragments in SE, while differing in the degree of conservatism. Recall that summarising a larger code fragment tends to yield more, and longer, summaries because of the larger size of execution subtree, thus is considered having more potential in increasing efficiency but more risky.

Loop-body SUMmarisation (LSUM) In this summarisation mode, any loop body including the loop condition is summarised. If there is a nested inner loop, this inner loop will be summarised first, and then the summaries of the nested loop will be used to construct the enclosing loop’s summaries. This strategy effectively unrolls the inner loops (using their
summarisation), and a depth limit can apply to this process to avoid infinite unrolling. Each summary in this mode represents the subpath constraint of each subpath going through the loop body in one loop iteration.

This heuristic suits a program with loops whose iterations are rather independent. Loops manipulating arrays or lists, with each iteration not interfering with others, are ideal targets. For a program with a large nested loop, this mode may be prohibitively expensive, owing to possible path explosion inside the loop.

**Loop-body-Acyclic SUMmarisation (LASUM)** In this mode, similarly, loop bodies will be summarised. The difference with LSUM is, inner nested loops break the outer loop’s summarisation. As above, each inner loop is summarised first, and then, the code blocks before and after the inner loop, but inside the outer loop, are summarised separately.

This heuristic also works well if there is no dependency among the iterations. For nested loops it mitigates the path explosion problem seen with LSUM, but is more conservative (produces fewer summaries), lowering the potential to reduce the number of solver calls.

**Small-step SUMmarisation (SSUM)** In this mode, consecutive code blocks uninterrupted by loops are summarised, e.g., consecutive if-then-else statements that do not contain a loop. Nested code blocks are summarised separately, e.g., the true branch of a if-then-else statement is summarised first before the code block of the statement itself. Additionally, the number of if-then-else statements summarised each time can be limited by a threshold value.

This heuristic produces smaller summaries than the previous two heuristics and is less expensive. Even when the summaries cannot reduce the symbolic execution’s solver call frequency as significantly as the other strategies, there is minimal additional cost.

Summarisation is performed before symbolic execution. We detect conditional statements and their boundaries, e.g., loop headers and exits through the CFG. All summaries are cached, as well as their beginning and ending locations.

The summarisation strategies are shown in Algorithm 3.2. We highlight the statements used exclusively by different strategies using different colours: orange for LSUM; blue for LASUM; green for SSUM. The **SUMMARISE** procedure performs SE on a code fragment \( f \), and eventually storing the summaries of \( f \) in **SummaryMap**. This storage lets SE index the summaries of a code fragment through the beginning or ending instructions. **NEWSTATE** initialises a symbolic state, similar to a traditional symbolic state, but with special fields useful in FGCSE, e.g., the path constraints are stored as condition pairs. **PROPAGATE** lets a state visit a summary, essentially symbolically execute the path summarised by this summary. Note that for simplicity of presentation Algorithm 3.2 does not strictly consider all the cases of nested code structure. For example, an inner loop does not necessary interrupt all paths of the outer loop; the two branches of an if-then-else statement could also be interrupted by a loop. \( CP_t, CP_f \) here represent the condition pairs of the branching instruction upon concatenation with the true and false branches respectively (see Section 3.3.2).

Figure 3.7 provides a general outline of the three strategies on a schematic program. One can see that the difference between the three heuristics is how they restrict the area of summarisation. The order of strictness is generally SSUM > LASUM > LSUM.

The features and expected efficiency of these strategies, relative to each other, are stated qualitatively in Table 3.1. Under any summarisation mode (X-SUM), the innermost scope will be summarised first using X-SUM. The larger scope is then X-summarised using the
Algorithm 3.2 Summarisation and the strategies.

**procedure** SUMMARISE\(f, \text{SummaryMap}\) \(\triangleright\) Add \(f\)'s summaries to \(\text{SummaryMap}\)
\[
\text{State} := \text{NEWSTATE}(f.\text{FIRSTINSTR}) \\
\text{States} := \{\text{State}\} \\
\text{while States} \neq \emptyset \text{ do} \\
\quad \text{State} := \text{States}.\text{POP} \\
\quad \text{if State.Instr is the header of a loop Loop then} \\
\quad \quad \text{SUMMARISE}(\text{State.HEAD}, \text{SummaryMap}) \triangleright \text{LASUM exclusive} \\
\quad \quad \text{SUMMARISE}(\text{Loop}, \text{SummaryMap}) \triangleright \text{SSUM exclusive} \\
\quad \quad \text{for each Summary} \in \text{SummaryMap.FIND}(\text{Loop.FIRSTINSTR}) \text{ do} \\
\quad \quad \quad \text{States}.\text{PUSH}(\text{PROPAGATE}(\text{State, Summary})) \triangleright \text{LSUM exclusive} \\
\quad \quad \text{end for} \\
\quad \text{else if State.Instr is an if-then-else statement ITE then} \triangleright \text{SSUM exclusive} \\
\quad \quad \quad \text{if State.ITECount} \geq \text{Step length} \text{ then} \\
\quad \quad \quad \quad \text{SUMMARISE}(\text{ITE.\text{TrueBranch}}, \text{SummaryMap}) \triangleright \text{Step length is predefined in SSUM} \\
\quad \quad \text{SUMMARISE}(\text{ITE.\text{FalseBranch}}, \text{SummaryMap}) \\
\quad \quad \text{for each Summary} \in \text{SummaryMap.FIND}(\text{ITE.\text{TrueBranch}.FIRSTINSTR}) \text{ do} \\
\quad \quad \quad \text{States}.\text{PUSH}(\text{PROPAGATE}(\text{State, ITE.CP}_f \otimes \text{Summary})) \\
\quad \quad \text{end for} \\
\quad \quad \text{for each Summary} \in \text{SummaryMap.FIND}(\text{ITE.\text{FalseBranch}.FIRSTINSTR}) \text{ do} \\
\quad \quad \quad \text{States}.\text{PUSH}(\text{PROPAGATE}(\text{State, ITE.CP}_f \otimes \text{Summary})) \\
\quad \quad \text{end for} \\
\quad \text{end if} \\
\quad \text{else if State.Instr} = 0 \text{ then} \triangleright \text{Reached the end of } f \\
\quad \quad \text{SUMMARISE}(\text{State.HEAD}, \text{State.TAIL}, \text{State.CP}) \\
\quad \text{else} \\
\quad \quad \text{NORMALISE}(\text{State}) \\
\quad \text{end if} \\
\text{end if} \\
\text{end while} \\
\text{end procedure} \\

**procedure** NEWSTATE(Instr) \\
\{ Head / Tail \mapsto 0, Instr \mapsto Instr, CP \mapsto \{EC / PC \mapsto \text{True}\}, ITECount \mapsto 0\} \\
\text{end procedure} \\

**procedure** PROPAGATE(STATE, Summary) \\
\text{State.CP} := \text{State.CP} \otimes \text{Summary} \\
\text{if State.HEAD} = 0 \text{ then} \\
\quad \text{State.HEAD} := \text{Summary.HEAD} \\
\text{end if} \\
\text{State.TAIL} := \text{Summary.TAIL} \\
\text{State.INSTR} := \text{Summary.INSTR} + 1 \\
\text{Return State} \\
\text{end procedure}
3.5 Implementation

Our FGCSE tool Cirrus is an experimental prototype, not necessarily intended to scale to large systems. Nonetheless, it is sophisticated enough to experiment with summarisation ideas on non-trivial programs, as described in the evaluation (Section 3.6). Cirrus accepts programs written in a basic C-like language with assignment, branching, and looping. The types supported include integers, floats, and arrays of these. Strings can be used but not in constraints. This is compiled to an IR-like language, upon which we apply our algorithms. We use Z3 [47] as the underlying constraint solver, to both produce summaries and perform the symbolic execution. Incremental solving is turned on, though we have a more mature implementation Cirrocumulus with this particular feature introduced in Chapter 5 and we discuss this until then.
3.5.1 Summarisation and the \textit{CallRatio} Threshold

Summarisation is performed before symbolic execution. It is essentially a small symbolic execution done on the code fragments automatically selected by the summarisation strategy. The results are the corresponding summaries for each of these code fragments in the form of condition pairs.

After that the summarisation needs to go through a screening process for the reason stated in Section 3.4.1. A summarised code fragment needs to satisfy

\begin{equation}
\text{CallRatio} < \text{Threshold} \quad (\text{Threshold} \in (0, 1))
\end{equation}

for its summaries to be considered useful. Otherwise, we discard the summaries and choose to use the naive path search later in the symbolic execution, because a summarisation above the threshold gives the smallest boost up in the speed while having a higher risk of containing dependent constraints. The smaller the Threshold is, the fewer summaries we will have. This turns down the intensity of summarisation to gain robustness.

After these code fragments are summarised and screened, their summaries can be reused. Each time a state visits such a code fragment, it will look up the associated summaries. Note that we do not merge the path summaries, as discussed in Section 3.1.1.

Our summaries can be potentially reused among different SE runs, assuming the same summarisation strategy. In Cirrus, however, we compute summaries before each SE run.

3.5.2 The Backward Reasoning Method

The symbolic execution performs path search as follows. The starting point is the target instruction. In our experiment, the target instruction is the unique program termination point (so we are doing exhaustive exploration). First, it finds all instructions that directly precede the current instruction in the CFG, and chooses one of them to proceed according to the specified search strategy (discussed below). Then it concatenates the current condition pair and the newly found condition pair. The entry condition in the concatenated condition pair will be pushed to the solver’s stack and checked. Any summary found in the middle of this procedure will be used instead of individual instructions’ condition pairs, and the search will skip to the beginning instruction of the selected summary. If a program entry is found, then the search has ended and it will backtrack until there is no more choice.

The summarisation allows more search heuristics because of enlarged search steps. At each branching point where summaries are available, different priority schemes can apply. Summaries gather information from a larger range of code, which encourages the decision maker to make a wiser choice while doing targeted symbolic execution. Since in BSE the target is always the program entry (in our case there is only one entry with a certain instruction number 0), we have only implemented one search strategy—greedy search. However, when doing forward targeted symbolic execution, we believe there could be more possibilities—this will be one direction of our future work.

At each branching point, the greedy search strategy will choose a previously-unchosen preceding instruction/summary that has the smallest instruction number/starting point number, in order to get closer to the program entry. To avoid getting stuck in a search, each path considered has a parameterisable maximum depth.
3.6 Preliminary Evaluation

In this section, we present an evaluation of Cirrus. We compare the three strategies, LSUM, LASUM, and SSUM, alongside an approach that uses no summarisation, which makes it equivalent to standard BSE. We measure the execution time and the number of calls to the constraint solver.

3.6.1 Experiment Design

Cirrus accepts a simple subset of C, providing us with a rather limited pool of experimental programs. Nonetheless it is sufficient for the observation of Cirrus’s performance. We take eight small but non-trivial programs, summarised in Table 3.2. We create a number of variants for each of them, which differ mainly in the loop boundaries. The “# Instrs” column shows the number of instructions of the smallest variant of each program. The “# Conds” column shows the number of conditionals, some of which are loops, shown under “# Loops”. Although these programs are small in terms of instructions, they still have an explosively large number of paths; in some cases, infinite.

binary_search and bubble_sort are the well-known searching and sorting algorithms. cba_example is the example program taken from Figure 1 (a) of Obdržálek and Trtík [96]. kmp is a Knuth-Morris-Pratt algorithm implementation.

due_date computes a due date from a start date and a period (in number of days).

matrices is a matrix multiplication program. These two programs contain multiplication and/or division operations, which are comparably hard for the solver, and this is reflected later in the results (see Table 3.3). tic_tac_toe and wumpus are programs of the games Tic-Tac-Toe and Hunt the Wumpus respectively.

We create five variants for every program that has only bounded loops, i.e., program binary_search, bubble_sort, cba_example, kmp, and matrices. We create them by varying a factor in each program: the factor for binary_search, bubble_sort, and cba_example is the length of the array to be searched, sorted, or matched; the factor for kmp is a pair \((x, y)\), where \(x\) stands for the length of the array to be searched and \(y\) stands for the length of the pattern; the factor for matrices is a triple \((x, y, z)\) which effectively means the program multiplies a matrix of size \(x \times y\) at maximum and a matrix of size \(y \times z\) at maximum. For each program with an unbounded loop, we set a maximum depth to limit the search.

We varied one independent variable in our experiment: the summarisation mechanism used. This implies four different values:

<table>
<thead>
<tr>
<th>Program</th>
<th># Instrs</th>
<th># Conds</th>
<th># Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_search</td>
<td>≥121</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>bubble_sort</td>
<td>≥127</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>cba_example</td>
<td>≥122</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>due_date</td>
<td>433</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>kmp</td>
<td>≥261</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>matrices</td>
<td>231</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>tic_tac_toe</td>
<td>952</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>wumpus</td>
<td>1 824</td>
<td>270</td>
<td>10</td>
</tr>
</tbody>
</table>
1. SE: only individual instructions’ condition pairs are cached, effectively no summarisation.

2. LSUM: the simple loop summarisation approach.

3. LASUM: the acyclic loop summarisation approach.

4. SSUM: the small-step summarisation approach, with the length limit set to 3.

The threshold of \( \text{CallRatio} \) is set to 0.8 for all the summarisation modes. A limitation of our work is that we cannot provide a guideline in choosing the step length of SSUM or the value of \( \text{CallRatio} \). So these numbers are chosen from our experience.

We measured the execution time for each approach on each program, and also the number of calls made to the underlying constraint solver, including those calls used to produce the summaries. Note that summaries are not shared between executions and always (re)computed at the beginning of each of them. Our evaluations are run on a laptop with Intel Core i7 CPU @ 2.20 GHz \( \times 8 \), with 8 GB of memory. Constraint solving is done with Z3 4.3.2.

### 3.6.2 Results

Table 3.3 shows the time consumption and Table 3.4 shows the number of solver calls of the complete symbolic execution on the test programs. In Table 3.4, we show the numbers of total solver calls for all modes, while also explicitly showing the calls made during summarisation in the parentheses. The threshold for \( \text{CallRatio} \) is set to 0.8 as explained in the experiment setup. Note that the \( \text{CallRatio} \) value in Table 3.4 is the overall ratio of all code fragments that have been summarised and whose summaries have past the threshold, not individual ones. It serves as an indicator of how the summarisation works but not necessarily reflects the actual effectiveness of the summaries. A summary is shown at the end of each table, including the total and the relative numbers. The calculation methods for the relative numbers are explained after each table. For those cases where no suitable summary is found, \( \text{CallRatio} \) is taken as 1 for the calculation of the average ratios.

### 3.6.3 Discussion

The results demonstrate the path explosion problem, showing that programs are non-trivial, albeit small. Additionally, they show the strong correlation between the symbolic execution time and number of solver calls, which indicates reducing the number of solver calls (through FGCSE) has promise in improving efficiency.

In summary, the results demonstrate that fine-grained summarisation can improve symbolic execution efficiency, mostly by reducing the number of calls sent to the constraint solver. In some cases, the change is marginal, but in others, such as \texttt{kmp} in LSUM and \texttt{tic_tac_toe} in all modes, execution time is considerably reduced. Of all the summarisation strategies, LSUM achieved the best overall result (20% time reduction).

However, in some cases, summarisation non-trivially increases the execution time. This happens more often in LSUM mode (\texttt{bubble_sort} and \texttt{tic_tac_toe}) than the other two modes. LSUM mode also tends to introduce a larger summarisation overhead. These leads to our second conclusion: a summarisation strategy that unrolls loops is more risky to apply than one that do not in terms of time efficiency.
Table 3.3: Experiment results on Cirrus. We show the symbolic execution time taken in the experiment.

<table>
<thead>
<tr>
<th>Program</th>
<th>Factor/depth¹</th>
<th>SE</th>
<th>LSUM</th>
<th>LASUM</th>
<th>SSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_search</td>
<td>5</td>
<td>1.3</td>
<td>1.2</td>
<td>1.2</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10.3</td>
<td>10.1</td>
<td>10.2</td>
<td>10.3</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>49.7</td>
<td>46.0</td>
<td>45.8</td>
<td>49.7</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>208.3</td>
<td>196.9</td>
<td>197.4</td>
<td>205.2</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>653.2</td>
<td>649.0</td>
<td>656.1</td>
<td>731.5</td>
</tr>
<tr>
<td>bubble_sort</td>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.4</td>
<td>0.4</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.3</td>
<td>3.8</td>
<td>2.6</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>36.3</td>
<td>50.5</td>
<td>25.1</td>
<td>36.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>407.8</td>
<td>807.1</td>
<td>290.4</td>
<td>402.1</td>
</tr>
<tr>
<td>cba_example</td>
<td>2</td>
<td>0.7</td>
<td>0.6</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5.5</td>
<td>4.5</td>
<td>4.5</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39.8</td>
<td>36.1</td>
<td>35.9</td>
<td>39.7</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>305.9</td>
<td>283.2</td>
<td>283.1</td>
<td>305.1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2.2k</td>
<td>2.0k</td>
<td>2.0k</td>
<td>2.1k</td>
</tr>
<tr>
<td>due_date</td>
<td>20</td>
<td>762.8</td>
<td>291.6</td>
<td>344.0</td>
<td>314.3</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>9.2k</td>
<td>773.4</td>
<td>920.3</td>
<td>1.6k</td>
</tr>
<tr>
<td>kmp</td>
<td>32</td>
<td>18.2</td>
<td>6.8</td>
<td>18.2</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>39.4</td>
<td>16.6</td>
<td>39.6</td>
<td>39.6</td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>96.5</td>
<td>39.2</td>
<td>97.7</td>
<td>96.5</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>288.5</td>
<td>127.7</td>
<td>289.7</td>
<td>287.4</td>
</tr>
<tr>
<td></td>
<td>43</td>
<td>840.4</td>
<td>397.3</td>
<td>825.0</td>
<td>829.4</td>
</tr>
<tr>
<td>matrices</td>
<td>211</td>
<td>1.4</td>
<td>0.5</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>221</td>
<td>20.9</td>
<td>8.8</td>
<td>21.7</td>
<td>21.5</td>
</tr>
<tr>
<td></td>
<td>222</td>
<td>173.7</td>
<td>62.5</td>
<td>172.0</td>
<td>168.6</td>
</tr>
<tr>
<td></td>
<td>321</td>
<td>213.8</td>
<td>16.7</td>
<td>214.4</td>
<td>211.0</td>
</tr>
<tr>
<td></td>
<td>331</td>
<td>1.6k</td>
<td>171.3</td>
<td>1.5k</td>
<td>1.5k</td>
</tr>
<tr>
<td>tic_tac_toe</td>
<td>25</td>
<td>8.5</td>
<td>19.0</td>
<td>18.8</td>
<td>5.6</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>32.6</td>
<td>40.4</td>
<td>40.4</td>
<td>33.1</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>72.3</td>
<td>42.9</td>
<td>43.1</td>
<td>62.1</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>189.4</td>
<td>99.4</td>
<td>99.3</td>
<td>183.2</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>810.6</td>
<td>140.3</td>
<td>139.0</td>
<td>662.2</td>
</tr>
<tr>
<td>wumpus</td>
<td>25</td>
<td>3.4</td>
<td>4.2</td>
<td>3.2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>11.5</td>
<td>13.6</td>
<td>6.0</td>
<td>12.6</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>30.3</td>
<td>26.5</td>
<td>16.4</td>
<td>31.1</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>71.5</td>
<td>53.4</td>
<td>43.9</td>
<td>73.1</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>185.0</td>
<td>118.6</td>
<td>145.9</td>
<td>195.3</td>
</tr>
</tbody>
</table>

| Total        | -             | 18.6k| 6.6k  | 8.6k  | 10.2k |
| Relative²    | -             | -    | 80%   | 89%   | 95%   |

¹ For due_date, tic_tac_toe, and wumpus this is the depth; for others, this is the factor, e.g., matrices 211 means \((x, y, z) = (2, 1, 1)\)—the multiplication of a matrix of size \(2 \times 1\) at maximum and a matrix of size \(1 \times 1\) at maximum.

² Let the result of a strategy (X-SUM) be \(x\) and the result of SE be \(y\) in each row. This is taken as the average of \(2x/(x + y)\).
Table 3.4: Experiment results on Cirrus. We show the number of solver calls in the experiment. The numbers in parentheses are the summarisation overhead.

<table>
<thead>
<tr>
<th>Program</th>
<th>Factor/depth(^1)</th>
<th>SE</th>
<th>LSUM</th>
<th>LASUM</th>
<th>SSUM</th>
<th># Solver calls</th>
<th>CallRatio (\times 0.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Binary_search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>254</td>
<td>276</td>
<td>276</td>
<td>254</td>
<td>(0)</td>
<td>3/4</td>
<td>3/4</td>
</tr>
<tr>
<td>10</td>
<td>689</td>
<td>726</td>
<td>726</td>
<td>689</td>
<td>(0)</td>
<td>3/4</td>
<td>3/4</td>
</tr>
<tr>
<td>15</td>
<td>1.3k</td>
<td>1.4k</td>
<td>1.4k</td>
<td>1.3k</td>
<td>(0)</td>
<td>3/4</td>
<td>3/4</td>
</tr>
<tr>
<td>20</td>
<td>2.2k</td>
<td>2.2k</td>
<td>2.2k</td>
<td>2.2k</td>
<td>(0)</td>
<td>3/4</td>
<td>3/4</td>
</tr>
<tr>
<td>25</td>
<td>3.2k</td>
<td>3.3k</td>
<td>3.3k</td>
<td>3.2k</td>
<td>(0)</td>
<td>3/4</td>
<td>3/4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>bubble_sort</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>27</td>
<td>23</td>
<td>27</td>
<td>27</td>
<td>(0)</td>
<td>4/13</td>
<td>2/3</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td>89</td>
<td>87</td>
<td>103</td>
<td>(0)</td>
<td>8/25</td>
<td>2/3</td>
</tr>
<tr>
<td>4</td>
<td>463</td>
<td>545</td>
<td>369</td>
<td>463</td>
<td>(0)</td>
<td>16/49</td>
<td>2/3</td>
</tr>
<tr>
<td>5</td>
<td>2.5k</td>
<td>4.8k</td>
<td>2.0k</td>
<td>2.5k</td>
<td>(0)</td>
<td>32/97</td>
<td>2/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>cba_example</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>379</td>
<td>316</td>
<td>316</td>
<td>379</td>
<td>(0)</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>3</td>
<td>1.6k</td>
<td>1.3</td>
<td>1.7</td>
<td>1.6k</td>
<td>(0)</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>4</td>
<td>6.8k</td>
<td>5.7k</td>
<td>5.7k</td>
<td>6.8k</td>
<td>(0)</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td>5</td>
<td>29.0k</td>
<td>24.8k</td>
<td>24.8k</td>
<td>29.0k</td>
<td>(0)</td>
<td>4/6</td>
<td>4/6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>due_date</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>218</td>
<td>173</td>
<td>173</td>
<td>222</td>
<td>(13)</td>
<td>12/21</td>
<td>8/12</td>
</tr>
<tr>
<td>25</td>
<td>603</td>
<td>402</td>
<td>402</td>
<td>555</td>
<td>(13)</td>
<td>12/21</td>
<td>8/12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>knp</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.8k</td>
<td>1.2</td>
<td>1.9k</td>
<td>1.8k</td>
<td>(0)</td>
<td>8/20</td>
<td>0/0</td>
</tr>
<tr>
<td>42</td>
<td>3.8k</td>
<td>2.3k</td>
<td>3.8k</td>
<td>3.8k</td>
<td>(0)</td>
<td>8/20</td>
<td>0/0</td>
</tr>
<tr>
<td>52</td>
<td>7.3k</td>
<td>4.5k</td>
<td>7.3k</td>
<td>7.3k</td>
<td>(0)</td>
<td>8/20</td>
<td>0/0</td>
</tr>
<tr>
<td>62</td>
<td>15.2k</td>
<td>10.4k</td>
<td>15.2k</td>
<td>15.2k</td>
<td>(0)</td>
<td>11/26</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>matrices</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>91</td>
<td>47</td>
<td>96</td>
<td>91</td>
<td>(0)</td>
<td>5/15</td>
<td>0/0</td>
</tr>
<tr>
<td>221</td>
<td>187</td>
<td>94</td>
<td>192</td>
<td>187</td>
<td>(0)</td>
<td>8/45</td>
<td>0/0</td>
</tr>
<tr>
<td>222</td>
<td>383</td>
<td>209</td>
<td>388</td>
<td>383</td>
<td>(0)</td>
<td>11/85</td>
<td>0/0</td>
</tr>
<tr>
<td>321</td>
<td>243</td>
<td>114</td>
<td>248</td>
<td>243</td>
<td>(0)</td>
<td>8/45</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>tic_tac_toe</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>592</td>
<td>882</td>
<td>882</td>
<td>448</td>
<td>(36)</td>
<td>198/774</td>
<td>1635/1635</td>
</tr>
<tr>
<td>30</td>
<td>2.1k</td>
<td>1.1k</td>
<td>1.1k</td>
<td>1.9k</td>
<td>(36)</td>
<td>198/774</td>
<td>1635/1635</td>
</tr>
<tr>
<td>35</td>
<td>5.4k</td>
<td>1.1k</td>
<td>1.1k</td>
<td>4.0k</td>
<td>(36)</td>
<td>198/774</td>
<td>1635/1635</td>
</tr>
<tr>
<td>40</td>
<td>15.0k</td>
<td>5.4k</td>
<td>5.4k</td>
<td>13.7k</td>
<td>(36)</td>
<td>198/774</td>
<td>1635/1635</td>
</tr>
<tr>
<td>45</td>
<td>39.5k</td>
<td>5.4k</td>
<td>5.4k</td>
<td>31.5k</td>
<td>(36)</td>
<td>198/774</td>
<td>1635/1635</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>wumpus</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2.8k</td>
<td>5.1k</td>
<td>2.2k</td>
<td>2.0k</td>
<td>(486)</td>
<td>317/341</td>
<td>502/3485</td>
</tr>
<tr>
<td>30</td>
<td>13.0k</td>
<td>15.1k</td>
<td>5.2k</td>
<td>2.3k</td>
<td>(511)</td>
<td>403/105k</td>
<td>343/212k</td>
</tr>
<tr>
<td>35</td>
<td>31.3k</td>
<td>28.7k</td>
<td>15.6k</td>
<td>31.8k</td>
<td>(511)</td>
<td>479/27.6k</td>
<td>344/42.2k</td>
</tr>
<tr>
<td>40</td>
<td>64.0k</td>
<td>53.0k</td>
<td>41.2k</td>
<td>64.5k</td>
<td>(511)</td>
<td>618/53.0k</td>
<td>345/92.2k</td>
</tr>
<tr>
<td>45</td>
<td>113.8k</td>
<td>84.5k</td>
<td>90.4k</td>
<td>114.3k</td>
<td>(511)</td>
<td>841/84.4k</td>
<td>347/20.4k</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>546.5k</td>
<td>454k</td>
<td>454k</td>
<td>394k</td>
<td>(189.1k)</td>
<td>3.9k/183.7k</td>
<td>2.9k/41.1k</td>
</tr>
</tbody>
</table>

\(^1\) Let the result of a strategy (X-SUM) be \(x\) and the result of SE be \(y\) in each row. This is taken as the average of \(x/y\).

\(^2\) Let the number of calls made during summarisation be \(x\) and the number of calls taken by X-SUM be \(y\) in each row. This is taken as the average of \(x/y\).

\(^3\) This is the average of the result in each row.
3.7 Conclusion and Future Work

Additionally, there is an interesting case where summarisation uses most of the solver calls (wumpus in LSUM). This is due to its unbounded nested loops, which LSUM keeps unrolling to generate more summaries. We suggest the use of the more conservative LASUM mode in the presence of unbounded loops.

Comparing SSUM with the other two modes, we can see that it is less often active and overall less efficient. This is because the summarisation process discards the summaries of a code fragment when it cannot pass the threshold $CallRatio$ and since SSUM works on consecutive ITEs, this happens more often. Note that most of such constructs never have more than one ITE in our experiment, with the exception of the last two programs. Despite this limitation of the program pool, from this, we conclude that fine-grained summarisation benefits SE the most through loop summarisation.

As a side note, there are some unusual results for tic_tac_toe. In LSUM/LASUM, the number of solver calls does not increase when the depth increases from 30 to 35 (similarly 40 to 45). This is because the program contains a large loop (a characteristic of a round-to-round game program), and the increase in depth does not create any new feasible paths (the ends of search of depth 30 and depth 35 are “trapped” in the same iteration where the game is unfinished). The summaries here are long enough that they pass this threshold, and without solving any constraint, Cirrus calculates that the game cannot finish between depth 30 and 35. It is an inspiration to us that with the summaries gathering information from a larger range of code, we could derive better heuristics for directed search.

3.7 Conclusion and Future Work

This chapter presented the idea of fine-grained summarisation, with which one can establish compositional symbolic execution that helps to mitigate path explosion. The fine-grained CSE approach attempts to prune redundant solver calls from any target code fragment. We provide guidelines for code fragment selection, and a quantifying method to measure or predict the efficiency of a set of summaries on a particular code fragment, as well as three summarisation strategies with different features. The experiment based on our FGCSE implementation Cirrus shows promising results when we combine this technique with BSE, helping to improve execution time by reducing the number of calls made to the underlying constraint solver. We also find that the summarisation strategy used is critical to the effectiveness of Cirrus.

We think more static analysis methods can be used to improve the strategies, such as data-flow analysis and other heuristics [30] to identify the dependency between code fragments. Furthermore, with proper summarisation strategies, fine-grained summarisation has small overhead and can be easily combined with other SE optimisations, e.g., function-level summarisation. In Chapter 5, we show that particular solver techniques can improve FGCSE.
Chapter 4

Stratified Symbolic Execution

In the last chapter we talked about one approach of over-approximation in symbolic execution, namely Compositional Symbolic Execution (CSE). CSE introduces over-approximation on the context of a function (or an arbitrary piece of code in fine-grained CSE) when doing SE in the function. In this chapter we present an opposite of this idea—over-approximating a function call during the global SE. The former we also refer to as contextual over-approximation and the latter regional over-approximation. This technique, which we call Stratified Symbolic Execution, provides a new and more efficient way to verify certain properties of programs by distinguishing high-level and low-level elements of program structure. We have implemented Stratified SE as a tool called Stratus based on KLEE and conducted experiments from different angles, demonstrating the advantages and shortcomings of Stratified SE.

4.1 Motivation

Stratified Symbolic Execution (Stratified SE) is “stratified” meaning that it has a layered view of a program, which is achieved by over-approximation on certain function calls. Stratified SE has the potential to solve a combination of problems that are not well addressed by existing symbolic execution tools. First we discuss these limitations, then we discuss the advantages of Stratified SE.

4.1.1 Limitations in Understanding and Configuring Search Space

Traditional SE provides users with many ways of configuring the search, including setting the symbolic variables, setting a depth or time limit, choosing the search algorithm, etc., so that the users can minimise the time used in SE while allowing it to analyse the parts of a program that the users consider the most critical. Otherwise, SE could take an enormous amount of time due to path explosion. However, these options do not always meet the users’ needs, due to the following issues:

- **Obscure to use.** Although they might be useful to an expert SE tool developer, the configuration parameters are hard to master for testers without a solid understanding of symbolic execution. For example, a user might not know what a depth limit means for test generation. In other words, we have exposed internal SE concepts to users.

- **Unbalanced coverage.** Some options guarantee a finite search space (e.g., setting a depth limit) and full confidence (i.e., soundness and completeness) in the results, but they lead to unbalanced coverage. For example, a depth limit requires the search to
catch every possibility of execution in the “shallow” parts of the program, but neglects many of those in the “deep” parts.

- **Unprioritised coverage.** With the help of heuristic searchers, SE can automatically optimise the coverage by assigning weights to the execution paths. Despite the fact that it is widely adopted in the state of the art, it tends to consider all uncovered code with a similar priority (in exhaustive exploration), e.g., incorporating the distance to uncovered code as the weight of a state. It is also hard to guarantee coverage of a specific part of the program.

- **No partial verification result.** On real-world programs, symbolic execution times out often and it is usually incomplete. This does not prevent symbolic execution being an anytime analysis in regard to bug finding, i.e., stop it at anytime and the results (bugs found) are still meaningful. However, it is not an anytime analysis regarding program verification—SE needs to fully explore a program in order to obtain interpretable results, i.e., the evidence of the absence of certain types of bugs.

### 4.1.2 Differentiating the Roles of Code within a Program

A program’s high-level structure is usually reflected by functions in the shallow parts of the program’s **call graph**. These functions are the coordinators of the lower-level program components. Symbolic analysis on higher-level code might produce higher-quality results, e.g., a set of tests with higher coverage, in a limited time. Existing symbolic execution approaches do not have the concept of “structural levels”, and as a result the distribution of symbolic analysis time tends to be more intense in the shallower parts of a program’s execution tree rather than the call graph. In order to achieve good coverage, symbolic execution should first get a grasp on a program’s high-level structure.

Figure 4.1 illustrates a possible way of classifying the code in a program according to the roles. The dashed arrows show the traditional visit order of SE, marked by the numbers in circles.\(^1\) The visit order decides how the control flow is unrolled into an execution tree for SE. However, this order has no obvious relevance to the roles of the code.

To improve the efficiency of symbolic execution, we need to differentiate high-level and low-level functions. We propose to use a layered view of programs in Stratified SE: the code that directs the major flow of execution will belong to the high-level structure layer; the helper functions, library code, and code that manages special cases should belong to the low-level details; in between there could be additional layers consisting of code dedicated to specific functionalities that a program wants to support, ordered by their domination relation.

### 4.1.3 Verifying Programs Partially

Having a layered view of a program further allows us to partially or iteratively verify a program. Program verification using symbolic execution is to prove that a program is free of a given type of error, a problem that is undecidable. For this reason partial results are to be expected. Methods such as limiting the size of a symbolic input or iteratively deepening in depth are obscure to use in the sense that they rely on the concepts “symbolic variable” and

\(^1\)We assume forward symbolic execution.
4.1. Motivation

**Figure 4.1: Differentiating the roles of code: traditional depth vs reality.**

“depth”, whose problems have been pointed out earlier: we can verify a program with an assumption on the symbolic input size, but the verification result cannot be directly associated with certain program locations, which can be an inconvenience for debugging; a depth limit cannot be immediately connected to a code location either, and it might be alien to some users not from an SE background; depth limit itself also has a shortcoming in maximising the coverage.

There are existing SE methods providing results similar to a partial proof (as opposed to methods towards a complete proof such as those based on CEGAR [38]), which are introduced in Chapter 2. We briefly touch upon them once again here and discuss the advantages of using Stratified SE in achieving this:

- **SE for program component.** In many large scale programs, verifying all parts of the code in the same time is not possible. It is cheaper to isolate different components of the code and verify them separately. We have discussed similar ideas of doing SE on program parts in Section 2.3.4. Their disadvantage is that they are is context-insensitive and are prone to false alarms (of incorrectness).

- **Incremental SE and regression verification.** Incremental SE can also be used to produce partial verification results (see Section 2.3.1 and Section 2.3.3). It usually has three important counterparts: automatic change detection, directed symbolic execution (which is needed to reach the modified code), and a systematic way to prove that the paths in the modified code and all of the paths affected by the modification is covered by this round of symbolic execution. As a more general usage, it can be used to prove program parts more efficiently than traditional SE. However, it is designed for small program changes or components, rather than high-level functions, in which cases it is no more efficient than traditional SE. An approximation of incremental SE’s cost is given in Appendix A.5.2.
• **Concretisation.** Concretisation is discussed in Section 2.3.3. It allows executing the code that is of less importance to the current task (such as a library or system call, essentially in a lower layer) at a minimum cost and is a powerful way of simplifying symbolic execution. However, concretisation under-approximates the problem, which means it creates proofs that might not hold\(^2\), if it is used in such an application.

As existing approaches have their limitations for this particular use of SE, we propose to use Stratified SE. Using over-approximation on specific program parts, Stratified SE can be abstract about them and more thorough on the rest—to prove the correctness of a part of a program we do not always need very precise information about all the other parts. We have seen an example of this in Section 1.1.2. The key here is to stratify the program under analysis and prove it layer by layer. Based on this idea, we can use Stratified SE to iteratively prove a program, which means it is a pseudo-anytime analysis—if a program really is correct, after verifying each layer, we will be more confident about that fact. Moreover, if there is a presumption that can be made about an error, e.g., the error must belong to the main layer, stratified SE might be more efficient in showing its reachability/unreachability.

Compared to SE with input space partitioning or depth limits, Stratified SE makes it very clear which code components or code levels are verified; compared to doing SE on components, it is context-sensitive; compared to incremental SE, it has a different use case, i.e., it is primarily used on high-level code, and can iteratively deepen into the low-level code; compared to concretisation, its proofs always hold.

There exists closely-related work that over-approximates certain details in a path during the analysis of a program just like Stratified SE: program and path slicing techniques identify irrelevant path components according to certain criteria and group paths with the same relevant path component (a slice) to reduce path explosion (introduced in Section 2.3.2); variably interprocedural analysis\([127]\) allows a depth of inspection into invocations (see SE for program components in Section 2.3.4), which is similarly differentiating code levels for symbolic execution; an existing proof method incorporates uninterpreted functions to abstract function calls, and is used in, e.g., CSE\([1]\) (see Section 2.3.4). A major difference between these and our work is that we allow the recovery from the over-approximation of the invocations (or similarly other program components). We can consider Stratified SE an extension to these approaches.

### 4.1.4 Optimising Symbolic Execution by Reordering

Stratified SE essentially reorders SE such that a function can be explored as an entity earlier than its callees—putting the related parts of a program together and thus constraints on related variables together can improve the efficiency for some tasks, in a way similar to Chapter 3 where we argued that we should allow relevant code explored as an entity by not blindly applying summarisation.

For example, in the piece of code from OpenSSL in Figure 4.2, we see two types of “intrusive” code for SE, line 6 to 7 and line 12. The message callback at line 6 and 7 “interrupts” the path reasoning in SE. For example, to reach the true branch of line 8, the path constraint is roughly

\[
\text{hbtype} = s[0] \land \text{Constraints on } s \text{ from the callback function} \land \text{hbtype} = \text{TLS1\_HB\_REQUEST}
\]

\(^2\)This makes them unsound proofs. However, this “unsoundness” is to be distinguished from the other usage of this term in this thesis.
4.2 Method—Stratifying a Program

In this section, we present how we achieve the above features using over-approximation. An overview of Stratified SE is shown in Figure 4.3. We explain the concepts that appear in this
4.2.1 Invocation Depth

We propose the concept of *invocation depth* as opposed to the traditional depth in symbolic execution. Recall that the traditional depth of a piece of code is relative to the number of conditionals on the execution path that reaches it. It captures how hard it is to reach such code. The invocation depth, on the other hand, captures the level a function is in—higher-level functions have smaller depth and vice versa, in order to differentiate the priority as shown in Figure 4.1. As we discussed earlier, these two depth concepts do not always coincide. The way traditional symbolic execution works will always prioritise the code at a smaller traditional depth. With the concept of invocation depth, we can prioritise exploration in the smaller invocation depths, corresponding to the high-level structure of a program. The hypothesis is that we will gain better coverage, especially of user-defined code, than when we prioritise the smaller traditional depths. In the rest of the chapter, “depth” means invocation depth unless specified otherwise.

We call symbolic execution with the awareness of invocation depth introduced above *Stratified Symbolic Execution* (Stratified SE). Stratified SE achieves its goal by constructing a stratified call graph of the program and performing *skipping* of some function calls, returning to *resolving* them at a later time.

**Figure 4.3:** How Stratified Symbolic Execution captures the different layers of a program.

**Stratified Call Graph**

The stratified call graph is the program call graph with functions labelled with an invocation depth. The entry function of symbolic execution (e.g., `main`) is assigned depth 0. The stratification mechanism for the functions reachable by the entry in the original call graph works
4.2. Method—Stratifying a Program

as shown in Algorithm 4.1.

Algorithm 4.1 Stratifying the call graph of a program.

```plaintext
procedure STRATIFY(CallGraph)
   for each Function f in CallGraph do
      f.IvcDepth := MaxInt
      f.Visited := False
   end for
   f0 := ENTRY(CallGraph)
   f0.IvcDepth := 0
   Queue.PUSH(f0)  ▷ Obtain the entry function
   for each f := Queue.POP do
      for each Function call g(...) in f do
         if g.IvcDepth > f.IvcDepth then
            g.IvcDepth := f.IvcDepth + IsSKIPPABLE(g)
         end if
         if ¬g.Visited then
            g.Visited := True
            Queue.PUSH(g)
         end if
      end for
   end for
end procedure
```

IsSKIPPABLE(g) here returns 1 if g is skippable and 0 otherwise. Algorithm 4.2 details the skippability of a call. Algorithm 4.1 specifies that a skippable call increases the invocation depth by 1, if the called function has not appeared in an equal or smaller invocation depth. If a call is not skippable, the callee’s invocation depth is at most the invocation depth of the caller.

More generally, we can also consider the code blocks in a loop belong to a larger invocation depth than the loop’s own depth. We leave this more fine-grained way of stratification to future work.

4.2.2 Skippable and Non-Skippable Function Calls

Now we explain the selection process in Algorithm 4.2. Ideally speaking, we want all of the calls skippable so that we can prioritise them most freely. Realistically, not every function is skippable. Skippable function calls are calls to functions that do not return pointers, and belong to one of the two kinds below:

- Functions that do no more than reading to the memory of the enclosing scope;
- Functions that may write but do not write data containing addresses (which we refer to as “non-pure” data) to the memory of the enclosing scope.

The memory of the enclosing scope of a skippable function call is referring to the memory space that can be accessed by the caller. In summary, we want functions that do not create symbolic pointers for their parents.

The reason why we impose such constraints becomes clearer after the introduction of the skipping and resolving process in Section 4.2.3. Simply speaking, compared to the over-approximation of a non-pointer variable, the over-approximation of a pointer is much more costly and less solvable. We can skip functions outside of this range, but we believe that this would be less useful for usual configurations of symbolic execution.
We do not consider external function calls skippable since they cannot be resolved (see Section 4.2.3). Special function sets WhiteList and BlackList are given by the user. They include or exclude (respectively) functions as skippable, which gives the user control over skipping if desired. The underlined text indicates that it is achieved by looking at function or parameter attributes (e.g., readonly in LLVM IR [87] indicates whether a function can write to the address space accessible by its parent or not). We build Stratus on top of KLEE [26] and LLVM, which is why we need to detect specific built-ins. We also assume the functions we skip do not modify globals (see Section 4.5.4).

In addition to Algorithm 4.2, in some special cases, symbolic execution needs to use runtime information to finally tell whether a call is skippable, e.g., a dynamic call is not skippable unless the callee is concrete and skippable; a function with a symbolic pointer argument is not skippable if we cannot pointer-resolve it.

We return to the skippability of functions with pointer arguments after we discuss skipping and resolving.

Algorithm 4.2 Skippability of a function call.

4.2.3 Skipping and Resolving and the Over-Approximation

Stratified SE allows skipping a function call and resolving it later. Skipping is the process of over-approximating the influence of the function call. When a function call is skipped, symbolic execution will not execute the callee but will over-approximate the call and directly

\[\text{Algorithm 4.2 Skippability of a function call.}\]

\[
\text{procedure } \text{isSKIPPABLE}(g) \\
\text{if } g \text{ is external then} \quad \triangleright \text{Not contributing a resolvable skip} \\
\text{Return 0} \\
\text{else if } g \text{ is a (KLEE or LLVM) built-in then} \quad \triangleright \text{Depending on implementation} \\
\text{Return 0} \\
\text{else if } \text{WhiteList } \neq \emptyset \land g \notin \text{WhiteList} \text{ then} \\
\text{Return 0} \\
\text{else if } g \in \text{BlackList} \text{ then} \\
\text{Return 0} \\
\text{else if } g \text{ does not return (the execution) then} \\
\text{Return 0} \\
\text{else if } g \text{ returns a pointer then} \\
\text{Return 0} \\
\text{else if } g \text{ may modify parent memory then} \\
\text{for each Parameter type } \text{ParamType} \text{ of } g \text{ do} \\
\text{if } \text{ParamType} \text{ is a non-pure-data pointer type and not by-value then} \\
\text{Return 0} \\
\text{end if} \\
\text{end for} \\
\text{if } g \text{ modifies globals then} \quad \triangleright \text{Suboptimally implemented} \\
\text{Return 0} \\
\text{end if} \\
\text{Return 1} \\
\text{end procedure} \]

\[\text{Traditional SE will have to “skip” such functions or concretely execute them anyway, but that kind of skipping is different from what we discuss here because there is not a corresponding “resolving” process. In stratified SE, we can simply handle such functions like a traditional SE would do.}\]

\[\text{Program and path slicing also uses data dependencies to identify slices, which is a relevant topic (see Section 2.3.2); the combination of program optimisation (in our case, it matters because it produces these attributes) and symbolic execution, or the testability optimisation is also a topic in the literature (see Section 2.1.2).}\]

\[\text{Symbolic pointers are typically introduced by pointer arithmetic involving other symbolic variables, e.g., incrementing a pointer by a symbolic value. In other words, we usually do not start SE with symbolic pointers as program inputs, which means in many cases a symbolic pointer does not represent an arbitrary value, and can be associated with certain concrete memory segments, e.g., the memory space of an array. Here, } \text{resolving} \text{ is a KLEE [26] terminology meaning that we identify the objects a symbolic pointer can point to and continue SE using a separate state for each object.}\]
execute what comes behind the call. Resolving is the process of revisiting the function calls that have been skipped and executing the callee. Essentially, skipping and resolving alter the order of constraint collection. With the following methods, the re-ordered constraints are equivalent to the original constraints.

The Skipping Process

When skipping a call, we replace the potentially-modified variable (or memory locations) in the caller scope with fresh symbolic variables. The return variable of the function is also considered symbolic. This over-approximates the influence of the function call. The symbolic variables will be used to do symbolic execution on the code that follows the call. For example, if a skipped function returns an integer, this integer will be considered symbolic in any expression later where it appears. Meanwhile, the replaced variables are remembered for later recovery of the calling context.

The example in Figure 4.4 can help us understand the skipping process. Suppose we are to skip function abs which takes a number and returns another. The changes in states during this process can be seen in Figure 4.5a. Here \( \eta \) and \( \mu \) are symbolic states, each storing
a current value for each variable and a path constraint. A line number means we are about to execute that line in the next step. The skipping happens between rows 2 and 3 in the table, marked by two dashed lines. We record the variables used as arguments in the function call, in this case $x = 5$ in the coloured cell. We also over-approximate the influence of the call, i.e., making the variables that might be modified by the function symbolic, in this case we make the return value symbolic $y = Y$. In the rest of the execution in this function, $Y$ is used in lieu of $y$.

The Resolving Process

When resolving a call, we use the constraints we have collected so far, including those from the code after the call, and symbolically execute the call as if we did not skip—only that we have additional constraint information from after the call. This is achieved by restoring the calling context with the variables remembered earlier.\(^6\)

When symbolically executing a call that has been skipped before (i.e., resolving), we can yet do another skip when we see a skippable call, using the above method. When we see a return from the function being resolved, we will add additional constraints to the states that equate the actual return value of the function or the parameters passed by reference—both could be symbolic expressions—with the corresponding symbolic variables we have used to replace them. This way, we relate all information about this execution path outside and inside the callee. Owing to over-approximation, we will find some of the states fail in these equality constraints. After we discard such states, we complete one resolving process, and can continue to resolve other skipped calls. After we complete all the resolving, the resulting states will be the same as if we have not skipped any call (assuming symbolic execution on this program terminates).

Using the same example in Figure 4.4, we can see how resolving is done. When we reach the end of the entry function (line 7, recall that it indicates termination in the function), we return to every function call we have not visited, i.e., the skipped call to $\text{abs}$. We first restore the calling context, making $\text{abs}$ take $x = 5$ as the argument instead of $x = 6$ or $x = 5 + Y$. Then we do symbolic execution in the skipped function, with the additional constraints we have collected. Now consider Figure 4.5b. Standard symbolic execution is performed until we reach the return, where an additional equality constraint between the return value (represented by $r$) and the symbolic variable we used to replace the return value with ($Y$) is added to the states. With this constraint, only $\mu$ is feasible now, while we know $\eta$ comes from over-approximation.

As we discuss in Section 4.1 and earlier on the motivating example in Section 1.1.2, under-approximation is usually not recoverable, and by using it one will lose confidence of a certain aspect of the code. However, stratified symbolic execution is over-approximating, and by introducing the resolving process, it preserves the chance of confirming whether an error is truly an error or a false alarm. Depending on the application, the resolving process can be enabled, disabled or selectively enabled.

We shall shortly detail the above two processes (Algorithm 4.3).

--

\(^6\) LLVM IR uses Static Single Assignment (SSA) so each modification to a variable also creates a renamed copy of it. This gives us the convenience that we do not need to remember a variable that is not replaced by a symbolic variable in order to restore the exact calling context. Implementations based on other frameworks can achieve the same with careful variable versioning such as what we introduce in Section 3.3.1.
4.2. Method—Stratifying a Program

Now we have a look at a harder problem. In the example in Figure 4.6, the function takes a pointer as its argument. Compared to the previous example, the function can potentially modify the memory pointed to by the pointer (denoted by $@32$). So, when skipping, in addition to recording the value of $x$ and as $@32$, and we have to make $@32$ symbolic (using $Y$); when resolving, we have to construct an equality constraint between $Y$ and the symbolic expression of $@32$ at the end of the execution of $abs$. We do not need to do this if the parameter passed is by-value.

Here we can see the skipping process has two guarantees:

- It over-approximates what the call can do;
- It leaves the opportunity of recovering from the over-approximation open.

When a function potentially modifies a variable, over-approximation requires that the variable is marked as symbolic; if it potentially modifies a memory cell, then the cell has to be marked as symbolic. Similarly, if a function modifies a variable of a pointer type, it should mark the variable as symbolic. However, this can be problematic, because in the worst case (no other constraints present in the current state), we have to make a very general assumption that this pointer can point to anything. For example, if the program is operating within a 1-kbit memory space, we should then make the most general assumption that an unconstrained symbolic pointer can point to anywhere in the 1-kbit memory space. This is overly general and too costly.

Dealing with symbolic pointers is one of the most time consuming processes in symbolic execution, for which approximation techniques are used, e.g., logical memory models [26] and write concretisation [29, 120]. We aim to avoid symbolic pointers, especially unconstrained symbolic pointers. We do not consider the functions that can modify pointers skippable—from the parent function point of view, they make those pointers symbolic. For such a function, we consider it in the same invocation depth as the parent (see Section 4.2.1), which effectively says they are of the same priority. Considering this in a different way, such a function and its parent are likely more dependent on each other than the functions that do not modify a pointer, so there is not as strong a reason to differentiate them.

An alternative to handle pointer-modifying a function call is by skipping the call but not making the potentially-modified pointers symbolic, effectively under-approximating. This method introduces under-approximation on the potentially-modified pointers and thus false negatives. We can consider this a potential point to make trade-off for efficiency.

(A) The entry function. (B) Function $abs$.

Figure 4.6: Example program illustrating skipping and resolving of a function that takes a pointer.

**Skipping Functions with Pointer Arguments**

Now we have a look at a harder problem. In the example in Figure 4.6, the function takes a pointer as its argument. Compared to the previous example, the function can potentially modify the memory pointed to by the pointer (denoted by $@32$). So, when skipping, in addition to recording the value of $x$ and as $@32$, and we have to make $@32$ symbolic (using $Y$); when resolving, we have to construct an equality constraint between $Y$ and the symbolic expression of $@32$ at the end of the execution of $abs$. We do not need to do this if the parameter passed is by-value.

Here we can see the skipping process has two guarantees:

- It over-approximates what the call can do;
- It leaves the opportunity of recovering from the over-approximation open.

When a function potentially modifies a variable, over-approximation requires that the variable is marked as symbolic; if it potentially modifies a memory cell, then the cell has to be marked as symbolic. Similarly, if a function modifies a variable of a pointer type, it should mark the variable as symbolic. However, this can be problematic, because in the worst case (no other constraints present in the current state), we have to make a very general assumption that this pointer can point to anything. For example, if the program is operating within a 1-kbit memory space, we should then make the most general assumption that an unconstrained symbolic pointer can point to anywhere in the 1-kbit memory space. This is overly general and too costly.

Dealing with symbolic pointers is one of the most time consuming processes in symbolic execution, for which approximation techniques are used, e.g., logical memory models [26] and write concretisation [29, 120]. We aim to avoid symbolic pointers, especially unconstrained symbolic pointers. We do not consider the functions that can modify pointers skippable—from the parent function point of view, they make those pointers symbolic. For such a function, we consider it in the same invocation depth as the parent (see Section 4.2.1), which effectively says they are of the same priority. Considering this in a different way, such a function and its parent are likely more dependent on each other than the functions that do not modify a pointer, so there is not as strong a reason to differentiate them.

An alternative to handle pointer-modifying a function call is by skipping the call but not making the potentially-modified pointers symbolic, effectively under-approximating. This method introduces under-approximation on the potentially-modified pointers and thus false negatives. We can consider this a potential point to make trade-off for efficiency.
## Chapter 4. Stratified Symbolic Execution

### (A) The skipping process in the entry function. The leftmost column is the line numbers from Figure 4.6a.

<table>
<thead>
<tr>
<th></th>
<th>State</th>
<th>$x$</th>
<th>$y$</th>
<th>$\oplus 32$</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\eta$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>$\eta$</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>$\eta$</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>$\eta$</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>$\eta$</td>
<td>32</td>
<td>$Y$</td>
<td>$Y$</td>
<td>True</td>
</tr>
<tr>
<td>6</td>
<td>$\eta$</td>
<td>32</td>
<td>$Y$</td>
<td>$Y &gt; 10$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$\mu$</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$\mu$</td>
<td>32</td>
<td>$Y$</td>
<td>$Y &gt; 10$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$\eta$</td>
<td>32</td>
<td>$Y+1$</td>
<td>$Y &gt; 10$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>32</td>
<td>$5 + Y$</td>
<td>$Y \leq 10$</td>
<td></td>
</tr>
</tbody>
</table>

### (B) The resolving process in the abs function. The leftmost column is the line numbers from Figure 4.6b.

<table>
<thead>
<tr>
<th></th>
<th>State</th>
<th>$z$</th>
<th>$w$</th>
<th>$\oplus 32$</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\eta$</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td>$Y &gt; 10$</td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td>$Y \leq 10$</td>
</tr>
<tr>
<td>3</td>
<td>$\eta$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y &gt; 10$</td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y \leq 10$</td>
</tr>
<tr>
<td>4</td>
<td>$\mu$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\mu$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$\eta$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y &gt; 10$</td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y \leq 10$</td>
</tr>
<tr>
<td>7</td>
<td>$\eta$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y &gt; 10 \land Y = \oplus 32$</td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>32</td>
<td>5</td>
<td>5</td>
<td>$Y \leq 10 \land Y = \oplus 32$</td>
</tr>
</tbody>
</table>

**Figure 4.7:** The symbolic states during skipping and resolving of a function that takes a pointer.
4.2. Method—Stratifying a Program

Having established the concepts of skipping and resolving, we can now talk about Stratified Symbolic Execution (Stratified SE). In Stratified SE, a state executes a program layer by layer, not following an actual execution path. An execution path travels between different invocation depths. Unlike traditional depth, the invocation depth does not strictly increase along this path. For example, a path can start from the *main* function, go into a function called by *main* which makes it depth 1, and then return to *main* where the invocation depth becomes 0 again. By skipping and resolving, Stratified SE unconventionally does not propagate a state along an execution path but collects constraints from one invocation depth and then another, incrementing it.

Algorithm 4.3 Stratified Symbolic Execution.

```
procedure STRATIFIEDSE(P)
    States := {INITSTATE(P)}  ▷ All fields empty, Instr set to the entry
    while States ≠ ∅ do  ▷ See Section 4.3
        State := SEARCHERSELECT(States)
        if State.Instr is a call and IS_SKIPPABLE(State.Instr) then
            if State.Instr ≠ State.Resolving then  ▷ Skipping
                State.SkipQueue.PUSH(State.Instr)
                RECORDCALLINGCONTEXT(State, State.Instr)
                State.Instr := State.Instr.NEXT
            end if
        else if State.Instr is a program exit then
            State.Reached := 0
            ENDIVCDEPTH(State)  ▷ See Section 4.4.2 for targeted exploration
        else if State.Instr is an error (or a target) then
            State.Reached := State.Instr
            ENDIVCDEPTH(State)
        else if State.Instr is a return and State.Instr = State.Resolving then
            State.Resolving := 0
            if ¬CHECKCONSISTENCY(State) then  ▷ An infeasible state from over-approximation
                TERMINATE(State)
            else if State.ResolveQueue ≠ ∅ then  ▷ Has another unresolved call
            else
                ENDIVCDEPTH(State)
            end if
        else
            NORMALSE(State)  ▷ When branching, all fields copied accordingly
        end if
    end while
end procedure

procedure INITSTATE(P)
    State := {IvcDepth ⇒ 0, Instr ⇒ 0, Resolving / Reached ⇒ 0, SkipQueue / ResolveQueue ⇒ ∅}
    State.Instr := ENTRYINSTR(P)
    Return State
end procedure

procedure ENDIVCDEPTH(State)
    if State.SkipQueue ≠ ∅ ∧ NEEDRESOLVE(State) then  ▷ Next IvcDepth, see Section 4.4.2
        State.ResolveQueue := State.SkipQueue
        State.SkipQueue := ∅
    else
        if State.Reached then
            REPORTREACHANDTERMINATE(State)
        else
            TERMINATE(State)
        end if
    end if
end procedure
```
Chapter 4. Stratified Symbolic Execution

Algorithm 4.3 shows the algorithm of Stratified SE which is an extension of normal SE. A state is also an extension of a normal SE state. In summary, it is a process of deepening in the invocation depth. Symbolic execution skips the skippable function calls on the execution paths. When it hits an end of the current depth, i.e., a program exit, an error, a target, or the return of the last function to resolve, it will start to explore in another invocation depth. In particular, failure in resolving a call will make a state terminate; success in resolving all skipped calls will confirm the feasibility of the execution path. Examples of it are already seen in Figure 4.4 and Figure 4.6. Comparing Figure 4.3 with Figure 4.1, we see how stratified SE is better at understanding different layers of a program than the traditional way.

NEEDRESOLVE always returns True in exhaustive exploration while targeted exploration is discussed later in Section 4.4.2. Note that an error discovered is only a potential error before all of the skipped calls are resolved, owing to the over-approximation. So is a target. If a state survives all the equality constraints from resolving the calls, an error or a target reach can be confirmed.

The searcher prioritises the states in the work stack. It is introduced in Section 4.3.

4.2.5 The Symbolic Execution Tree with Skipping and Resolving*

We can see skipping and resolving’s potential by using symbolic execution trees as a tool for cost estimation. That is, we have the following approximation:

Proposition 1 (Approximation of the size of the problem when skipping is on) Let $T'$ and $T$ be the symbolic execution trees with and without skipping respectively. They satisfy

$$\exists T'' \left( T' \sim T'' \land T'' = \text{mg} \right)$$

While merging reduces the size of the symbolic execution tree, over-approximation might increase it, which is why we discuss the suitability of functions for skipping.

The resolving process is effectively doing symbolic execution for particular parts (the skipped functions) of a program, with the additional information from the parent invocation depths. In this regard, each resolving process has a separate symbolic execution tree, approximating the size of the problem. We have the following theorem:

Proposition 2 (Approximation of the size of the problem in Stratified SE) We use $m$ to denote the size of the problem in Stratified SE:

$$m = |T_0| + \sum_{T \in T_1} |T| + \sum_{T \in T_2} |T| + \ldots$$

where

$$T_i = \begin{cases} 
{T_0}, & i = 0 \\
\{T(PC(R)) \mid T \in T_{i-1} \land u \in \text{leaves}(T) \land R = \text{path}(T, u)\}, & i > 0
\end{cases}$$

$T_0$ is the symbolic execution tree for symbolic execution at invocation depth 0. $PC(R)$ is the postcondition of path $R$ and equivalently the precondition to the resolving process.

This formula gives the sum of the size of the symbolic execution trees of each invocation depth. The number of symbolic execution trees in each invocation depth depends on the number of leaves in the trees of the parent invocation depth.
4.2.6 Convergence to Standard SE

We can see that stratified SE is equivalent to traditional SE on any program because for every symbolic state, stratified SE computes an equivalent path constraint after resolving the skipped calls in this state. That is, we have the following:

**Theorem 1 (Equivalence between Stratified SE and standard SE)** Stratified SE and standard SE converge given unlimited execution time and space.

Given a program $P$, suppose a stratified symbolic executor $\epsilon'$ uses the same set of rules in the interpretation of the program as a normal symbolic executor $\epsilon$, except that it does skipping and resolving upon seeing skippable function calls (the skippability is defined by Algorithm 4.2) using Algorithm 4.3. For any program execution, if $\epsilon$ decides that it is feasible, then for $\epsilon'$, there exists a time point after which it also decides it is feasible, otherwise $\epsilon'$ is not terminating; if $\epsilon$ decides it is not feasible, then $\epsilon'$ will never conclude that it is. Here, the term feasible for executions is from Appendix A. The time it takes for stratified SE to decide the feasibility of an execution is not necessarily finite, hence the non-terminating case. This is to be expected: when the search strategies are different, a symbolic execution tool cannot guarantee to reach a state that another tool can reach in finite time. Conversely, traditional SE cannot guarantee to reach a state that stratified SE can reach in finite time.

4.3 A Difficulty and a Solution—The Stratified Searcher

In the previous sections, we have introduced the framework of stratified symbolic execution. In this section, we introduce its searcher.

Generally speaking, in symbolic execution, a searcher is an algorithm that identifies the most beneficial state to execute, based on the statistics associated with each state, e.g., the number of unique lines it has covered. The symbolic executor (see Appendix A for a definition) itself also prioritises the exploration. A search algorithm is a refinement on such prioritisation.

Their roles' difference is mainly that, an executor chooses which part of a program to search first, and then generates states to cover it; a searcher chooses which state to reason first. An executor might choose to explore the program top-down or bottom-up; it might choose to explore it path by path or function by function. Depending on different ways of exploring the program, it gets different set of states in the process. A searcher in the meantime takes the set of states and deciding the order of reasoning them.

A good searcher prevents the exploration from being oblivious such as getting stuck in loops. In some applications, it helps the exploration reach a target code area faster, or it helps symbolic execution abort useless states.

In traditional and stratified symbolic execution, the executor considers a program differently, thus we also need to adjust the searcher so that it understands what is the best choice given such different set of states to choose from. Unlike the traditional approaches, stratified exploration creates a difficulty for the searcher due to over-approximation.

4.3.1 Search Heuristics and Discard Policy

Typically, a searcher assigns scores to the states, and does it differently according to different needs [24] (more about this topic can be seen in Section 2.3.2). In our research, we are particularly interested in coverage-time efficiency, so the state that leads to the most coverage
with the least cost is considered the best state to select next. Therefore, we use line coverage as a measure. Therefore, our searcher will need to prioritise the state with the most potential in new line coverage. Here, new line coverage means coverage on lines that have not been covered yet. How is such potential quantified? In many traditional symbolic execution tools, this can be speculated by looking at the new lines that a state historically covers. For example, in KLEE, to evaluate a state’s importance the default search strategy is taking in account how long ago it covers a new line and its distance to another new line (see Section 4.3.3).

In KLEE, states that cannot produce more coverage compared to other states are practically “useless”. KLEE aims for a minimum set of states to generate tests to avoid large test suites. It is costly to store them, run them, or analyse them. Therefore, it does not let those useless states generate tests, effectively discarding them.

With over-approximation, some of these concepts are still applicable to stratified symbolic execution, but the others need some adaptation.

### 4.3.2 Must-Instructions and May-Instructions

From now on until the end of this chapter, we may refer to program “lines” as “instructions”. Assuming traditional symbolic execution is precise enough—in particular, being sound at any instant of execution—the searcher can consider each instruction covered by a state to be a must-covered instruction, or must-instruction. However, during stratified symbolic execution, not all instructions visited by states are must-instructions.

If a state that has skipped a function reports that it covers an instruction, one of three cases could occur:

- It appears before any call skipped by this state in its execution path. In this case, we can consider the instruction a must-instruction.
- It appears after some skipped calls in the execution path, but these are all resolved successfully, i.e., the execution path up to the instruction is proven feasible. In this case we can still consider it a must-instruction.
- Before the instruction some call was skipped but not resolved, in which case, there may be evidence from that call that contradicts the existence of such execution path and the coverage can be a false claim. We call this kind of instructions may-covered instructions, or may-instructions.

Figure 4.8 is an example with must- and may-instructions. Suppose \( \text{id} \) (an identity function) and \( \text{exp} \) (an exponential function) are both skipped by a state. Only \( \text{id} \) is successfully resolved and \( \text{exp} \) is yet to be resolved. Suppose that it claims to have covered line 1 to 6. For this state, we know that the claims on line 1 to 5 are valid, but the claim on line 6 is not. Line 6 may or may not be covered depending on whether \( z < 0 \) is satisfiable or not. In this particular case, it is not satisfiable because no paths in \( \text{exp} \) can give a negative return value. Symbolic execution will only know the claim is false after it has resolved the skipped function before a covered line. On the other hand, if a different state claims to have covered line 1 to 5 and line 8, it will be able to find a feasible path in \( \text{exp} \) when resolving it, and thereafter know that the claim remains true.

---

7We also evaluate branch coverage for most of our experiments to aid the analysis and discussion. We consider path coverage the most fine-grained and full path coverage the theoretical goal. Although for some applications, path coverage is still insufficient [76].
4.3. A Difficulty and a Solution—The Stratified Searcher

The importance of a state can be hard to judge, owing to the existence of may-instructions. Specifically, there are two challenges:

- **A searcher cannot prioritise states using the old strategies.** Traditional searchers use strategies that depend on identifying the distinct must-instructions in a state in order to score it. With may-instructions, how is the score calculated?

- **An executor cannot discard states using the old strategies.** Traditional symbolic execution selectively discards states when they reach their end of execution, based on the distinct must-instructions they have. In stratified symbolic execution, a state cannot be discarded when it has a may-instruction that is not also a must-instruction in any other state, because it can possibly cover that instruction which no other state can confirm covering.

The example program in Figure 4.9 and Figure 4.10 demonstrates how discarding a state with a may-instruction can be bad. Suppose \( \eta_1, \eta_2, \eta_3 \) are states that have skipped \( \text{hash}(x) \) and claim to have covered lines as shown in the comments. Suppose they are not going to resolve the call because of its hardness. Now, \( \eta_1, \eta_2 \) clearly has distinct covered lines (line 3 and 10, with 10 being a may-instruction, which still makes \( \eta_2 \) stand out). For maximum line coverage, we need to generate tests for these two. Should we generate for \( \eta_3 \) as well? Because line 8 is a may-instruction, neither of \( \eta_1, \eta_3 \) are certain that they can cover it. For maximum line coverage, we have to produce a test for \( \eta_3 \) in order to increase the chance of hitting line 8.

However, it turns out that with skipping, many states have may-instructions that other states cannot confirm as a must. So, we have to generate tests for all of them, which as a result gives us an enormous set of tests. If we discard states like \( \eta_3 \) instead, we will end up losing potential coverage. This becomes another problem of may-instruction.\(^8\)

\(^8\)A direct solution is adding a process of test selection (e.g., concretely executing the tests and identifying the useful tests) after symbolic execution. However, this can still be costly.
Chapter 4. Stratified Symbolic Execution

\( \eta_1 \) \( \eta_2 \) \( \eta_3 \)

<table>
<thead>
<tr>
<th></th>
<th>( \eta_1 )</th>
<th>( \eta_2 )</th>
<th>( \eta_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( p )</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( q )</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(A) No state resolves the call. Line 8 and 10 are may-instructions.

(B) \( \eta_1 \) resolves the call, making line 8 must-covered.

\textbf{Figure 4.10:} Must- and may-coverage of states. The leftmost column is the line numbers. An underlined number means it is covered as a may-instruction in all of the covering states. Number “1” stands for “cover” and “0” otherwise. A bolded number “1” stands for distinct coverage, assuming \( \eta_1, \eta_2, \eta_3 \) appear during symbolic execution in said order. \( p \) is the number of may-instructions in each state and \( q \) is the number of distinct must-instructions (the “1”的s) in each state.

4.3.3 The Search Algorithm in Stratified Symbolic Execution

This section introduces a solution to the difficulties brought by may-instructions.

\textbf{Prioritisation}

Our searcher is based on KLEE’s Non-Uniform Random Search (NURS) with Coverage-New (\texttt{nurs:covnew}) [26]. \texttt{nurs:covnew} prioritises the states by calculating a score consisting of two factors:

\[
Score(\eta) = \left( \frac{1}{\max(1, d_1 - 1000)} \right)^2 + \left( \frac{1}{d_2} \right)^2
\]

Here, \( d_1 \) is the distance (the number of instructions) between the last instruction and the last new instruction of the state \( \eta \). \( d_2 \) is the topological distance to the closest instruction that has not been covered. In the case that there is no new instruction to cover, \( d_2 \) is 10000. The instructions are all considered as must-instructions in KLEE. States have a possibility proportional to \( Score(\eta) \) to be selected and executed.

In Stratus, we incorporate another factor \( p \) which is the number of may-instructions, and the score of each state \( \eta \) becomes

\[
Score'(\eta) = Score(\eta) + \left( 1 - \frac{1}{p + 1} \right)
\]

When \( p = 0 \), we have \( Score'(\eta) = Score(\eta) \); when \( p > 0 \), the score increases as \( p \) increments (see Figure 4.11). Our stratified searcher behaves similarly as \texttt{nurs:covnew}. It will prioritise the states with the higher scores by having higher chances to select them. The possibility is proportional to \( Score'(\eta) \). Note that the weight of the three score component (depending on \( d_1, d_2 \) and \( p \) respectively) is the same. They all vary between 0 and 1.

A number of may-instructions \( p \) is maintained by every state. When a state covers an instruction it will mark it as a must-instruction if it has no unresolved skip before it, or it
4.3. A Difficulty and a Solution—The Stratified Searcher

will mark it as a may-instruction if there is at least one unresolved skip before it. If a state successfully resolves a skipped call, it will update the must/may status of the instructions it covers so that they comply with this requirement, because instructions after the resolved skip may be now confirmed as must-instructions. When an instruction in the program is confirmed a must-instruction by any state, all other states that mark the instruction as a may-instruction must change it to a must-instruction.

Note that we are not proposing a search algorithm but rather a way to let searchers work with may-instructions: other search heuristics (see Section 2.3.2) can also incorporate this factor to measure the confidence of a state’s coverage so as to work with Stratified SE.

Figure 4.10 shows how we mark the instructions as must- or may-covered and how we change them for the program in Figure 4.9. Let us assume that \( \eta_1, \eta_2, \eta_3 \) appears in order during symbolic execution. The first state to reach a line not reached by the other states can mark it as distinctly covered. The latecomers can mark it as covered but not distinctly. Such order also depends on the prioritisation given by the executor and the searcher, but this is not important here.

Coverage on may-instructions cannot count as distinct. For example, neither \( \eta_1 \) nor \( \eta_3 \) can make line 8 a distinct coverage. Instead each may-instruction covered counts towards \( p \). We can see that all three states have \( p = 1 \) in Figure 4.10a because they all cover one may-instruction.

Suppose \( \eta_1 \) successfully resolves the call thus confirming the coverage on line 8. It now becomes a must-instruction for \( \eta_1 \), and since it is the first one to do so, we mark it as distinct. Line 8 is still may-instruction for \( \eta_3 \) but it does not matter any more, as some other state \( \eta_1 \) can confirm the coverage on it. \( \eta_3 \) is no longer useful, at least for the purpose of covering line 8. Therefore line 8 is equivalently a non-distinct must-instruction for \( \eta_3 \) and we decrement its \( p \).

**Discard Policy**

How do we discard tests of states so that we are not overwhelmed by a high number of tests eventually?
If states do not fail on resolving a call, then the instructions after that can always be confirmed as must-instructions. Note that a state eventually will have no may-instruction if it has resolved all the calls it has skipped. Though sometimes a state stops execution before it can finish the intended exploration, e.g., when a certain depth/invocation depth limit is reached, or when the solver timeout is exceeded. These are the states that might contain may-instructions. We want to make sure that the may-instructions in these states are worth test generation. However, there is no direct way to obtain such information. So instead, we further restrict the skippable calls using runtime information, in addition to the requirements discussed in Section 4.2.2.

We introduce a limit on the number of times the states fail to resolve a call. If we have many of states failing on resolving a particular call, i.e., a lot of states become infeasible after resolving the call, we will accept that the call is too hard or too expensive to resolve and we switch to normal symbolic execution for that call. Later in symbolic execution, no state will skip that call. By doing this, we can still apply the traditional discard policy (treating may-instructions as new instruction for every state), only we have significantly reduced the number of states that contain unreliable coverage on may-instructions, and significantly reduced the number of tests.

Take Figure 4.10 as an example, when \( p \) and \( q \) are both non-zero, we cannot discard a state; when they are both 0, we can consider it useless in terms of line coverage. We do not generate a test for such a state.

### 4.4 Applications

We have discussed how to achieve stratified symbolic execution. In this section, we introduce the applications of this technique.

#### 4.4.1 Exhaustive Symbolic Execution

Stratified SE can be used to exhaustively explore programs. When we do not specify any target in Algorithm 4.3, it is a exhaustive exploration algorithm. An optional invocation depth limit can be set in stratified symbolic execution, as shown in Algorithm 4.4. NEEDRESOLVE is further explained in the next section.

#### 4.4.2 Targeted Exploration

Stratified SE can be used for targeted exploration using selective resolving. Here, targeted exploration means SE having a target line to reach in a program. We have seen what a target line is in Algorithm 4.3. Specifying a target line is an available option in Stratus. In this section, we show how selective resolving is achieved and how it is beneficial.

In targeted exploration, Stratified SE can selectively resolve a state. This is shown in Algorithm 4.4, which is part of Algorithm 4.3. When a state has not reached any target\(^9\), and when it is at or over a depth \( \text{TargetIvcDepth} \), it will not be resolved. The depth \( \text{TargetIvcDepth} \) is an optional value set by the user. It means the maximum depth at which a target can be, in case that user can provide information about the target’s location. For example, if the target appears only in the \texttt{main} function, we can set this value to 0. When it is not set, this condition is not checked (greyed out and surrounded by parentheses).

\(^9\)Here for simplicity we consider reaching a target and an error similar thing, both represented by \texttt{Reached}. 
4.4. Applications

**Algorithm 4.4 Selective resolving.**

```
procedure NEEDRESOLVE(State)
  if State.IvcDepth ≥ IvcDepthLimit then
    State.Reached := 0
    Return False
  else if TargetedExploration ∧ ¬State.Reached ∧ State.IvcDepth ≥ TargetIvcDepth then
    Return False
  else
    Return True
  end if
end procedure
```

![Diagram](image-url)

**Figure 4.12:** Using selective resolving to do targeted exploration.

Because Stratified SE prioritises the exploration in smaller-invocation-depth code, it can reach a target line faster than the traditional way, even the target is deep in the sense of traditional depth. For example, a target line is set near the end of the main function. Traditional SE will have to go through all the function calls in the middle before reaching that line while stratified SE can skip some of them. Stratified SE would need to resolve the skipped calls later, but it only has to do this for the states that have potential coverage on the target. It gains time reduction by not spending time on the other states. Also, because it can get close to the target earlier, it might obtain useful information about how to reach the target.

Using the example in Figure 4.12, similar to the ones we have seen before, we can see how selective resolving is done. Here, we show three different states generated during SE. Suppose η₂ and η₃ hit the error, denoted by a red cross. We only need to resolve them when we reach the end of the target invocation depth, which is assumed 0. As a result, we can save the time from executing η₁. On the other hand, an error is not confirmed until the skipped calls are resolved. In this case, η₃ failed in a resolving process, making the state infeasible. Only η₂ is confirmed to reach the error.

A concrete example can be seen in Figure 4.13 and Figure 4.14. Suppose we want to reach the assertion line. We mark it as the target and because we know it is at invocation depth 0...
Chapter 4. Stratified Symbolic Execution

(A) The entry function.

(B) Function gcd.

\[ \begin{array}{l}
\text{State} & x & y & z & \text{Constraints} \\
1 & \eta & 0 & 0 & 0 & \text{True} \\
2 & \eta & X & 0 & 0 & \text{True} \\
3 & \eta & X & X & 0 & \text{True} \\
4 & \eta & X & X & Z & \text{True} \\
5 & \eta & X & X & Z & \text{True} \\
6 & \mu & X & X & Z & \text{True} \\
7 & \mu & X & X & Z & \text{True} \\
\end{array} \]

Figure 4.13: Example program illustrating selective resolving.

(A) The skipping process in the entry function. The leftmost column is the line numbers from Figure 4.13a. Suppose line 5 is the target. The target line and the state reached the line is shown in bold.

(B) The resolving process in the gcd function. The leftmost column is the line numbers from Figure 4.13b. For simplicity, we replace X with its concrete value. State \( \eta \) is effectively a concrete state. We can execute such a state quickly. At line 5, skipping and not skipping the call makes no difference (because the call is immediately resolved after skipped) besides a temporary symbolic variable \( C \). Still, it does not create any additional symbolic state so it is not much of a difficulty.

The remaining resolving process is similar.

Figure 4.14: The symbolic states during targeted exploration using Stratified SE.
and not deeper, we make TargetIvcDepth 0. Stratified SE will only resolve states that have reached the target. In this example, only $\eta$ reaches the line and $\mu$ does not. $\eta$ is very simple to execute since it is effectively concrete. At the end of invocation depth 0, $\mu$ is terminated, and only $\eta$ gets resolved. Resolving $\eta$ is fast because it does not generate other states. In contrast, using traditional symbolic execution, the initial state has to dive into \texttt{gcd} and the conditionals in it generate additional states. The symbolic execution tool cannot reach the target line before it unfolds the call four times. Meanwhile it generates many useless states.

A good heuristic searcher in a directed symbolic execution tool might be able to find the shortest path to a target line, but generally speaking this is still expensive because it is being too faithful to the actual execution path. Besides, stratified SE can be used in combination with heuristics after taking may-instructions into account.

### 4.4.3 Partially Proving a Program

Algorithm 4.3 can be used for incrementally proving a program. All we need is a searcher prioritisation function that makes the states at smaller invocation depths come ahead of the others. Most naively, this can be achieved by Algorithm 4.5, in place of the original SearchERSelect.

**Algorithm 4.5** Prioritising smaller invocation depths.

```plaintext
procedure SearchERSelectSmallest(States)
    Smallests := \{State | \forall State' \in States (State.IvcDepth \leq State'.IvcDepth)\}
    Return SearchERSelect(Smallests)
end procedure
```

This way, stratified symbolic execution will propagate all the states at the smallest invocation depth first. Each time a depth is fully explored, i.e., every state on this depth terminates or its depth increments, we know that we have fully verified this depth. Because this is using over-approximation, it is a complete analysis (but not sound). Compared to traditional symbolic execution, we can get partial results from it even if it does not terminate.

Furthermore, if a depth limit IvcDepthLimit or a target depth limit (TargetIvcDepth) is set, stratified SE makes use of such information and will be more efficient. This is similar to the previous sections, i.e., stratified SE can selectively resolve only the states that have reached an error in the small depths. We saw an example in Figure 4.13. Note that unlike the concept of symbolic variable length or traditional depth which is typically used to limit the search space, invocation depth is meaningful in software design. It is associated with the level of code in the program structure, which is a more friendly concept to testers. It can easily be visualised using a stratified call graph.

This idea can be illustrated using Figure 4.12. Suppose there is no error in the program, and we want to prove that. Traditional symbolic execution can prove this program only when it finishes exploring the entire code base. Using stratified SE, we can prove it iteratively. After exploring each depth, we get a meaningful result: the functions in this depth are correct. The user can then proceed to deeper depths by continuing to resolve; or stop but still end up with something useful. When there are targets that we want to prove unreachable, we can also use the way we have seen in Section 4.4.2.
4.4.4 Limitations

Stratified SE has limitations due to that the advantages it have are based on a certain level of understanding of the program under test.

There is not a universally-good configuration for Stratified SE. Stratified symbolic execution provides more ease in configuring the search space for an efficient run, but it does not provide a configuration itself. At the current stage of Stratus, it will skip function calls that are skippable, but some of them are not actually suitable for skipping. That is, the skippability of a function call does not necessarily distinguish the callee’s role from the caller; a function call being non-skippable does not necessarily make the callee belong to the same level as the caller.

We have also assumed an application of Stratified SE where the skipped functions are available for analysis. In real-world programs, for many functions this might not be the case, e.g., some of the library functions, or, they might not be suitable for analysis due to complexity, e.g., hash functions. Stratified SE can still skip those functions and introduce over-approximation, but it might not be able to resolve them, which makes it fall back to classic SE. Compared to Stratified SE, there are other technique more suitable for such an application, e.g., S2E [33], which is capable of automatically switching concretisation during the execution of code that is not the analysis target, such as the system kernel, and then switching back to symbolic execution for the analysis target.

Additionally, Stratified symbolic execution allows functions to be skipped and resolved through over-approximation, but it is a costly process. When skipping a function, it will introduce symbolic variables for each argument passed by reference and also for the return value. This will increase the solving time. More importantly, symbolic variables increase the number of states dramatically.

We have discussed how reordering the exploration can be beneficial in Section 4.1.4. This compensates for the negative effects of over-approximation, but it is also dependent on how well functions are chosen for skipping.

Lastly, although we have introduced the solutions to the problems of may-instructions, it nevertheless increases the number of redundant tests.

In Section 4.5, we can see how these limitations affect the performance of stratified symbolic execution, and how it still outruns tradition symbolic execution in some applications in spite of them. Specifically, we also discuss how a function can be identified as suitable for skipping.

4.5 Evaluation

We try to answer the following questions in this section:

- How is stratified symbolic execution’s performance compared to traditional symbolic execution in general, and when is it more efficient or less efficient?

- How is stratified symbolic execution’s performance when we additionally have knowledge about functions to skip or not to skip and how should we make use of this information?

- How effective is using stratified symbolic execution in proving reachability or unreachability?
4.5. Evaluation

We implemented Stratified SE on top of KLEE [26], the state-of-the-art symbolic execution tool. More implementation details and an explanation of the options that are given in Appendix B and Appendix B might help in establishing a concrete understanding of the ideas in the chapter. We compare the results against the original KLEE in the following experiments.

4.5.1 Evaluating Exhaustive Exploration with Automatic Skipping

In this section we show the coverage rates of Stratus and KLEE in exhaustive symbolic execution, i.e., run SE until it fully explores a program or reaches a timeout. In particular, Stratus automatically selects functions to skip (Algorithm 4.3, with basic LLVM and KLEE intrinsics blacklisted).

In this experiment, we use 89 programs (same as the original paper) from Coreutils 6.11 [40] as subjects, and use the same KLEE configuration as well, except that, to achieve determinism in results, we make these adjustments [98]:

- Turn off random branch selection;
- Turn off the Random Path Searcher;
- Make the default weighted searcher identify states using deterministic identifiers instead of pointer values;
- Enable deterministic memory allocation;
- Turn off Address Space Layout Randomisation (ASLR).

The Coreutils programs under experiment are all optimised so as to provide useful function attribute flags (e.g., readonly). By KLEE’s default configuration, some optimisations are already applied.

We measure the coverage rates of the two approaches when given a limited time. The coverage is measured by Gcov and does not consider library code. Symbolic execution and the replaying process were run under limited privilege which might lower the overall coverage. We were not able to obtain meaningful results for all 89 programs, due to four of the runs resulting in runtime errors during symbolic execution using this version of KLEE as well as Stratus, and that two of them were not replayable without causing side effects that may cause inaccuracy. In addition to the one-hour-timeout setup, we also do an experiment with a timeout of 30 minutes, in order to observe how the SE coverage progresses.

---

10Tagged v1.3.0.

11For example, the test cases from cp will used too much disk space and we often have to stop executing them prematurely.
Chapter 4. Stratified Symbolic Execution

Figure 4.15: These two graphs show the comparison between the branch coverage rates. A single data point stands for a program; the x-value stands for the coverage using KLEE; the y-value stands for the coverage using Stratus.

Table 4.1, Figure 4.15a, and Figure 4.15b show a summary of our experiment, including the average coverage rate and the differences in the coverage.

Discussion

From the coverage rate results in Table 4.1 as well as in Figure 4.15a and Figure 4.15b, we can see that although in most of the cases they are comparable, the original KLEE is generally better than Stratus in branch coverage.

Now we make a more detailed inspection into which functions are skipped during the stratified SE process. Figure 4.16 shows the most frequently skipped functions and the number of times they are skipped during our experiments. We find that the total number of times library functions are skipped is much larger than that of user functions. In fact, the top ten functions skipped are all C library functions or operating system interfaces.\(^\text{12}\)

Our hypothesis is then, skipping these kind of functions might have created too much over-approximation which increased the SE cost, more than that can be saved from skipping them. We made more analysis on the worst cases in this experiment and discuss them in Section 4.5.2.

Comparing the coverage rates in the two experiments of different timeouts, we can see that SE made little progress in terms of coverage during the extra 30 minutes (in the one-hour experiment). We observe that it is harder to increase coverage in a later stage of symbolic execution. In fact, KLEE experiences difficulties in progressing through the argument parsing of Coreutils programs and entering the code that contains the main functionality [134]. This indicates that a technique that can quickly get over the parsing and obtain constraint information from the main code is needed. At the current stage, Stratified SE has experienced similar difficulties. However, it can produce some distinct coverage compared to KLEE (Table 4.1), which indicates a clear potential.

\(^\text{12}\)We blacklisted KLEE intrinsics but not the system calls, though some of them should be blacklisted for similar reasons, which is the case in the rest of the evaluation. Here, we demonstrate the results from the most naive skipping strategy.
4.5. Evaluation

Figure 4.16: The ten most skipped functions and the number of times skipped in the two exhaustive experiments. These listed functions are all C library functions or operating system interfaces.

Table 4.2: The worst cases from the evaluation of Stratus with exhaustive exploration and automatic skipping.

<table>
<thead>
<tr>
<th>Program</th>
<th>Rel. cov.</th>
<th>A brief summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(test)</td>
<td>12%</td>
<td>Carry out various tests, e.g., file existence</td>
</tr>
<tr>
<td>stat</td>
<td>41%</td>
<td>Print file status, e.g., modification date</td>
</tr>
<tr>
<td>paste</td>
<td>64%</td>
<td>Merge lines of files</td>
</tr>
<tr>
<td>nl</td>
<td>67%</td>
<td>Number lines of files</td>
</tr>
<tr>
<td>unexpand</td>
<td>68%</td>
<td>Covert spaces to tabs in files or the input stream</td>
</tr>
</tbody>
</table>

4.5.2 Further Analysis

Stratified SE can be used to exhaustively explore programs. However, unlike other symbolic execution tools, there is not a versatile configuration for doing this, making it possibly skipping functions of importance which creates over-approximation, as can be seen in Section 4.5.1. In this section, we take the five worst cases from the previous part of evaluation where Stratus has lower relative coverage compared to KLEE, and study the reasons why the results are as such. These programs can be seen in Table 4.2, where the second column shows the relative coverage (we took the average of the two exhaustive experiment), comparing Stratus to KLEE. We also discuss an intuition of using skipping in this section.

In order to get more deterministic results, we additionally disable the weighted searcher and stratified searcher, and use standard Depth-First Search (DFS)—our goal is not to obtain the best coverage but to understand how KLEE and Stratus work differently. We inspect the reasons for the lower performance on these programs by looking at their designed purposes, their source code structure, the calls in it, and the calls we skipped. By blacklisting
unwanted skips, and whitelisting wanted skips, we can get a performance at least as good as the traditional symbolic execution. We provide a detailed study of the most interesting case we found when redoing the experiment.

This is a program typically used in the conditions of if statements in a script. Given different arguments, it evaluates a string differently, e.g., the -n option checks if a string is empty and the -f option checks if there is a file named that string. In the source code, the program first checks the number of arguments then calls different functions (one_argument, two_argument, three_arguments) to process the different cases. In the previous experiment the skipped call is two_arguments, which is also called by three_arguments. This call leads to a large portion of code and because skipping it means putting the code in a lower priority, symbolic execution did not spend enough time exploring the code in the call.

**stat** This program prints the status of a file. It has a large amount of code formatting the output message according to the input arguments. Three functions have been skipped in the previous experiment: strlen, write, and memcmp. strlen has an important place in parsing the input format argument so that the output is correctly formatted (it is also important in parsing other arguments, discussed in the unexpand case). It is important in the control flow and because of this, skipping it creates lots of redundant states thus the performance is weaker than not skipping. write is a system call and returns value indicating success or errors which will also affect the control flow, but if we do not consider symbolic files created after skipping (see Section 4.5.4), then skipping it should not have significant impact. Recall that skipping a call does not mean we do not visit it. Return values of memcmp also affect the control flow therefore skipping it will affect the coverage, but indefinitely because it is less frequently seen in the code.

**paste** The skipped calls are strlen and write. This is a heavily string/file processing program. These two calls have similar levels of importance in this program as in stat. The performance is weaker due to the generated redundant states.

**nl** This is also a file processing program. Beside strlen and write, in this program we also skipped strcmp and feof_unlocked (tests if a file ends) which can create more redundancy. Specifically, skipping strcmp over-approximates whether two strings equal to each other, so a state might consider two compared strings equal after skipping the call, even when they are not, or vice versa. Skipping feof_unlocked over-approximates whether a file ends. Skipping this call is in our experience not entirely unwanted, because the functions making such call care about only the two possibilities: a file being ended or unended. They do not care about how it ends which is specified in feof_unlocked, so we can put the job of checking out the multiple possibilities of how a file ends in lower priority. The potential cost of skipping is comparably lower to that of other functions like strcmp—intuitively, if we know that two strings are equal or not equal, we can eliminate a lot of states early, but knowing a symbolic file ends or not is not going to help much, because a symbolic file is much less constrained like a string since a program rarely writes to a file (puts constraints on it) and reads from it later. In summary, we want to skip calls that perform tasks less independent on the information from the callers. In this case, there are no actual difference
between Stratus and KLEE after we specifically whitelisting `feof_unlocked` because compared to the overall time consumption, the difference in the cost of before and after altering the priority of this function is small.

**unexpand** The skipped calls in the exhaustive experiments are `write` and `strlen`. The call to `write` is outputting information as discussed before, therefore we think it has no significant impact on the coverage. However, being a string processing program, `strlen` is critical. As discussed earlier, skipping this function means Stratus is temporarily over-approximating how long a string can be and thus introducing a great number of unwanted possibilities. In fact `strlen` is important for most programs that need to parse arguments. `unexpand` has the following call:

```
getopt_long (argc, argv, ",0123456789at:", longopts, NULL)
```

It specifies the acceptable options for this program and calls `strlen` (and similarly other string functions) indirectly. The first time a test containing a valid option in KLEE’s experiment is much earlier than in Stratus’s. In KLEE, the 19th test is `unexpand -0 - -`, which instantly increase the coverage by 8.95% (absolute); in Stratus, the first such test was not produced until the 53th `unexpand - - -`.. Skipping `strlen` postponed the time when the match was successful, which is why the coverage is not as high.

**Redoing the Experiment on unexpand**

To validate our analysis above, we redo the experiment with DFS and whitelisting a single function `unexpand` in program `unexpand`, which is the function doing the conversion. It is called once after the argument parsing, and is one of the most complicated functions in this program. In DFS, once a state reaches the function `unexpand`, it will spawn a lot of states and be stuck in it for the most of the time. By skipping it, we want to prioritise the states that goes into different places of the main function, e.g., parsing the arguments differently or reach an error, e.g., line 9 in Figure 4.17 which shows a snippet of `unexpand` (in this case, the line is not reachable).

The results can be seen in Figure 4.18. Figure 4.18a shows the branch coverage rate’s change over time. Although both of them use DFS, Stratus has a better coverage because it explores different places in the main function by skipping `unexpand`. We can see the first ten tests produced by the two approaches in Figure 4.18b. KLEE dived into the `unexpand` function quickly. The tests it produced differed in the stdin variable from which the function read. The initial tests of Stratus is more spread out in the main function as they are tests distinguishing different cases in the argument parsing process. This case study shows how
important the high-level code is in providing coverage with diversity at an early stage and distinguishing it from the rest is crucial. Compared to Breadth-First Search (BFS), Stratus is similar in that they both try to get a diverse set of test cases, but they are different because BFS deepens in the traditional depth and Stratus deepens in the invocation depth, in other words, the depth of importance and priority, which more precisely captures the role of some particular code. Moreover, we favour an executor compatible with DFS rather than one using BFS due to the possibility to use incremental solving. This is a topic in Section 5.2.2.

Discussion

Through further analysis, we can see that, at least in Coreutils, skippable calls are usually library or system calls, they are sometimes very important in the control flow, and skipping them creates redundancy. In fact, in most programs during the exhaustive experiment we have skipped strlen and write (automatically detected). There are not a lot of calls to user functions skipped. Interestingly, unlike the five worst cases, the five best cases (chcon, ln, base64, mkfifo, mv) are programs relying much less on string manipulation. This suggests that skipping strlen can create more redundancy in string processing programs. Further, we conclude that skipping functions that are highly involved in a programs’ conditionals is not desirable, which makes most string functions unsuitable for skipping. We will discussed desirable skippable calls later in this discussion.

The study of the worst cases also shows that the use of stratified symbolic execution requires careful configuration. In the worst cases where functions suitable for skipping are not immediately clear, we can always return to traditional symbolic execution. However, when we have a skippable function and whose impact after skipping can be clearly seen, we can make use of the skipping technique to aim SE at the code we care the most about.

Here, we can draw a conclusion regarding the first two questions:

- In the general applications of SE, where coverage and efficiency are the two objectives SE aims at and no user guidance is given, Stratified SE is not as competitive as traditional SE due to that it relies on a certain level of understanding of the programs.
- With better selection of functions to skip, Stratified SE has the advantage of searching in the wanted code and achieves desired coverage faster. We can further use Stratified
SE to cover code that is hard to reach by the other techniques, as shown in Section 4.5.1. An intuition of selecting functions is given below.

Regarding the choices of skippable function calls, our intuition is to look for supported functions that are responsible for certain removable functionalities. In other words, the preceding functions in the call chain do not heavily depend on the skipped functions. An example of heavy dependency is the argument parsing function—the execution of the rest of the program depends on it. An example of weak dependency is a file processing function—it may be an important function of a program, but without it, the rest of the program still runs, only that certain files would not be produced at the end of the execution. We will often consider these functions with a definite (as opposed to varying) level of importance. Recall that skipping a function does not always mean we do not verify it, but rather it means we place it in another priority level. Given this idea, we will usually find a suitable skippable function in the user code rather than the library, which again, requires a good understanding of the program under test. Library calls usually achieve small and specific functionalities, and the usage of them is indefinite, i.e., they can appear in different places of a program, associated with different priorities, therefore they are not desirable functions to skip. For library calls that are not likely to cause redundancy, e.g., printf, skipping them (and not resolving them because they are not important to the control flow) can be a good idea, but in this specific case of printf, concretising it (Section 2.3.3) achieves a similar result. In summary, with the intention to design a heuristic for identifying functions to skip in Stratified SE, a skippable function with the following property is potentially desirable: it represents removable or distinctive functionality, appears in a specific location, and is not a library or system call.

A future direction of Stratified SE is to incorporate other kinds of analysis in the selection of skippable functions.

4.5.3 Evaluating Targeted Exploration

In this part of the evaluation we explore how well stratified symbolic execution searches for a path leading to a target, or proving that there is no path leading to a target in a program. Generally speaking, such proving is comparably a harder problem than reaching. The subject programs are from various sources:

(A) Four of these programs used are taken from related work: the first three of them (directedse*) were used to evaluate the performance of directed symbolic execution [90]; the last one (multipass) is an example of the application of symbolic execution in verification of client behaviours in a network (multipass verification) [32].

(B) Another 17 programs are programs from the “recursive-simple”, “bitvector”, and “nt-drivers-simplified” categories of SV-COMP ’17 [122] that have at least one nondeterministic (symbolic) variable.

(C) The last ten programs are randomly chosen from Coreutils [40] as used earlier. We only choose ten because we need to insert reachable and unreachable targets into the programs. This takes a certain level of understanding of the programs. Also, as a result, each program gets multiple variations and in total we have 100 of them in this category.

Each program has one or multiple variations that have small differences in between. The major difference is whether an error line is reachable or not. The error lines in these programs
are the targets that symbolic execution needs to reach or prove unreachable. For programs of Category (A), the targets are the same as the ones used in the original papers. We have also made modifications on these targets to create unreachable variations of them. For programs of Category (B), the targets and the variations are all original. For programs of Category (C), targets are selected by hand and we ensure that they are deep in the main functions and not all states can reach one: for each program we identify a single “significant” function call (except for id we have four) in main based on our understanding of the program, e.g., the call to unexpand in Figure 4.17, which accomplishes the majority of the functionality this program is made for (see Figure 4.19); we whitelist this function, but whether the function is actually skipped still depends on the tool; we do not blacklist any function; we put a conditional assertion immediately behind this call, which is only reachable by some (or none, when proving unreachability) of the paths in the program. To neutralise the bias of paths selection, we create ten variants of each program, allowing different paths reaching the target or different conditionals that make the target unreachable. Using this method, the targets are relatively deep in terms of traditional depth, and programs’ original structures are not affected. An invocation depth limit is given to Stratus indicating the range of search for the targets, e.g., the targets of Category (C) are in the main functions, so we specified 0 as the depth to search for a target, but to reach it or prove it unreachable, Stratus still have to resolve in the deeper functions. To obtain meaningful results instead of time-outs, we chose smaller symbolic variables for each of these programs. The rest of the configuration is similar to that in Section 4.5.1.

We measure the time taken to reach these targets, or the time taken to prove the unreachability of these targets. The timeout for each experiment is 5 minutes except that for the Coreutils it is 20 minutes. For a program having multiple variations, we sum up the time used by the reachable ones and unreachable ones respectively. We repeat symbolic execution 10 times and take the average (of the sums). A summary of the results can be seen in Table 4.3 and Table 4.4. In the tables, $n_v$ is the number of reachable or unreachable variations of the program. The average of the sums of the time taken for them is shown in the table. The numbers to the right of the time data represent the number of time-outs over the $n_v$ variants. For example, KLEE timed out on of 6 of id’s 12 variants when trying to prove target unreachability. If there is a time-out for a variant of a program, then we exclude this variant in the calculation of time data for both approaches, e.g., in the entire id row in Table 4.4, we
**4.5. Evaluation**

Comparing the time to reach targets using Stratus and KLEE.

<table>
<thead>
<tr>
<th>Program</th>
<th>$n_v$</th>
<th>KLEE</th>
<th>Stratus</th>
<th>Rel. time $^1$</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>directeds1</td>
<td>1</td>
<td>0.16 s</td>
<td>0.17 s</td>
<td>103%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>directeds2</td>
<td>1</td>
<td>- 1</td>
<td>14.64 s</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>directeds3</td>
<td>1</td>
<td>- 1</td>
<td>21.78 s</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>multipass</td>
<td>1</td>
<td>1 m 04.94 s</td>
<td>7.62 s</td>
<td>21%</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>

- **id** 14 | 2 m 02.46 s | 1 m 24.68 s | 82% | < 0.01 |
- **sum_non** 2 | 0.04 s | 0.10 s | 143% | < 0.01 |
- **sum02** 1 | - 1 | - 1 | - | - |
- **soft_float** 0 | - 0 | - 0 | - | - |
- **s3_ssrvr** 0 | - 0 | - 0 | - | - |
- **s3_clnt** 3 | 1.28 s | 1.32 s | 102% | 0.08 |
- **parity** 0 | - 0 | - 0 | - | - |
- **num_conversion** 0 | - 0 | - 0 | - | - |
- **modulus** 0 | - 0 | - 0 | - | - |
- **jain** 0 | - 0 | - 0 | - | - |
- **interleave_bits** 0 | - 0 | - 0 | - | - |
- **gcd** 0 | - 0 | - 0 | - | - |
- **byte_add** 1 | 1.75 s | 2.41 s | 116% | < 0.01 |
- **cudaudio** 1 | 0.48 s | 0.24 s | 67% | < 0.01 |
- **diskperf** 0 | - 0 | - 0 | - | - |
- **floppy** 2 | 1.14 s | 1.09 s | 98% | 0.11 |
- **kbfiltr** 1 | 0.13 s | 0.16 s | 110% | < 0.01 |
- **shuf** 5 | 51.37 s | 47.00 s | 96% | 0.60 |
- **ptx** 5 | 1 m 24.22 s | 59.39 s | 83% | 0.02 |
- **stty** 5 | 4 m 41.14 s | 5 m 11.48 s | 105% | 0.58 |
- **seq** 5 | 4 m 59.74 s | 3 m 54.90 s | 88% | < 0.01 |
- **chcon** 5 | 1 m 58.21 s | 2 m 00.13 s | 101% | 0.86 |
- **id** 5 | 21.40 s | 42.77 s | 133% | < 0.01 |
| ] 5 | 23.12 s | 14.92 s | 78% | 0.08 |
- **base64** 5 | 2 m 46.91 s | 2 m 56.98 s | 103% | 0.82 |
- **factor** 5 | 4 m 30.72 s | 2 m 57.97 s | 79% | 0.31 |
- **head** 5 | 6.85 s | 8.01 s | 108% | < 0.01 |

**Summary**  | 50 | 25 m 16.06 s | 5 | 21 m 31.34 s | 2 | 96% | - |

$^1$ Stratus’s result relative to the average of Stratus’s and KLEE’s results.

only consider the 6 variants that have not timed out KLEE. An exception is when either approach cannot produce a result in any variant but the other one can, in which case we show the result from the latter nonetheless, using a greyed out text. In each table’s summary row, we show the total number of variants, total time consumption, total number of time-outs, and the average of relative time results.

Programs of Category (A) and the experiment configurations can be seen in Appendix B. They are also good examples of how stratified symbolic execution performs in targeted exploration.

**Discussion**

From Table 4.3 and Table 4.4, we see that Stratified SE achieved overall better efficiency (15% and 24% absolute time reduction or 4% and 10% relative time reduction on average in the two experiment groups respectively) and fewer time-outs. In some cases, it is significantly faster. Compared to traditional SE, Stratified SE is more efficient to reach or prove the unreachability of targets in high-level code when sufficient information is provided to differentiate code levels. More specifically, we make the following observations on the efficiency.
### Table 4.4: Comparing the time to prove target unreachability using Stratus and KLEE.

<table>
<thead>
<tr>
<th>Program</th>
<th>$n_v$</th>
<th>KLEE</th>
<th>Stratus</th>
<th>Rel. time$^1$</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>directedse1</td>
<td>1</td>
<td>1.09 s 0</td>
<td>0.21 s 0</td>
<td>32%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>directedse2</td>
<td>1</td>
<td>- 1 1 m 22.93 s 0</td>
<td>- -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>directedse3</td>
<td>1</td>
<td>- 1 1 m 16.26 s 0</td>
<td>- -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>multipass</td>
<td>1</td>
<td>1 m 04.51 s 0</td>
<td>7.20 s 0</td>
<td>20%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>id</td>
<td>12</td>
<td>0.10 s 6</td>
<td>2.20 s 0</td>
<td>191%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>sum_non</td>
<td>2</td>
<td>- 2 - 2</td>
<td>- 2 - -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>sum02</td>
<td>1</td>
<td>- 1 - 1</td>
<td>- 1 - -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>soft_float</td>
<td>5</td>
<td>- 5 16 m 48.13 s 0</td>
<td>- -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>s3_srvr</td>
<td>6</td>
<td>0.04 s 5</td>
<td>0.05 s 5</td>
<td>111%</td>
<td>0.02</td>
</tr>
<tr>
<td>s3_clnt</td>
<td>3</td>
<td>2.77 s 0</td>
<td>2.85 s 0</td>
<td>101%</td>
<td>0.03</td>
</tr>
<tr>
<td>parity</td>
<td>1</td>
<td>9.69 s 0</td>
<td>9.50 s 0</td>
<td>99%</td>
<td>0.04</td>
</tr>
<tr>
<td>num_conversion</td>
<td>2</td>
<td>1.50 s 0</td>
<td>1.56 s 0</td>
<td>102%</td>
<td>0.11</td>
</tr>
<tr>
<td>modulus</td>
<td>1</td>
<td>- 1 - 1</td>
<td>- 1 - -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>jain</td>
<td>6</td>
<td>- 6 - 6</td>
<td>- 6 - -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>interleave_bits</td>
<td>1</td>
<td>0.03 s 0</td>
<td>0.03 s 0</td>
<td>100%</td>
<td>0.18</td>
</tr>
<tr>
<td>gcd</td>
<td>4</td>
<td>1 m 01.28 s 0</td>
<td>36.83 s 0</td>
<td>75%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>byte_add</td>
<td>2</td>
<td>1.65 s 0</td>
<td>1.69 s 0</td>
<td>101%</td>
<td>0.02</td>
</tr>
<tr>
<td>cdaudio</td>
<td>1</td>
<td>1.12 s 0</td>
<td>1.13 s 0</td>
<td>100%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>diskperf</td>
<td>1</td>
<td>- 1 - 1</td>
<td>- 1 - -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>floppy</td>
<td>2</td>
<td>1.94 s 0</td>
<td>2.07 s 0</td>
<td>103%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>kbfiltr</td>
<td>2</td>
<td>0.40 s 0</td>
<td>0.41 s 0</td>
<td>101%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>shuf</td>
<td>5</td>
<td>19 m 36.73 s 0</td>
<td>19 m 43.03 s 0</td>
<td>100%</td>
<td>0.24</td>
</tr>
<tr>
<td>ptx</td>
<td>5</td>
<td>- 5 46 m 44.32 s 0</td>
<td>- -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>stty</td>
<td>5</td>
<td>25 m 24.34 s 0</td>
<td>25 m 38.59 s 0</td>
<td>100%</td>
<td>0.19</td>
</tr>
<tr>
<td>seq</td>
<td>5</td>
<td>12 m 50.00 s 0</td>
<td>10 m 22.68 s 0</td>
<td>89%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>chcon</td>
<td>5</td>
<td>25 m 41.48 s 0</td>
<td>21 m 07.93 s 0</td>
<td>90%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>id</td>
<td>5</td>
<td>26 m 43.69 s 0</td>
<td>26 m 51.60 s 0</td>
<td>100%</td>
<td>0.34</td>
</tr>
<tr>
<td>{</td>
<td>5</td>
<td>28 m 59.93 s 0</td>
<td>9.70 s 0</td>
<td>1%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>base64</td>
<td>5</td>
<td>16 m 24.64 s 0</td>
<td>16 m 42.37 s 0</td>
<td>101%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>factor</td>
<td>5</td>
<td>- 5 5.22 s 0</td>
<td>- -</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>head</td>
<td>5</td>
<td>3 m 21.20 s 0</td>
<td>1 m 40.79 s 0</td>
<td>62%</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>106</strong></td>
<td><strong>161 m 28.13 s 0</strong></td>
<td><strong>39 123 m 22.42 s 16</strong></td>
<td><strong>90%</strong></td>
<td><strong>-</strong></td>
</tr>
</tbody>
</table>

$^1$ Stratus's result relative to the average of Stratus's and KLEE's results.
Efficiency depends on the usefulness of information near a target. For programs that have targets inside conditionals, e.g., programs from Category (A) which can be found in Appendix B, the conditionals contain useful information and it is not available before the execution visit them. By skipping functions, stratified SE can reach this information earlier than traditional SE. Such information is very useful for removing irrelevant states at the time it is obtained as well as in the rest of the exploration. This is not always the case. Take program unexpand from Figure 4.17 as an example. Suppose we want symbolic execution to reach the error line 9. By skipping function unexpand, Stratus can obtain the information “have_read_stdin must be True” comparably earlier. It is useful to prune states that do not satisfy this constraint, but it is not useful in pruning states when it explores in the function unexpand later. This is because we have to assume that unexpand function can change have_read_stdin’s value anytime, e.g., right before returning. Therefore, we cannot prune any state when exploring the function using this information before we hit its return. On the other hand, by exploring elsewhere, KLEE might be able to find more useful information than this.

Efficiency requires supervision or learning for selecting skippable functions. Programs from, e.g., “ntdrivers-simplified” (Category (B)) are much less understood by us compared to the other programs. We did not provide any guidance when it comes to choosing skippable functions. That is, the tool chooses the skippable functions on its own. As a result, in most cases, KLEE and Stratus present no major difference. On the other hand, when we can control what to skip or not to skip (if there are skippable functions), and when we can presume the general target location, then we can achieve significant reduction on time consumption or fewer time-outs, e.g., for programs in Category (C).

4.5.4 Threats to Validity

Stratus does not consider the possibilities of a function to create symbolic variables in its parent beside using return values and passing by reference/pointer. This creates problems when global variables or files are also modified by skipped functions and later read by their parents. This essentially creates under-approximation.

In complicated programs where functions are frequently accessing resources that have not been declared through its interface declaration, we will need to consider those possibilities, otherwise the execution of generated tests will frequently diverge from expectation.

In Section 4.5.3, the programs or targets chosen are so that we do not need the information of globals in order to make symbolic execution reach/prove the unreachability of a target, thus this will not invalidate our results.

In the exhaustive evaluation, there are nondeterministic behaviours even after the said methods are used. This is commonly known for adaptations of KLEE. This reduces the confidence in the results.

The scale of the experiment and the size of the program base are limited; as stated before, we avoided programs with globals for the SV-COMP program set; the choice of targets and conditions for Coreutils in Section 4.5.3 tried to avoid biases, but there nonetheless is due to the limited number of variants we can create. The user knowledge we have provided in the experiments is hard to measure and is thus another source of bias.
4.6 Future Work

We discuss potential improvements and applications of Stratified SE here.

4.6.1 Identifying Functions Suitable for Skipping

As discussed in Section 4.5.2, the type of functions skipped is crucial to Stratified SE’s performance. We think it is possible to improve it with a more intelligent function selection process, using static analysis [30] or learning on the functions. As discussed in Section 4.2.2, we currently use no such analysis other than looking at function and parameter attributes.

4.6.2 Seeding with Stratified SE

We could combine Stratified SE with symbolic execution seeding. Seeding is to use an input to direct symbolic execution to a particular part of program to achieve a better start. This is very useful when a test leading to an interesting part of code can be easily constructed by hand. When doing seeding, symbolic execution will first follow the path chosen by the “seed input” and branch states from the execution path of that input. After that, the set of states will be very close to that path. This effectively gives symbolic execution a rough initial direction.

When faced by the high cost of over-approximation and stratified SE, we think about using this technique to generate a small number of tests (seed inputs), so that we can take advantage of stratified symbolic execution’s capability of getting a good grasp at a program’s high-level structure, while ignoring the minor cost difference between it and normal symbolic execution. These seeds could then be used in normal symbolic execution, which is not so good at locating itself in a program’s structure but very dedicated to details. Our intuition is that this can give the symbolic execution faster coverage and make it more reliable.

The potential competitor is the intelligent searchers. As seen from Section 4.3.3, searchers also take into consideration where easy-to-cover instructions are, and have proven to be very efficient. However, they are different and not conflicting approaches. Also a key difference between the algorithms is that, searchers tend to be more greedy, while stratified symbolic execution is neutral.

4.6.3 Skipping Functions in Common Libraries

Another usage of stratified symbolic execution is for dealing with external calls, which is the one of the two obstacles in symbolic execution (the other one is path explosion). Currently, there are mainly three ways to let symbolic execution handle external calls such as library functions. One is substitution—doing symbolic execution on simpler versions of these calls that still capture the essentials of them. The other one is modelling—using a pre-made collection of constraints for each call. A third way is to use concretisation. We have discussed this in Section 2.3.3.

We believe over-approximation can be used (as an additional approach) to handle library calls. As we discussed in Section 4.5.2, it is not recommended to skip library calls because they do not have specific roles, and skipping them creates too much over-approximation, such as malloc (allocating memory and returning an address) and strlen (string operation). However, if a library is commonly used, such as the standard C library, we can analyse
them first to identify functions suitable and not suitable for skipping, and apply this to speed up future Stratified SE activities.

Over-approximation can be combined with substitution or modelling in dealing with library calls. The differences between over-approximation and concretisation (under-approximation) is that it is recoverable and complete in principle (see discussion in Section 4.1).

4.7 Conclusion

In this chapter, we have introduced Stratified Symbolic Execution—a novel symbolic execution approach with the ability to understand and distinguish program layers, that is, the code in a program with different roles. We implemented this idea as Stratus, based on the state-of-the-art SE tool KLEE. Stratus uses skipping and resolving to prioritise the exploration in high-level code. It can use selective resolving to narrow the space of search for a target, and furthermore, prove the correctness of a program iteratively. Additionally, the concepts of invocation depth and skippable function may be more friendly to a tester from the development of a program, compared to the concept of traditional depth and many others in configuring the search space of symbolic execution. With the more accurate instruction from a program tester, Stratified SE can achieve significant performance improvement, especially in targeted exploration, as suggested by our evaluation. Stratified SE can be improved by better function selection methods. It has many promising applications, e.g., to be used as a seeding method.
Chapter 5

Handling Over-Approximation Using Incremental Solving


In the last two chapters, we have seen two approaches of combining symbolic execution with over-approximation. Although over-approximation shows strengths in some situations, it has some inherent drawbacks. It can create too many summaries for summarisation (Section 3.4) and it can make the searcher overestimate a state’s importance (Section 4.3).

In this chapter, we propose the combination of CSE with incremental constraint solving techniques. Specifically, this CSE approach is based on the idea of FGCSE introduced in Chapter 3. In this chapter, we stress the use of an incremental constraint solver and its assumption-based features in CSE, in order to overcome its weakness, i.e., the lack of context during summarisation. We show that the combination of CSE and incremental solving is beneficial, in the sense that CSE can benefit from incremental solving while incremental solving is more suitable for use in CSE than in other SE approaches.

After this, we have another discussion about using an incremental solver in Stratified SE that is introduced in Chapter 4. We envision what can or cannot be thereby improved.

5.1 Introduction

Before we motivate the use of incremental solving in symbolic execution, let us briefly recall the concepts of CSE, FGCSE, and incremental solving. These have been systematically introduced in Chapter 2 and Chapter 3.

5.1.1 Compositional Symbolic Execution

Compositional Symbolic Execution [58, 1, 34, 107, 97] (CSE) is a SE technique that mitigates path explosion by limiting the redundancy that springs from exploring the same part of a program repeatedly. In CSE, parts of the source code are processed prior to (or in an ad hoc way during) SE to produce summaries.

In Chapter 3, we generalised the concept of summaries to describe any arbitrary code fragment (Fine-Grained CSE or FGCSE) instead of entire functions.

In this chapter, we continue to explore the concept of FGCSE. In particular, we are especially interested in how incremental solving techniques can make a difference in it. FGCSE is
Chapter 5. Handling Over-Approximation Using Incremental Solving

Figure 5.1: Example showing the advantage of incremental solving.

a generalisation of typical CSE with function-level summarisation. The conclusions drawn from this chapter apply naturally to other bottom-up CSE approaches. Some of them can apply to SE in general.

The work from this chapter is completed some time after what is presented in Chapter 3. Our initial FGCSE implementation Cirrus was built from scratch but handled programs written in our own C-like programming language. To increase its applicability, we have a more mature implementation based on the LLVM infrastructure [87]. Apart from targeting mainstream programming languages via LLVM, this tool also extends the previous tool with interprocedural analysis, optimised summarisation strategies, and a more sensitive and configurable memory model. It also uses more advanced constraint solver features. This implementation is called Cirrocumulus (see Section 5.3). The results from this chapter also complement those from Chapter 3 in arguing that FGCSE can reduce the number of solver calls used in SE. However, we note that the experiment in Chapter 3 not only provides evidence on FGCSE’s better efficiency but also a comparison of different strategies.

5.1.2 Incremental Constraint Solving

Incremental constraint solving [77] is the ability to solve a constraint by reusing the knowledge from the previous states of the solver, where a subset/superset of the current constraint has been solved. Typically, incremental solvers manage constraints through a stack by pushing and popping. We have seen some discussion in Section 2.2.2.

Incremental solving is of value in SE. Consider symbolically executing the code in Figure 5.1. Suppose the executor visits the code inside the inner if statement. The solver will be asked for feasibility of two constraints: $x > 0$ and $\text{prt} = \text{True}$; in that order. Combining them with the context (that has already generated additional constraints, denoted by context), the solver will first solve context $\land x > 0$ and then context $\land x > 0 \land \text{prt} = \text{True}$. Incremental solving is more efficient than solving en bloc if the second query comes immediately after the first one. Incremental solving would avoid re-learning context $\land x > 0$ more than once. In spite of this, incremental solving is rarely used in CSE [115, 89]. This is detailed next. We call the SE approaches without incremental solving monolithic SE (MonSE).

5.1.3 The Attempts to Use Incremental Solving in SE

The code fragment in Figure 5.1 follows a common pattern. However, most state-of-the-art SE tools do not make use of the incremental solving abilities of modern SMT solvers [89] for two main reasons. First, SE tools often have sophisticated search strategies [26, 24]. This also includes path merging or path searching guided by concrete execution (which makes it concolic). These make symbolic execution jump back and forth in control-flow of the code under test. For the example in Figure 5.1, the tool may jump from the $x > 0$ state to some other unfinished state that has different constraints, to which no previously gained knowledge can apply, negating the value of incremental solving.
The second reason is that there are means of constraint simplification that can (partially) replace incremental solving. Notably, constraint independence checking and caching significantly improves the constraint solving efficiency in SE (see query optimisation in Section 2.3.2), but they are not equivalent to incremental solving. Note that the term incremental solving only refers to the incremental solving functionality that can be enabled or disabled in a solver, not including these optimisations.

Independence checking applies syntactic checking on long constraints and identifies independent constraint components, solving them separately. Solutions to the shorter constraint components can be reused when such constraints are encountered later. However, consider the following constraint on $x, y$ and arrays $a_0, \ldots, a_{32}, b$:

\[
\begin{align*}
a_0 \neq b \\
\land a_1[x] &= b[y] \land \forall i \neq x \ (a_1[i] = a_0[i]) \\
\land a_2[x+1] &= b[y+1] \land \forall i \neq x+1 \ (a_2[i] = a_1[i]) \\
\vdots \\
\land a_{31}[x+30] &= b[y+30] \land \forall i \neq x+30 \ (a_{31}[i] = a_{30}[i]) \\
\land a_0 &= a_{31} \\
\land a_{32}[x+31] &= b[y+31] \land \forall i \neq x+31 \ (a_{32}[i] = a_{31}[i])
\end{align*}
\]

Suppose the constraint needs to be solved once in the middle (at the dashed line), and once after the final assertion is made. Such a constraint cannot be separated into smaller components as the variables are syntactically related. To solve non-incrementally, the assertions above the dashed line need to be solved twice. Note that this constraint has a pattern commonly seen in programs with array copies. It is similar for other constraints with correlated variables. We use this to get significance in our demonstration: we solved it using the above two methods and the same solver in Z3 4.4.2 \[47\] for 1000 times, which cost 80 seconds and 136 seconds respectively. This shows incremental solving can still make use of previously-learned knowledge even if it is from solving constraints correlated to the newly-added constraint.

Constraint caching can also provide a certain extent of incrementality, such as KLEE’s counterexample caching ability \[26\]. Counterexample caching is capable of reasoning about subset/superset relations between new and old sets of constraints, and reusing old solutions to try out the new ones. It serves as an incremental query simplifier for KLEE’s non-incremental solver and was shown to increase efficiency of KLEE. However, incremental solving cannot be replaced by counterexample caching either. Consider solving the following two constraints in order:

\[
\begin{align*}
x > 0 & \quad \text{Cache a candidate solution: } x \mapsto 1 \\
x > 0 \land x > 1 & \quad \text{Cache a candidate solution: } x \mapsto 2
\end{align*}
\]

In this case independence checking is not able to separate the constraints $x > 0$ and $x > 1$ since they both depend on $x$, meaning $x > 0$ is effectively solved with duplicated work.

\[1\] Incremental and non-incremental solving is provided by different underlying solvers in Z3. To remove the solver variable, we use the incremental solver for both methods. In this case, the non-incremental solver takes even longer.
in the second time; counterexample caching cannot guarantee to cache a good candidate of $x$. It might choose to cache 1 after solving the first constraint, but this value does not satisfy the second constraint. So most likely a non-incremental solver will not be as fast as an incremental one in this case.

Moreover, an incremental solver generally performs worse than a non-incremental solver if not solving incrementally, therefore it is not used widely in symbolic execution.

In Section 5.2, we reassess the potential of incremental solving in SE.

5.1.4 Assumption Checking

Assumption checking, i.e., checking with assumptions, is another feature in modern SMT solvers, which also requires incrementality in a solver. Assumptions are a kind of constraints but are different from the usual hard constraints. They are allowed to be retracted by the solver. When a query consists of hard constraints (that are satisfiable) and some assumptions, a solver can provide a reason of failure with respect to the assumptions when this query fails. That is, the solver can try systematically removing assumptions from the set of assertions, and once it finds that the unremoved set of assertions is satisfiable, it obtains a reason, which is called an unsatisfiable core (unsat core), of why the query fails. Compared to the full query with all the assumptions asserted, an unsat core is simpler and cleaner since some assertions have been removed. An unsat core provides more general information about a fact and can be used to subsume a set of symbolic states in SE. This means that this fact (the unsatisfiability) also holds for these states [78]. We have seen its application in Section 2.3.2.

Although similar, related work [76, 78] has not used unsat cores for the correction of bottom-up summarisation, which is an over-approximation, and not compared with other solving techniques. We shall discuss this application of assumption checking in this chapter.

5.1.5 Chapter Outline

We first discuss our motivation for incremental solving. This motivation has two aspects:

- Incremental solving promises great performance gains and provides solution to an important weakness of CSE.
- CSE, using summaries, intuitively is a better host for incremental solving than classic SE.

We then propose a combination of incremental solving and CSE. We present Cirrocumulus based on FGCSE, which is an augmentation and reimplemention of Cirrus. Along with many added functionalities (Section 5.1.1), it enables the use of features available in a modern SMT solver. Finally, we evaluate empirically the efficiency of CSE with/without incremental solving.

5.2 Motivation for Incremental Solving in CSE

We now explain why we have found it worthwhile to reconsider the use of incremental solving in the context of CSE, apart from the raw efficiency gain in constraint solving.
5.2. Motivation for Incremental Solving in CSE

5.2.1 Compensation for the Shortcoming of Summarisation

Summarisation suffers the loss of context and over-approximates the number of subpaths in code fragments. The assumption checking from an incremental solver can compensate for this shortcoming.

The Loss of Context

An obstacle for bottom-up CSE is a lack of context information during summarisation. In Chapter 3, we have discussed this summarisation technique that summarises all paths of a code fragment without assuming a context in order to let the summaries be reusable, even when some of these paths will not eventuate in any given execution of the program that has a certain context. This over-approximation is inherent and has a great impact on CSE’s performance compared with classic SE.

The example in Figure 5.2 illustrates the issue. Consider summarising the function bar before the symbolic execution of function foo—its calling context. The summarisation module will not possess the information from the context. As a result, five subpaths will be found in bar and summarised into five summaries. They present five choices for the global symbolic execution. In the given parent function foo, however, the first argument is a constant, only the second argument is truly symbolic. Given \( x = 10 \), the function bar can only have two feasible paths; the additional three summaries are unnecessary (at least for foo) and introduce extra cost for decision making.

Next we show that incremental solving mitigates this problem.

Assumption Checking

We propose to use a particular technique called assumption checking provided by an incremental solver to address the above problem. We can see that the above three summaries are useless simply because of the constraint \( x < 0 \) they all have. If we can identify the cause of unsatisfiability earlier, and rule out summaries that share it, we can be more efficient in reusing summaries.

An assumption [12] is a type of constraint defined by the user that the solver can assert or retract so as to provide reasons for an “unsat” (a constraint being unsatisfiable). In SMT-LIB, assumptions can only be Boolean constant symbols or their negation, but we can associate a free Boolean constant symbol to any constraint to equivalently make this constraint an assumption.

To explain how it works, we first elaborate a property of the summaries. A summary collects constraints from the subpath of the code fragment it is representing. Constraints
have two origins: normal instructions and conditional instructions. Conditional instructions include, in LLVM IR [87], conditional br, switch, etc., as well as instructions that can cause program failure, e.g., store/load (cause failure when visiting an address that is not allocated for an object). Conditional instructions are those (and only those) that produce a nonempty entry condition in their summary. When a satisfiable set of constraints is conjoined with the entry condition of a summary, the constraints from conditional instructions contained in the entry condition are the only reason that can cause the solver to fail the check. We can formalise the above as a proposition and use it as a basis for applying assumption checking to SE.

**Proposition 3 (Constraints from non-conditionals do not make a path unsat)** Let the set of constraints from normal instructions be $C$ and let the set of constraints from conditionals be $C'$. Let

$$EC(R) = c_1 \land \cdots \land c_m \land c'_1 \land \cdots \land c'_n$$

be the entry condition of an arbitrary path $R$ with $c_i \in C$ ($1 \leq i \leq m$) and $c'_i \in C'$ ($1 \leq i \leq n$). We can rewrite it to

$$EC(R) = \psi \land \neg b_1 \land \cdots \land \neg b_n$$

with $\psi = c_1 \land \cdots \land c_m \land (c'_1 \lor b_1) \land \cdots \land (c'_n \lor b_n)$, and $b_i$ ($1 \leq i \leq n$) being $n$ free Booleans. Then for any symbolic state with $\eta \in \text{Sat}$, we have $\eta \land \psi \in \text{Sat}$, with Sat representing all the states with satisfiable path constraints.

Proposition 3 is basically stating that constraints from $C$ alone can never make a state’s path constraint unsatisfiable, and if there are some constraints that can, one or more of them is from $C'$. Given this, when we need to find the cause of a solver check failure, we naturally think of pushing the $c$s as hard constraints and the $c'$s as assumptions (with the help of few free Booleans).

Let us return to the example in Figure 5.2 to see how assumption mode works. We have listed the five summaries for function $\text{bar}$ in Figure 5.3. We marked the constraints from summaries with rectangles that cross them: rectangles with a dashed border for constraints from conditional statements—later to become the assumptions, and the others rectangles for
the constraints from non-conditionals—later become the hard constraints. The subscript of a variable is its version number explained in Section 3.3.1. As can be seen, the summaries always have constraints in common, including those from conditional statements. Continuing the example in Figure 5.2, the symbolic execution state with the context information \( x = 10 \) realises that summary B is unsatisfiable after a solver check:

\[
x_1 = 10 \land y_1 = Y \land EC(R_B)
\]

i.e.,

\[
(x_2 = -x_1 \land ret_1 = y_1/x_2) \land (x_1 \neq 0 \land x_1 < 0 \land y_1 > 0 \land y_1 < 10).
\]

We use \( Y \) to denote a symbolic variable, and \( EC(R_B) \) to denote the entry condition of the path represented by summary B. Again, the version alignment is discussed in Section 3.3.1. To help readers see clearly, we also group the constraints from non-conditionals and conditionals separately in two pairs of parentheses. At this point the state has not checked summaries C, D, and E yet.

If the assumption mode is on, the symbolic execution will receive a reason why summary B fails—in our case \( x_1 < 0 \), as this constraint alone can cause the query (which has the context \( x_1 = 10 \)) to fail, regardless of the others. The constraint \( x_1 < 0 \) and the hard constraints from the context constitute an unsatisfiable core (unsat core). We map a constraint to its source and to the summaries that contain it, so that we can still identify the similarity between summaries after the alignment. We can see that in this case summary C contains this unsat core too, and we know that it alone will make the query fail in the current context. So we can prune summary C directly for this context, rather than search in that direction. Similarly, if the solver finds the unsat core to be \( y_1 > 0 \land y_1 < 100 \) within another context, we can prune summary D.

We note that an unsat core can contain more than one assumption and it is not necessarily minimal, which requires more similarities between summaries for pruning. That said, the efficiency of the assumption utilities depends on the underlying solver. Large unsat cores are less favoured than small and minimal ones. Another issue is the selection of unsat cores, as there may be more than one. The selection also affects the effectiveness of pruning.

### 5.2.2 The Potential Value of Summaries

A main reason for not using incremental solving in SE is its conflict with best-first searches. We now show the theoretical possibility of achieving locally best-first searching in CSE without the use of these searchers, thus mitigating the potential downside of using incremental solving. We shall loosely refer to path constraints that share a significant prefix of conjuncts as being consonant. Consonance is not assured when we use best-first or even breadth-first searchers, because these searchers can jump between totally unrelated states and produce totally unrelated constraints.

Summaries can provide information of the code being visited, thus encourage wiser decisions in searching, which have to be achieved by sophisticated searchers otherwise. For example, KLEE [26] has a Weighted Random Searcher that prioritises states by using measures of weight like depth, solver query cost, minimum distance to uncover new instructions, etc. Searcher heuristics, i.e., the different ways to score a state, are introduced in Section 2.3.2. For simplicity, let us consider a searcher that prioritises states according to the number of uncovered non-controls it can potentially cover. We denote such a searcher by \( W \).
Chapter 5. Handling Over-Approximation Using Incremental Solving

... for (;;) {
    if (ch >= 32) {
        if (ch < 127) {
            // subpath A
            *bpout++ = ch;
        } else if (ch == 127) {
            // subpath B
            *bpout++ = '^';
            *bpout++ = '?';
        } else {
            // subpath C
        ...
            // updating *bpout and bpout
    } else if (ch == '	' && !show_tabs) {
        // D
        *bpout++ = '	';
    } else if (ch == '
') {
        // subpath E
        newlines = -1;
        break;
    } else {
        // subpath F
        *bpout++ = '^';
        *bpout++ = ch + 64;
    }
    ch = *bpin++;
} ...

Figure 5.4: A code fragment (simplified) in cat from Coreutils.

To see how a summary-aided searcher achieves a result similar to the weighted searcher $W$, consider the example in Figure 5.4. Here, we only need to focus on the code structure and leave the semantics. For simplicity, let us limit the scope of the exploration and prioritisation to within the loop body. At each branch, the two choices, represented by two states, will have two scores associated with them. For searcher $W$, the score equals the number of uncovered non-controllers on the path from the current code location, to the end of the loop body. If there are multiple paths to the end of the loop body, then the numbers add up. For example, at the conditional of line 3, going down the true branch potentially covers line 5, 7, and 8 (we omit line 10 for simplicity). So the true branch gets a score 3. All the scores are calculated and shown in Figure 5.5 (top left).

Given these scores, at each SE step, $W$ will choose to execute whichever state has the highest score. This is illustrated using a line with an arrowhead in Figure 5.5 (top right). $W$ makes jumps like the ones marked with dashed lines, as opposed to the dotted lines. The jumps can also be made to any unfinished state, not limited to states in the current loop body. This imposes context-switching on the solver. Hence weighted search is less suitable for incremental solving. Nevertheless, this searcher produces the optimal result, as shown under the figure, putting all paths with score 2 over those with score 1.

Simple weighted DFS can be implemented to explore the same code (bottom left in Figure 5.5). Compared with $W$, DFS-like search algorithms provide the necessary consonance between path constraints to make incremental solving useful. That is, at each branch, SE issues a query to the solver that is similar to the last one, because it does not make long jumps like $W$ does.

DFS processes the code fragment in a suboptimal way (putting subpath D before B). The bottom right corner of Figure 5.5 shows how DFS can be improved by summaries. As six summaries and the weight information has already been collected in the summarisation stage, there is no need to make decisions in the middle of the code fragment during the symbolic execution phase. This feature is has been discussed in Chapter 3. The interesting thing is that we can order the summaries to reflect the prioritisation of different subpaths, which gives us the same result as searcher $W$, and still allows us to use incremental solving,
Figure 5.5: Searchers’ paradigms and how they affect search results and query consonance. The top left figure is the execution tree of the loop body of the example in Figure 5.4. The number beside a branch is the score of a symbolic state that takes the particular branch, given by a heuristic that count the number of non-controls it can possibly cover. The other figures are a comparison of weighted searcher \( W \) (top right), weighted DFS (bottom left), and weighted DFS with summaries (bottom right). The first few steps of the searchers are illustrated using lines with arrowheads. Below each of them is the order of visiting different paths in the loop. In this case, \( W \) represents the optimal results as it puts paths with higher scores before the ones with lower scores.
because this searcher only backtracks within the summarised code block. In other words, compared to $W$ and DFS, it makes medium-range jumps. The example shows how CSE is better at regional prioritisation while still using incremental-solving-compatible searchers.

5.3 Implementation

In this section, we introduce our implementation of CSE with the support of an incremental solver, which we call Cirrocumulus.

5.3.1 A More Sophisticated Implementation of FGCSE

As mentioned in the former part of the chapter, we are using an FGCSE tool implemented separately with Cirrus from Chapter 3. Cirrocumulus is built on the LLVM [87] infrastructure, making it understand LLVM IR and consequently other languages that can be compiled into it. There are a few instruction types or usages not yet supported, e.g., calling with function pointers, but any unhandled instructions will create imprecision which will be reflected by the coverage rates through our experiment. We summarise the additional non-solver-related features in this section.

Summarisation Strategy

As discussed in Section 3.4, CSE should have a strategy to select targets for summarisation. In Cirrocumulus the summarisation module identifies code fragments suitable using the following strategy: function bodies or loop bodies will be summarised unless they are interrupted by function calls or loops, in which case the code before and after the interruption will be summarised separately. This makes any code fragment we choose to summarise acyclic, similar to the LASUM strategy introduced in Section 3.4.2 (with an extension to support interprocedural analysis), which achieves the best/most stable overall efficiency. For example, the loop body in Figure 5.4 will be summarised.

Modelling the Memory in CSE

The importance of modelling the memory in SE is twofold. First, obscure program faults often relate to memory operations (e.g., pointers out of bounds) and only a sufficiently precise memory model can capture these. Secondly, constraints produced by the memory operations are usually more complex and time consuming than those produced by other operations.

Our memory model in Cirrocumulus is similar to existing models [28, 51, 140]. We use an array (in a supporting constraint solver Z3 [47]) to represent the memory, and array constraints to shadow memory operations. In summary, the memory model is

- At variable level instead of bit or byte level—we minimise the number of array constraints needed to do a memory operation on a variable of Single Value Type [87] to one.
- Typed—we use a bit-vector array to model the integers of different sizes and a Real array to model the floating-points.
- Bounded—the allocated address space for each variable is recorded (with write concretisation [29, 120]) and used to check memory violation.
While this approach is not entirely new to memory modelling, there are differences for CSE. Classic SE typically keeps a shadow memory in each symbolic execution state and evolves it along with the state [28, 26]. It tracks the relation between the symbolic input and the current memory state, which CSE also does in the symbolic execution phase. In the summarisation phase, however, it is the relation between the regional “symbolic inputs” and the consequent memory state that is modelled. Also, for a code fragment, the memory may be undefined as well. Therefore, the memory is assumed to be symbolic and treated as versioned variables just as the other variables.

Memory versioning is briefly mentioned in Section 3.3.1. However, it is a simplified memory model, which is not able to capture as many types of program behaviours as the one in this chapter can. Here we should note that having a nearly fully-featured symbolic memory is costly, sometimes not worth the additional types of errors it allows to capture, compared to a simplified memory model. We think the current memory model is still too heavyweight, which makes it less practical, as shown in the evaluation (Section 5.4). For example, we use a single array variable (with adjustable size) to represent the memory. Compared to using multiple “memory objects” (effectively using multiple smaller arrays) as in KLEE [26], it allows to additionally capture the relations between different memory allocations, but is much more expensive. The latter, which is also called logical memory models [8], is more scalable in practice.

5.3.2 Integration of Incremental Solving into (Compositional) SE

CSE communicates with the constraint solver through the constraint solving interface. In Cirrocumulus, a state (and its substates) in CSE is always interacting with a unique SMT solver instance, to benefit from incremental solving, in contrast to using multiple instances or a portfolio solver [98]. We are not aware of an SMT solver that can share information learned from previous checks further than within a single instance. The interface provides three kinds of utilities that are necessary for Cirrocumulus.

Common Utilities

This includes the basic functions that allow symbolic execution to add constraints to the solver, check the satisfiability of the constraints, get a model after a “sat”, etc. Note that the insertion of the constraints should be done incrementally. Recall the example from Section 5.1.2, elaborated in Figure 5.6. We have marked out each program location where symbolic execution needs to interact with its solver.

Here we add the constraints from the long piece of code (context) only once at the point labelled with “A”. At point B or C, we never need to add them again. Note that this requires...
incrementality in the management of constraints, but not the incrementality in solving them, which is to be introduced next.

**Incremental Solving Utilities**

This includes the push—creating backtracking point, and the pop—solver backtracking functions. Take Figure 5.6 for example, after adding the constraints at point A, we immediately create a backtracking point as well. The backtracking point remembers the current state of the solver. Later, when we are done exploring the true branch of the first if statement, we can use solver backtracking to restore the solver to the exact state recorded in that backtracking point, so as to explore the other branch. Note that non-incremental solvers might use push and pop as well, but they are shortcuts for resetting and making the same assertions again rather than reusing previously built lemmas.

**Assumption Utilities**

The assumption mode uses the assumption interface provided by the solver. In addition to invoking the solver for normal checking, symbolic execution also needs to specify the different types of constraints, namely which are hard constraints, and which are removable assumptions. As already stated, we mark the constraints from conditional instructions as assumptions and the rest as hard constraints. The assumption utilities also need to retrieve unsatisfiable cores (unsat cores) for symbolic execution when the solver finds any summary unsatisfiable, and do the subsequent similarity checking between summaries. Cirrocumulus uses unique identifiers for constraints and can identify the locations the constraints are from so as to map them to corresponding summaries.

### 5.3.3 Limitations

Although we have good reasons to use incremental solving and we see a great fit between incremental solving and summarisation, there are drawbacks as well.

**DFS Is Not Fully Prioritised**

DFS with Summarisation is a compromise between DFS and fully prioritised searchers. On the one hand, it weights different subpaths more precisely than DFS within a certain scope. On the other hand, it cannot do as well as a fully prioritised searcher at the global scope. Summarisation gives symbolic execution more flexible choices, but it is still depth-first: it has to finish one choice before it switches to another one, and that one can only be chosen from the last choice point.

Moreover, DFS searchers are not suitable for targeted search, and neither is incremental solving, consequently. However, other search algorithms that are not breadth- or best-first could be used while still exploiting incremental solving.

**The Cost of Incremental Solving**

Each call to an incremental solver is believed more expensive than a non-incremental one in general, and much more so in the assumption mode. However, if used intelligently, it can benefit CSE.
5.4 Evaluation

We ask the following questions in our experiment:

1. What is the impact of incremental solving on CSE and how is the efficiency of CSE with incremental solving compared to without?

2. To what extent can the assumption mode mitigate the problem of the loss of context in CSE?

We focus on the solver utilisation efficiency since constraint solving is the dominating activity in symbolic execution [19].

5.4.1 Setup

Our CSE tool Cirrocumulus uses incremental solving, including the assumption mode. In the experiment we denote this configuration by CSE-IA (CSE with Incremental solving and Assumption checking). Constraint solving is done with Z3 4.4.2 [47]. We also evaluated two other versions of Cirrocumulus: configurations CSE-I (CSE with Incremental solving but without assumption checking) and CSE-NI (CSE with No Incremental solving and consequently no assumption checking).

We compare the number of solver calls used by different tools to achieve the same coverage up to the same depth in a set of programs. We also measure the time elapsed in solver invocation to understand the difference made by incremental solving. Note that all of our statistics include only the solver invocation during the execution of branches: calls to the solver for memory violation checks or variable concretisation are not counted.

The options used are listed in Figure 5.7 for completeness. We briefly explain the meanings here:

- **-simp-unsat-core** means to use assumption mode to simplify the summaries for a piece of code, which is only set for CSE-IA.

- **-inc-solve** enables incremental solving (including the assumption mode), which is unset for CSE-NI.

- **-solver-logic** lets Z3 make the solver from a specific logic instead of automatically choosing.

- **-assumpt-limit** sets the minimum number of assumptions from a summary (in this experiment, this is the number of constraints from conditionals in a summary) in order to enable the assumption mode, as we do not want to use it for short summaries for efficiency consideration.

- **-cell-size** and **-ptr-size** are configuring the memory model.

- The other options are straightforward to understand.

Additionally, each program has a five-minute timeout.

We used the Verisec Suite 0.2 [85], which contains close to 300 segments of open-source programs. Each of these segments demonstrates either a known buffer overflow vulnerability or a corresponding fix. We selected 283 of these (the others each contains at least one

\[2\] More specifically, it uses logic “QF_ABV” [118]. This is experimental and unset in our experiment. Therefore the solver is not specialised to a certain logic.
Chapter 5. Handling Over-Approximation Using Incremental Solving

```c
-cse-simp-unsat-core=<CSE-IA?1:0> -inc-solve=<CSE-NI?0:1> \
-solver-logic=0 -assumpt-limit=4 -solver-timeout=5 \
-cell-size=32 -ptr-size=16 -depth-limit=15 <prog>
```

**Figure 5.7:** Configurations used in the comparison the multiple variations of CSE.

**Table 5.1:** Summary of the experiment on CSE with incremental solving.

<table>
<thead>
<tr>
<th></th>
<th>For 231 in 283</th>
<th>For 266 in 283</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solving time (s)</td>
<td>Branch coverage</td>
</tr>
<tr>
<td>CSE-NI(^1)</td>
<td>52/283</td>
<td>8.318</td>
</tr>
<tr>
<td>CSE-I(^2)</td>
<td>12/283</td>
<td>4.582</td>
</tr>
<tr>
<td>CSE-IA(^3)</td>
<td>16/283</td>
<td>4.666</td>
</tr>
</tbody>
</table>

\(^1\) CSE without incremental solving  
\(^2\) CSE with incremental solving (without assumption checking)  
\(^3\) CSE with incremental solving and assumption checking both on

library call that cannot be resolved to a stub provided by Verisec). Cirrocumulus uses the clang front end (Cirrocumulus is based on LLVM 3.8 [87]). The default -O0 option is assumed. Evaluation is performed on a laptop with Intel Core i7 CPU @ 2.20 GHz \( \times 8 \), with 8 GB of memory.

### 5.4.2 Results

For the convenience of comparison, we collate two sets of results: we first present results on 231 programs which did not time out on any one of the experiment subjects; then we present results on 266 programs that timed out on neither CSE-I, nor CSE-IA. These summaries can be found Table 5.1, which includes the time taken by solver calls and the resulting branch coverage. Note that the constraint solver time usually dominates the symbolic execution time. Since we do not account for all types of solver calls as explained in Section 5.4.1, we do not show the total execution time here. We also specifically compare the solver invocation frequency of CSE-I and CSE-IA on those 266 programs in Figure 5.8. We use the number of solver calls of a monolithic, non-compositional SE tool under the same experiment configuration as a reference of the program’s complexity.

### 5.4.3 Discussions

We make three observations from the experiment which answer our three questions respectively.

**Incremental solving increases the efficiency of CSE** We can draw the conclusion by comparing CSE-IA/CSE-I with CSE-NI using the data in Table 5.1: using non-incremental solving, CSE had significantly more time-outs than using incremental solving; for the programs that CSE-IA/CSE-I and CSE-NI both finished, the latter cost around 1.8\( \times \) the original time.

**CSE uses fewer solver calls on more “complex” programs** We continue to observe efficiency gain provided by summarisation as we have seen in Chapter 3. If we consider the complexity of the program in terms of how much it costs for classic SE to explore it (which
5.4. Evaluation

![Comparison of CSE with and without assumption checking.](image)

**Figure 5.8:** Comparison of CSE with and without assumption checking. The x-axis is the number of solver calls used by a classic monolithic SE tool (referred to as MonSE) as a reference of program complexity; the y-axis is the difference in the number of solver calls between MonSE and a CSE variation, expressed in the form of $\text{diff}(x, y)$ which is defined as $\left(\text{Number of solver calls for } x - \text{Number of solver calls for } y\right)$.

means the programs on the left in Figure 5.8 are comparably easier than those on the right), we can see that CSE performs similarly to SE on the simple programs (with cases where it is worse due to the overhead of summarisation) and performs better on the more complicated ones. Both CSE-I and CSE-IA have significant improvement over SE in terms of solver utilisation when it comes to the rightmost area, where the constraint solving is most intensive.

**The assumption mode is effective but needs more support from the underlying solver**

From the comparison of CSE-IA and CSE-I in the bottom graph in Figure 5.8 we can see that the assumption mode constantly reduces solver invocation frequency by a noticeable number, with few exceptions. The assumption mode not only increases the efficiency of CSE on complicated programs, but also the tolerance of over-approximation on the programs that CSE performs badly. However, the overall time consumption of CSE-IA compared with CSE-I does not reduce in accordance with the number of solver calls as seen from Table 5.1. This is because the calls to a solver with assumption checking enabled are usually more expensive (among 63,817 solver calls that CSE-IA made, 922 were issued with assumptions to check). It would be very useful if more tuning and configurable options were available from the solver. For example, in our case, small unsat cores are more useful than larger ones. If the solver provides options to choose unsat cores or stop trying if it cannot find an unsat core within a size limit, then we could potentially benefit from this feature in more ways.

**Additional observation on the summarisation overhead**

A reason why CSE is less efficient in simple programs is the summarisation overhead. Such programs are usually for simple purposes only and have simple constructs. The summarisation strategy is being conservative and often fails to make useful summaries when it frequently sees delimiters, i.e.,
Chapter 5. Handling Over-Approximation Using Incremental Solving

indicators of the end of summarisation in FGCSE such as a function return or loop latch. This wastes the summarisation work. In other words, overly short functions or loops are not suitable for summarisation. One direction of improvement is to continue the summarisation when we see delimiters, but, in order to prevent introducing large summarisation overhead, we need to be selective about subpaths we summarise. For example, we could restrict the number of loop iterations and the size of the call stacks. This results in an incomplete summarisation, but is possibly a worthwhile trade-off.

5.5 Incremental Solving in Stratified Symbolic Execution

Conceptually speaking, incremental solving could increase the general constraint solving speed in stratified symbolic execution. Moreover, Stratified SE can improve the consonance of queries because constraints in similar levels are put closer, or in the simplest cases, constraints in the same function are put closer without the interruption of constraints from functions of larger depths. On the other hand, the assumption checking can also provide correction for the over-approximation during Stratified SE, in the same way (see Section 5.2.1) it does for CSE.

However, heuristic searchers are still of vital importance in stratified symbolic execution, especially in targeted exploration as we propose in Section 4.4.2. This means incremental solving is not suitable for some of the applications of stratified symbolic execution.

5.6 Conclusion

This chapter proposes the combination of over-approximation techniques and incremental solving, in particular, CSE and incremental solving. We have implemented fine-grained CSE for LLVM IR, in the form of Cirrocumulus. We equipped it with the ability to use incremental solving. We compared its time efficiency with incremental solving turn on/off. We found in the experiment that incremental solving is more efficient than non-incremental solving in CSE. We attempted to use incremental solving and especially assumption checking to overcome the weakness of context loss in CSE, and it showed sufficient potential that it deserves to be reconsidered in SE.
Chapter 6

Conclusion and Future Work

Symbolic Execution (SE) is an automation technique for program verification or testing. It is hard to scale SE to large programs because of the problems brought by path explosion. Approximation, especially under-approximation, is a popular solution yet existing techniques have not covered all aspects of these problems. In this thesis, we investigated the use of over-approximation in SE, and identified possible improvements that can be made on this kind of technique. We showed that, with our novel contributions, they can be used to improve SE’s scalability.

First, in Chapter 3, we generalised the idea of Compositional Symbolic Execution (CSE) by allowing code fragments to be summarised in addition to functions, which can be combined with the latter to achieve better efficiency. We call this Fine-Grained CSE (FGCSE). FGCSE can precompute summaries from code fragments before SE, which are reusable during SE. FGCSE is over-approximating because the summaries are context-insensitive. We discovered that choosing different code to summarise reveals different efficiencies, because they produce different numbers of feasible and infeasible summaries depending on how correlated the summarised code and the context are. We designed three strategies to choose summarisation targets. They mainly differ in how they treat the conditionals. Loop-Body Summarisation (LSUM) summarises a loop body, unrolling nested loops in it; Loop-Body-Acyclic Summarisation (LASUM) summarises a loop but will stop before a nested loop and reinitiate summarisation after it; Small-Step Summarisation (SSUM) summarises consecutive if-then-else statements. We implemented this idea as Cirrus, an experimental tool. We used backward reasoning in Cirrus, but it does not mean FGCSE is limited to a certain direction. We compare the three strategies with traditional backward SE, and discovered improvement achieved by reducing the number of constraint solver calls.

FGCSE has the highest granularity in CSE. It is also the first work that considers optimising summarisation results using different strategies. A future direction of FGCSE is to come up with better summarisation strategies. We can potentially use basic data-flow analysis to discover relationship between code fragments, and choose less dependent parts for separate summarisation. Other heuristics can also be used, similar to Chakrabarti and Godefroid [30]. Another possible direction aims to identify the most frequently executed code. We can first profile the code using a high-coverage test suite and see the most repeatedly executed code. Then we can use this as an indicator to suggest summarisation targets. This might work for CSE techniques in general.

In Chapter 4, we identified a second way of over-approximating and achieving improvement is through Stratified SE. In contrast to FGCSE which over-approximates the context, Stratified SE over-approximates the details of calls. We implemented this idea on top of
KLEE [26], as Stratus. Stratus has many advantages, namely, it gives the user ways to differentiate the roles of code according to its importance in generating tests, and based on which selectively and iteratively performs SE; it can retrieve some critical information faster to reach a target, or to prove a target unreachable. Compared to the traditional SE approaches, in certain cases it is more likely to obtain a proof (although it is partial) because it does not have to explore the whole program. We show these by experimenting on Stratus and KLEE. During exhaustive SE without any user input to differentiate the code, Stratus’s performance cannot match that of KLEE. In more a detailed study, we provided the reasons why and discuss how to apply Stratus according to needs. We then experimented on three different sets of programs with reachable and unreachable targets, where Stratus outperformed KLEE by reaching targets or proving the unreachability significantly faster.

Stratified SE brings a new way of configuring the search space which not only facilitates the deployment of SE but also improve its efficiency in certain applications. The limitations of this work include automatically identifying the difference in the roles of code. Again, we think that static analysis, heuristics, and learning can be used, in a way similar to what we discussed for FGCSE. Stratified SE has many potential applications. For example, we can generate seeds with Stratified SE. Seeding is to find initial inputs that suggest broader search directions for SE to achieve high coverage sooner. Stratified SE, being able to focus the search in high-level code, can potentially provide good seeds for traditional SE that is of a generally lower cost.

Finally, in Chapter 5, we showed that incremental solving, a particular solving technique that can reuse results from previous solving activities, can be used in combination with over-approximation techniques. Incremental solving not only increases the solving speed, when combined with CSE, it also shows other valuable qualities. In particular, a technique related to incremental solving is assumption checking, which retrieves unsatisfiable cores that can be used to identify infeasible summaries efficiently. We implemented another FGCSE tool, Cirrocumulus, based on LLVM [87]. We showed that incremental solving significantly increases the SE speed if we use a compatible searcher. We also showed that assumption checking improves SE, while it can use more configurable options such as specifying the maximum size of unsatisfiable cores. For the other over-approximation approach, stratified SE, we think incremental solving and assumption checking is also applicable. However, heuristics are important to stratified SE in the applications such as targeted SE, for which reason using incremental solving can be less efficient.

Our integration of incremental solving in SE is of value to both the SE community where the utilisation of solvers is a critical topic, and the constraint solver community in that it shows the expectancy from a particular kind of user.

A general limitation in our work is that the conclusions from this thesis are not generalised for concolic execution. Concolic execution is a concept of combining concrete execution and symbolic execution. We do, however, think that some techniques such as FGCSE and summarisation strategies can be used in combination with it, which is another future direction.

**Final thoughts** We investigated the application of over-approximation in SE which is understudied. We conclude that over-approximation is a useful addition to SE in that it can be used to create reusable constraints to avoid redundancy or prioritise SE. When designing a scalable SE tool, we should consider it as a complement to other optimisation techniques.
Appendix A

Symbolic Execution Tree

We have seen different ways of doing symbolic execution and different ideas, in particular, using over-approximation to overcome the obstacles in symbolic execution. In this chapter, we give a definition of symbolic execution tree, and based on this we give definitions of splitting, merging, under-approximation and over-approximation on symbolic execution trees, which are the defining features of SE approaches. We use this mainly as a tool to approximate and discuss the size of their search space. We abstract an SE approach from its implementation details (underlying infrastructure, target language, evaluation of particular expressions, etc.). We also do not distinguish the order of exploration (prioritisation). In this way, we can capture the size of the search space (the number of decision making points) for most SE approaches. The limitations are that we do not capture the difference between individual constraint solving activities\footnote{With query optimisations and incremental solving, individual solving cost becomes more similar; with techniques like state merging, individual solving cost differs. See Chapter 2 for more detail.}, and that it is not yet generalised for DSE.

The main chapters use some notations and terminology defined in this chapter.

A.1 Tree and Structural Tree Relation

We lay down the definition of a tree first. This definition should be standard in the sense that it is close to the tree data structure in computer science. Here, we will emphasise on the concept of reachability, the subtree relation, and an additional kind of tree relation: approximation.

A.1.1 Node and Edge

A node is the smallest topological component of a tree. An edge has the form \((u,v)\) with \(u, v\) being two nodes. We consider only directed edges, i.e., \((u,v) = (v,u) \Rightarrow u = v\). We say \((u,v)\) is an out-edge of \(u\) and in-edge of \(v\).

For a set of edges \(E\), \(\text{nodes}(E)\) is the set of nodes in \(E\):

\[
\text{nodes}(E) = \bigcup \{\{u,v\} \mid (u, v) \in E\}
\]

A.1.2 Path

A path \(R\) is a sequence of nodes \(u_1, u_2, \ldots\). It is possibly infinite. A path corresponds to a set of edges:

\[
\text{edges}(R) = \begin{cases} 
\{(u_i, u_{i+1}) \mid 1 \leq i < k\} & R = u_1, \ldots, u_k \\
\{(u_i, u_{i+1}) \mid i \in \mathbb{N}_1\} & R = u_1, u_2, \ldots
\end{cases}
\]
Appendix A. Symbolic Execution Tree

By definition, a path of a single node corresponds to an empty set of edges.

A.1.3 Reachability

Given a finite path \( R = u, \ldots, v \), we say \( u \) can reach \( v \) via \( R \). This is denoted by \( u \xrightarrow{R} v \).

We can define reachability over a set of edges \( E \). We say \( u \) can reach \( v \) via \( E \) if and only if \( u \) can reach \( v \) via a path \( R \) that corresponds to a subset of \( E \). Without ambiguity, this is denoted by \( u \xrightarrow{E} v \). The above is formally put as

\[
\text{If } u \xrightarrow{E} v \text{ then } \exists R (u \xrightarrow{R} v \land \text{edges}(R) \subseteq E)
\]

By definition, any node can reach itself via an empty set of edges, i.e., \( n \xrightarrow{\emptyset} n \), which means it is reflexive. It is also transitive, meaning \( u \xrightarrow{E_1} v \land u \xrightarrow{E_2} w \Rightarrow u \xrightarrow{E_1 \cup E_2} w \).

The set of paths between two nodes via a set of edges is denoted by

\[
\text{paths}_E(u, v) = \{ R \mid u \xrightarrow{R} v \land \text{edges}(R) \subseteq E \}
\]

This is the set of ways to “go” from \( u \) to \( v \) in \( E \).

Reachability is only discussed over finite paths, which means we do not discuss what a node can reach given infinite “steps”.

When there is no ambiguity of what the path \( R \) or the set of edges \( E \) is, “\( u \) can reach \( v \)” means “\( u \) can reach \( v \) via \( R/E \)”.

A.1.4 Tree

In this thesis, a tree is a commonly-understood rooted tree. It is a tuple \( (N, E, n) \), where \( N \) and \( E \) is the set of nodes and edges in this tree, and \( n \) is the root of the tree.

Some notations we use on a tree \( T = (N, E, n) \) are listed below:

- \( \text{root}(T) = n \) The unique root of the tree
- \( \text{nodes}(T) = N \) The set of nodes in the tree
- \( \text{children}(u) = \{ v \mid (u, v) \in E \} \) The child/children of a node in the tree
- \( \text{leaves}(T) = \{ u \mid u \in N \land \text{children}(u) = \emptyset \} \) The leaf/leaves of the tree
- \( \text{path}(T, u) = R \ (R \in \text{paths}_E(n, u)) \) The unique path from \( n \) to \( u \) in the tree

By common definition, a tree \( T = (N, E, n) \) satisfies:

\[
N = \text{nodes}(E) \cup \{ n \} \land \\
\forall u \in N \ (n \xrightarrow{E} u) \land \\
\forall u \in N \ (|\text{paths}_E(n, u)| = 1)
\]

We assume that \( N \) is nonempty, in other words, the tree is nonempty. We also assume that \( N, E \) are possibly infinite, that is, the tree can be infinite. The set of nodes in \( E \) is equal to \( N \), except that when \( E \) is empty, \( N \) can have a single element \( n \).
For tree $T = (N, E, n)$, the root $n$ can reach any node in the tree via $E$, which makes sure it is a connected graph. Path uniqueness requires that there is no more than one in-edges for each node and there is no loop.

A node can always reach a leaf in the tree via $E$ if the tree is finite, but not always when the tree is infinite. Infinite tree might not even have a leaf.

A path $R$ can also be seen as a special case of tree defined above, i.e., $R = (N, E, n)$. It is a tree but each node has at most one out-edge, i.e., $\forall u \in N \ (\text{children}(u) \leq 1)$.

### A.1.5 Subtree Relation

The subtree relation is a binary relation between two trees $T' = (N', E', n')$, $T = (N, E, n)$, denoted by $T' \subseteq T$:

$$T' \subseteq T \iff N' \subseteq N \land E' \subseteq E$$

We say $T'$ is a sub tree of $T$, i.e., it is a substructure of another tree. Note that a subtree relation is merely subset relations on the first two elements of a tree. Not every node “below” the to-be subtree’s root is considered the subtree’s node.

A subtree $T' = (N', E', n')$ of $T = (N, E, n)$ rooted at one of $T$’s node $u$ is denoted by $T' \subseteq_u T$. That is, $T' \subseteq_u T$ if and only if $T' \subseteq T \land n' = u$.

Further, we define a $T[u]$ as the maximum subtree of $T = (N, E, n)$ “below” $u$ (inclusive). That is, let $T' = (N', E', n')$,

$$T' = T[u] \iff T' \subseteq_u T \land \{v \mid u \overset{E}{\rightarrow} v\} \subseteq N'$$

For the above two trees, it is easily seen that if $T' = T[u] \land u \neq n$, then $((N \setminus N') \cup \{u\}, E\setminus E', n)$ is still a tree. We denote such tree by $T \setminus T'$. When $u = n$, then $T \setminus T' = \emptyset$.

We can see an example of each kind of subtree relation in Figure A.1. Each lower-case character represents a node. Since we have not talked about what a node corresponds to, we do not need to look at the set of numbers next to each node for now. The $\sqsubseteq$ and $\propto^+$ relations are introduced in Section A.3.2.
A.2 Semantic Concepts

Now we have synchronised our understanding on the tree concepts, we can look at the subjects of discussion, program states and program execution. First we discuss what they are and then we can talk about how to describe them.

A.2.1 Concrete Program Execution and States

We take the following existing [71] definitions of concrete program execution and states: program execution is a transformation of program states; a state is a function that assigns variables to values.

At a certain instant during concrete program execution, the program is said to be in a certain state $S$. This state assigns variables to values and one of the variables is a program counter, i.e., a state also specifies its location in a program.

All program execution starts from one of the initial states. An initial state contains mappings from the program inputs to certain values. The set of initial states is denoted by $K_0$, which is assumed according to the problem we are trying to solve, e.g., we usually assume a precondition on the inputs to our program.

Given a program state, the next state is unique and decidable\(^2\). We use $\Gamma : \kappa \rightarrow \kappa$ to represent the transformation function of states. Here, $\kappa$ is the domain of states. $\Gamma$ is given by a specification of program execution. In this thesis, we discuss concrete execution and symbolic execution: concrete execution has a set of rules that define how to interpret an instruction given to a machine and consequently change the machine’s state; symbolic execution is discussed later.

We denote the transformation from state $S$ to $S'$ by $\Gamma(S) = S'$. Let $\Gamma^*$ be the reflexive, transitive closure of $\Gamma$. This notation can be extended to express a chain of transformation, e.g., from state $S_1$ via $S_2$, ..., to $S_k$: $\Gamma^*(S_1) = S_k$. Note that we do not discuss transformation over infinite “steps”. A chain of transformation is called an execution, denoted by $r = S_1, ..., S_k$.

Given $\Gamma$, every state must transform to another state otherwise the program should terminate at the state. We call this a terminal state. Suppose $S$ is such a state, we say $\Gamma(S) = \bot$. On the other hand, we say $\Gamma(\top) = S$ for all $S \in K_0$. The set of terminal states is denoted by $K_t$.

We can also define $K$ with respect to $\Gamma$, as the set of states that can be transformed to from a state in $K_0$ over a certain number of “steps”. We call the $\Gamma$, $K$ given by concrete execution the function of feasible transformation and the set of feasible states respectively. An execution $r = S_1, ..., S_k$ is feasible if and only if $\forall 1 < i \leq k$ ($S_i = \Gamma(S_{i-1})$) where $\Gamma$ is the function of feasible transformation. Without specification, we assume concrete execution for $\Gamma$ and $K$.

Take the program in Figure A.2 as an example. Here, $pc$ is the program counter, indicating what is going to be executed next. In this example it is impossible for the program to have $x$ not equal to 1 after executing the first line. For example, suppose the set of feasible states is $K$. A state $S = (x \mapsto 2, pc \mapsto 2)$ will not belong to $K$. Similarly, we have $S(pc) = 3 \Rightarrow S \notin K$ because line 3 is not possible to reach. Here, state $S$ is treated as a function. State $(pc \mapsto 6, x \mapsto 0)$ is a terminal state. Note that $pc \mapsto 6$ can both mean a state where the program is about to execute line 6 and a state where the program has just finished.

\(^2\)This is except when the state takes more inputs during the execution. For simplicity, we do not discuss this. We can also consider that a state contains input assignments for all inputs prior to execution.
executing line 5. Even though line 6 does not actually exist, it does not prevent us from using it to denote a terminal state. This is similar for the other examples.

### A.2.2 Classifying the States

Symbolic execution treats statements in a program differently. A difference is whether it makes solver calls at a certain kind of statement. For some statements, symbolic execution collects constraints and moves on very quickly; for the other statements, symbolic execution issue queries to the solver, which is a time-consuming process. We consider the latter sensitive statements and the former conversely insensitive. At a sensitive statement, symbolic execution checks the possibilities of different execution directions.

Now we define the classification of program statements during SE, as a method to distinguish sensitive and insensitive statements. A state located at (indicated by pc) a sensitive statement is called a sensitive state, otherwise an insensitive state.

#### Conditionals and Non-Conditionals

Conditionals and non-conditionals in a program provide a classification. All symbolic execution tools we know make distinction between them, i.e., make solver calls on conditionals.

We denote a state $S$ at a conditional by saying $S \in K_c$ and at an non-conditional by saying $S \in K_n$. $K_c, K_n$ are the set of conditional states and non-conditional states with respect to the set of feasible states $K$. 

---

```plaintext
1 x := 1
2 if (x < 0)
3   x := 0
4 else
5   x := x - 1
```

**Figure A.2:** Example program to explain the sets of feasible transformations and states.

**Figure A.3:** A state transition graph of concrete program execution of the program in Figure A.2 (suppose the program takes a 32-bit integer). Each circle is a state in $K$. Each arrow is a feasible transformation given by $\Gamma$. It also shows the different kinds of feasible states.
Note that a state $S$ is conditional does not mean that it can have two different transformation outcomes. A state still only has one next state $S' = \Gamma(S)$ for a certain transformation function $\Gamma$, e.g., in Figure A.3, there is only one outcome from the conditional state.

A.3 Execution Tree

In this section, we define execution tree.

A.3.1 Relation of Nodes and States

A node in an execution tree represents a set of states. From now on, we consider a node as if it is a set of states.

Notations on Node/Set of States

We clarify the notations used on sets of states. Since a node represents a set of states, the notations naturally apply to it.

Let $u$ be a set of states, we define

\begin{align*}
\text{conds}(u), \text{nconds}(u) & \quad \text{The sets of conditional and non-conditional states in } u \\
\text{sens}(u), \text{insens}(u) & \quad \text{The sets of sensitive and insensitive states in } u
\end{align*}

We abuse the notation of $\Gamma$ to apply to a node. Let $u$ be a node. $\Gamma[u]$ returns the set of states they transform to in one “step”, i.e., $\Gamma[u] = \{\Gamma(S) \mid S \in u\}$.

The set of all possible values of program counters in the states of a node is called the line numbers covered by the node. Previously we used $S(pc)$ to get the location of state $S$. Now we use $pc(S)$ as an equivalence. We can do the same for a set of states to get a set of covered line numbers: $pc[u] = \{pc(S) \mid S \in u\}$. We also abuse these notations for program elements that typically have program counters.

A.3.2 Tree Relations with Semantic Implications

In addition to structural relations, in this section, we introduce other tree relations that relate the sets of states in the corresponding nodes in these trees. These are shown in Figure A.1.

Node-Wise Subset Relation

The node-wise subset relation is saying two trees $T', T$ has the same structure only that each pair of corresponding nodes satisfy a subset relation. This is denoted by $T' \sqsubseteq T$. Formally, $T' \sqsubseteq T$ if and only if $\text{root}(T') \subseteq \text{root}(T)$ holds, and there is a bijection $f$ between $\text{children}(\text{root}(T))$ and $\text{children}(\text{root}(T'))$ such that

\[ \forall n \in \text{children}(\text{root}(T)) \exists T_a, T_b \left( T_a \sqsubseteq_n T \land T_b \sqsubseteq_{f(n)} T' \land T_b \sqsubset T_a \right) \]

Tree Approximation

Now we introduce the approximation relation between two trees. When two trees $T_a$ and $T_b$ approximate each other, we say $T_a \preceq T_b$ or $T_a \preceq^\uparrow T_b$. We have $T_a \preceq^\uparrow T_b \Leftrightarrow T_b \preceq^\downarrow T_a$. Since
one relation ($\propto^\uparrow$) is the inverse of the other ($\propto^\downarrow$), we only define $\propto^\downarrow$ below and $\propto^\uparrow$ will be automatically defined thereafter.

Let $T_a = (N_a, E_a, n_a), T_b = (N_b, E_b, n_b),

\[ T_a \propto^\downarrow T_b \iff \exists T (T_a \subseteq T \land T \subseteq n_b) \]

A.3.3 Concrete Execution Tree

In this section, we are going to use all the tools we have introduced above to define concrete execution tree, i.e., a tree given by concrete execution (which defines a $\Gamma$). It is the baseline of symbolic execution trees.

Definition 4 (Concrete execution tree) Given a program $P$, a set of initial states $K_0$, and the $\Gamma, K$ associated with concrete execution, a concrete execution tree $T = (N, E, n)$ is the tree with the smallest $N, E$ and each node with the smallest set of states, such that $K_0 \subseteq n$ and

\[
\forall u \in N
\Gamma[\text{conds}(u)] \subseteq u \land \\
\forall l \in \text{pc}[w] \exists v \in \text{children}(u) (w_l \subseteq v)
\]

Here, $w = \Gamma[\text{conds}(u)]$ is the set of states the conditionals in $u$ transform to, and $w_l = \{ S \mid S \in w \land S(\text{pc}) = l \}$ is the set of states in $w$ with line number $l$.

This definition has two parts: it requires every node is a $\Gamma^*$ closure on non-conditional states; it requires that the destination states of a nodes’ conditional states are partitioned and put into different children according to their pcs.

A.3.4 Symbolic Execution Tree

The concrete execution tree is a special case of symbolic execution tree. A difference is that the above concrete execution tree is only branching at conditionals, while symbolic execution tree can branch at more kinds of statements, e.g., memory operations, in order to analyse individual cases such as the cases of pointer being in bounds and out of bounds.

More generally, we can partition the set of sensitive states (in a set of states denoted by $K$) using the rules $C = \{ c_1, \ldots, c_k \}$ (assuming a $\Gamma$). $c_1, \ldots, c_k$ are disjoint rules that a sensitive state should satisfy one and only one of them, so that they are put into different Equivalence Classes (ECs), denoted by $\text{ec}_i(K)$. Here, $\text{ec}_i(K)$ means the EC identified by $i$. For convenience, we use $C(S)$ as the identifier of the EC $S$ belongs to.

The transformation function $\Gamma$ in symbolic execution can also be different. We discuss this later.

Definition 5 (Symbolic execution tree) Given a program $P$, a set of initial states $K_0$, and the $\Gamma, K$ associated with symbolic execution, a symbolic execution tree $T = (N, E, n)$ is the tree with the smallest $N, E$ and each node with the smallest set of states, such that $K_0 \subseteq n$ and

\[
\forall u \in N
\Gamma[\text{insens}(u)] \subseteq u \land \\
\forall \text{ec}_i(u) \exists v \in \text{children}(u) (\Gamma[\text{ec}_i(u)] \subseteq v)
\]
Faithful symbolic execution tree  If using the same classification and the same partitioning, a symbolic execution tree is the same as the concrete execution tree. We call such a tree Faithful Symbolic Execution Tree (FSET), meaning that SE is doing an analysis on a program the most faithfully to concrete execution.

Conditioned tree  A conditioned tree is a concrete or symbolic execution tree with a condition \( c \) on its initial set of states, i.e., using \( \{ S \mid S \in K_0 \land c(S) \} \) as the set of initial states. Here, \( c(S) \) is the evaluation of the condition \( c \) based on the variable assignments in \( S \). We denote this by \( T(c) \), in respect of a \( T = T(True) \). They satisfy \( T(c) \propto T \).

A.4 Symbolic Executors and More Complex Symbolic Execution Trees

Symbolic execution is a generalisation of concrete execution in many ways. We use an executor \( \epsilon \) to characterise an SE approach. In our discussion, the executor \( \epsilon \) is a triple of a transformation function \( \Gamma \), a classification of states, and a partitioning of sensitive states.

In symbolic execution, due to the nature of the applications and efficiency concerns, we usually alter the way to interpret a program, but we still want to make sure our “execution” is a sufficiently precise approximation of the real execution. In this section, we show that the different ways of interpretation between symbolic execution approaches have impact on the size of their symbolic execution tree, and the latter can be used in the discussion of the difficulty of the goals they want to achieve.

A.4.1 The Size of Execution Trees

In Section 2.2.1 we discuss why the number of constraint solver calls made in SE is as important as the difficulty of individual solver calls. Here we argue that this number can be partially reflected by the size of the symbolic execution tree.

Symbolic execution tools make solver calls at conditionals, more generally, sensitive statements, which might include conditionals, memory operations, divisions, dynamic invocations. These statements are where the SE tools might want to further investigate by separating one symbolic state into multiple ones. The number of nodes in an execution tree, as we define, is exactly how many times symbolic execution sees sensitives in a program. This is the reason why we think we can approximate the number of solver calls and thus the difficulty of SE by using a symbolic execution tree.

The size of a tree \( T \) is denoted by \( |T| \).

Limitations

Individual constraint solving time is decided by many factors and varies drastically. To name a few reasons, it is decided by the number of variables, how entangled are those variables, and what theories are involved. Moreover, in concolic execution, part of the constraint solving activities at conditionals or sensitives are replaced by concrete evaluation, which is comparably cheaper than constraint solving. Therefore, to approximate the time taken at individual decision making points is hard. Our approach consider a constraint solving call as the smallest unit of “cost”. Therefore it cannot be very precise.
A.4. Symbolic Executors and More Complex Symbolic Execution Trees

However, when incremental solving or other optimisations such as independence checking are enabled, the individual query solving time is not that different [19]. (On the other hand, techniques such as state merging complicates the problem.) Besides, the number of variables does not grow exponentially and there usually is a limit on how correlated they can be, considering large programs are large mainly because they have multiple modules, which can be independent with each other.

A bigger limitation is that we do not consider the solver calls that fail, because our execution tree representation does not contain infeasible states (or nodes that have infeasible states). For example, if a conditional has only one feasible branch, the other branch will not be reflected in the tree. Essentially, the degree of a node matters in this approximation, and we currently only take into account the feasible branches. This can be corrected by adding nodes that represent infeasible choices that take solver calls.

### A.4.2 Features of Symbolic Executors

In this section we discuss four kinds of symbolic executors that produce a symbolic execution tree that is different from the concrete execution tree or FSET.

#### Splitting

Some symbolic executors specialise different cases for some particular kinds of statements. Depending on the application, symbolic execution needs to capture some behaviours (usually errors) during execution in addition. For example, it often needs to specialise for memory visits where there could be an within-bounds case and an outside-bounds case.

An example of splitting can be seen in Figure A.4 and Figure A.5. The program tries to load from an address that can possibly go out of bounds (point to memory space without previously allocated data). A symbolic executor splits the original node \( n \) into two nodes to handle the cases of the pointer being out of bounds and being within bounds. The symbolic execution tree changes accordingly—it is no longer an FSET.

Here we can see that a splitting symbolic executor can be sensitive to more than just conditionals. In this case, it is sensitive to a load statement. We use line number 5a and 5b with 5a being a termination line\(^3\) to make it look like it is a conditional. Essentially, SE create two states each time it sees such a statement. In practice, symbolic execution tools can do this for divisions (checking whether or not the divisor can be 0), dynamic calls (checking each individual case where a function pointer points to different functions), and many other kinds of statements were there other requirements.

---

\(^3\)Strictly speaking, the termination is under-approximation. This is discussed later.
Appendix A. Symbolic Execution Tree

Figure A.5: Example of splitting. Node $n$ is split to distinguish two cases: an outside-bounds case and an within-bounds case. The symbolic execution tree changes accordingly from an FSET to a more complicated one.

Let us consider the relation between FSETs and such symbolic execution trees. To ease the process of understanding, let us first consider the situation where only one node is split.

After the splitting of a node, the symbolic execution tree is always more or equally complicated in that it has more nodes than it used to have. We call the nodes after splitting a virtual node (e.g., the middle and the right boxes in Figure A.5). A virtual node is a small tree structure. It can be easily visualised as a single node from the outside. Once a node is split, the states in the nodes originally reached by it will split up and reconnect to the virtual node in other forms. Apart from that the original node becomes multiple nodes (the virtual node), there are also more or equal number of nodes "below" them. We put this formally in the following theorem:

**Theorem 2 (Symbolic execution tree with single-node splitting)** Let $T = (N, E, n_0)$ be the original FSET when doing faithful symbolic execution on a program $P$. Let $T' = (N', E', n'_0)$ be the symbolic execution tree which results from splitting a single $T$-node $n$, which becomes a virtual node $v$ in $T'$. We have

$T \setminus T[n] = T' \setminus T'[v] \quad \land \quad $ **Equality above/at splitting**

$\forall (v, w) \in E'$

$\exists! (n, u) \in E (T'[w] \preceq T[u])$ **Under-approximation below splitting**

On the other hand, children($v$) is a partition of $\bigcup$ children($n$). In other words, no additional concrete states has been added to the tree.

Splitting will not change the set of states in a tree. This means that splitting in theory increases the difficulty of symbolic execution but does not introduce anything that cannot happen or miss anything that can happen in reality (the analysis will be as sound and complete as the analysis based on the original tree).

The relation between trees before and after the above single-step splitting is denoted by $T' \overset{sp}{=} T$ (with $n$ being the split node in $T$ and $T'$ being the split tree). Splitting can apply to other symbolic execution trees beside FSETs. More generally, splitting is not limited to
A.4. Symbolic Executors and More Complex Symbolic Execution Trees

Figure A.6: Example program for merging.

```
1  x := read
2  if (x < 0)
3    { x := -x }
4  else
5    { x := 1 }
6  y := read
7  if (y = 5)
8    x := x + 1
9  else
10   x := x + y
```

making non-conditionals sensitive but also includes making insensitives sensitive or altering
the definition of sensitives to further partition ECs.

Splitting is usually done regarding a kind or many kinds of statements, so more than one
node is split. The relation before and after splitting on multiple nodes of any execution tree
is denoted by

\[ T' = T \]

**Theorem 3 (Symbolic execution tree with multiple-node splitting)** Let \( T = (N, E, n) \) and
\( T' = (N', E', n') \) be two trees. Relation \( T' = T \) holds if and only if there exists a tree \( T_a = (N_a, E_a, n_a) \)
sp
satisfying \( T_a \equiv T \), and there exists a bijection \( f \) between children\((n_a)\) and children\((n')\),
such that

\[ \forall u \in \text{children}(n_a) \ T'[f(u)] = T_a[u] \]

**Merging**

**Merging** is the opposite of splitting. Merging is to reclassify some sensitive states as insensitiv-
es in the symbolic execution tree. We can see an example of merging in Figure A.6 and
Figure A.7. In this example, the executor is no longer sensitive to the conditional at line 2,
and we compute a different symbolic execution tree. We notice the node with the reclassi-
ified states (states at the \( \text{if} \) conditional at line 2) and its children are merge into a single node.
We first discuss the properties of such single-step merging where the reclassified states only
appears in a single node. Similarly, this node and its children are collectively referred to as a
virtual node.

Again, we care about the relation between the original tree and the merged tree. We can see that the virtual node and the original node always have the same set of states. Let
\( T = (N, E, n_0) \) and \( T' = (N', E', n'_0) \) be the trees before and after a single-step merging,
\( T \mathrel{\overset{\text{sp}}{\equiv}} T' \) holds between them if we consider the leftmost box as the virtual node \( (v) \) and the
rightmost box as the “split” node \( n \). We denote this by \( T' \overset{\text{sp}}{\equiv} T \). We have \( T' \overset{\text{sp}}{\equiv} T \iff T \overset{\text{sp}}{\equiv} T' \).

In a similarly way, a multiple-step merging, e.g., summarising a function which means
treating every originally sensitive states at or in a call as insensitives, develops a relation
between \( T, T' \), denoted by

\[ T' = T \overset{\text{mg}}{\equiv} T \overset{\text{sp}}{\equiv} T' \]

Merging is used in symbolic execution when we do not want to distinguish certain cases,
in contrast with splitting. **State merging** has been discussed in Section 2.3.2, which is a typical
application of this.
We have discussed two symbolic executor features, namely splitting and merging, one is the opposite of the other. We have also seen the changes splitting or merging can make to an execution tree. They have common properties in that they do not change the set of feasible states. In other words, any kinds of analysis performed on such execution tree can remain sound and complete. In the next sections, we are going to see two other kinds of symbolic executors that break either the completeness or the soundness.

**Under-Approximation**

Some symbolic executors can under-approximate the set of feasible states. When a smaller set of states can be used as an representative for a bigger set of states, symbolic execution can choose to neglect the rest of the states for the purpose of efficiency. This has a wide range of applications. For example, in some situations, symbolic execution tools use concretisation on variables which fixes them onto certain values so that SE is more efficient and robust (Section 2.3.3).

Let us see an example of under-approximation in Figure A.8. Here, the function call at line 2 ($\text{hash}(x)$) computes a hash for $x$. A symbolic execution may find this too expensive to investigate and choose to concretise the input $x$. Suppose it fixes $x$'s value to 1 and suppose $\text{hash}(1) = 6$. We can illustrate the relation between the FSET and the execution tree after under-approximation using Figure A.9.
A.4. Symbolic Executors and More Complex Symbolic Execution Trees

In Figure A.9, under-approximation is achieved by removing transformations from \( \Gamma \), i.e., making \( \Gamma(S) = \bot \) for the states that satisfy \( S(x) = 2 \land S(x) \neq 1 \). Some subsequent states are gone from the tree as a result, e.g., states with \( pc \mapsto 4 \).

The theorem below shows the relation between two trees, one is an under-approximation of the other:

**Theorem 4 (Relation before and after under-approximation)** Let \( T = (N, E, n_0) \) be the symbolic execution tree a symbolic executor \( \epsilon \) searches in given a program \( P \), and let \( T' = (N', E', n'_0) \) be the tree it searches in with under-approximation. They satisfy

\[
T' \preceq T
\]

After under-approximation, the set of states is a subset of the original one. An under-approximation executor sacrifices completeness of its applications.

**Over-Approximation**

Over-approximation is the counterpart of under-approximation. Consider the program in Figure A.10 where we want to know if line 4 is dead code. When \( x \) is symbolic, the function can potentially create a large number of paths in itself and therefore is expensive SE. However, we know \( \exp(x) \) is always positive. A good strategy is to not execute the function and over-approximate \( y \) as \( y > 0 \). This requires two operations on the symbolic execution tree: merging and over-approximating. Over-approximation means to assume a larger set of states, in this case, it is to assume a larger range of values for a certain variable.

In Figure A.11, we show the original symbolic execution tree and the tree after merging and over-approximation. The merging process does the same thing we already know from
Appendix A. Symbolic Execution Tree

```plaintext
1: x := read
2: y := exp(x)
3: if (y < 0)
4:   x := x + 1
5: else
6:   x := x + y
```

**Figure A.10:** Example program for over-approximation.

![Symbolic Execution Tree Diagram](image)

**Figure A.11:** Example of over-approximation.

Section A.4.2: the nodes in, before and after the called function all merge into a another node; the subtrees “below” them are also merged. The over-approximation process changes the set of states in this node: instead of specifying each state with \( pc \mapsto 3 \) has \( y \) mapped to a result of the function, it specifies that \( y \) can have any value greater than 0. This includes every value that is a result of the function and every positive value that is not. This is achieved by adding transformations to infeasible states into \( \Gamma \), such as \( \Gamma(\top) = (pc \mapsto 3, y \mapsto 1) \). The executor also allows them to transform to other states. As a result, additional states are added to the node \((pc \mapsto 6, 7)\).

What is slightly more interesting is when we further over-approximates \( y \) to be any possible value. We can see in the rightmost tree in Figure A.11 that it has an additional node representing the case when \( y \) is less than 0, which we know is not feasible. This shows the weakness of over-approximation being that it might introduce additional decision making points, which increase the cost of SE.

Also, it seems that the symbolic execution tree after merging is the most simplified one in the four trees in Figure A.11 in terms of number of nodes, and also in terms of the number of states in nodes compared to the over-approximated trees. However, we argue that using over-approximation is potentially better considering that it does not have to explore the \( \exp \) function, as is needed by merging. We have seen this idea as skipping in Chapter 4.

Similar to under-approximation, we have the following theorem:

**Theorem 5 (Relation before and after over-approximation)** Let \( T = (N, E, n_0) \) be the symbolic execution tree a symbolic executor \( \epsilon \) searches in given a program \( P \), and let \( T' = (N', E', n_0') \) be the tree it searches in with over-approximation. They satisfy

\[ T' \preceq T \]
In over-approximation, the set of states is a supersets of the original one. An under-approximation executor sacrifices soundness of its applications.

A.5 Comparing Symbolic Execution Tasks

Modern symbolic execution tools use the four different techniques we have summarised in combination. We can consider an executor applying these techniques as applying a combination of operators on the execution trees. We have the following four operators:

\[ T' = sp(T) \Rightarrow T' = T \]
\[ T' = mg(T) \Rightarrow T' = T \]
\[ T' = uapprox(T) \Rightarrow T' \propto T \]
\[ T' = oapprox(T) \Rightarrow T' \propto T \]

We note that the four operators are different from the four relations. An operator on \( T \) returns a certain result \( T' \) and implies a certain relation, while for a certain \( T \), there are many possible \( T' \) that has such a relation with it.

We also note that these operators are not commutative:

\[ \forall f, g \text{ are any operators above. Changing the order of them produces possibly different results even though the individual operations are the same.} \]

We can use the four operators to describe different symbolic execution approaches and use the implied relations to see the differences in between the SE tasks.

For example, KLEE uses splitting to discover memory violations; it concretises variables when there are external calls; it also concretises sizes of symbolic inputs (under-approximation as well). By design, KLEE does not use summarisation or over-approximation techniques. Suppose \( T \) is the FSET of the program KLEE runs on. KLEE will produce a corresponding symbolic execution tree

\[ T' = uapprox(sp(T)) \]

for this program. The difficulty of KLEE’s task (exhaustively exploration of the program for errors) can be approximated by the tree \( T' \) satisfying

\[ \exists T'' (T' \propto T'' \land T'' = T) \]

We can say things related to KLEE’s properties when reading into the relations, e.g., in principle, KLEE does not produce false alarms.

Next, we discuss some particular SE techniques using an execution tree presentation. This aids the discussion in our main chapters.

A.5.1 An Approximation of CSE’s Cost

In this section we identify the characteristics of CSE again, but using the concept of execution tree.
CSE, in the discussion of this thesis, is an application of contextual over-approximation in symbolic execution. It essentially has two phases:

1. Local symbolic execution on the code to be summarised—functions in the existing work or code fragments to be more general.
2. Global symbolic execution on the whole program, making use of the summaries generated from the previous process.

The former process over-approximates the context of the summarised code, and the latter process will have to deal with the summaries generated because of the lack of context information.

We can roughly quantify the size of the problem of CSE. A symbolic execution tree can be used to approximate the difficulty of SE. A merged symbolic execution tree introduced in Section A.4.2 is the most suitable in describing the global symbolic execution process. Each merging process applies to the nodes that corresponds to the code locations inside a piece of summarised code. Note that a node is a point where symbolic execution needs to make constraint solver calls in order to make a decision. The number of (satisfiability checking) calls is at least the number of children of this node. Merging makes multiple decision making points inside the summarised code become a single decision making point.

**Theorem 6 (Approximation of the size of the problem in CSE)** Let \( f_i \) denotes a function or a code fragment with identifier \( i \). Let Targets be a set of identifiers of summarisation targets, and let us assume \( f_i \) is not called by \( f_j \) for all \( i, j \in \text{Targets} \). Suppose \( f_i \)-local symbolic execution implies a symbolic execution tree \( T_i \). The exploration of \( f_i \) under a calling context \( c \) (as a precondition on \( T_i \)) is then implying a tree \( T_i(c) \) and we have \( T_i \propto \uparrow T_i(c) \). Suppose global symbolic execution without summarising any function, i.e., traditional symbolic execution implies a tree \( T \). The size of the problem \( m \) can be approximated as

\[
m = \sum_{i \in \text{Targets}} |T_i| + |T'| \]

where

\[
T' = T_{mg}
\]

We can see the CSE is potentially more efficient because the size of \( T' \) can be smaller than \( T \) (see merging in Section A.4.2). Note that nodes being merged are different considering different summarisation approaches. As we discussed in Chapter 3, our approach only merges nodes along a single path in the summarised code, while some approaches merge nodes in the whole summarised code. Also note that compared to the whole symbolic execution tree \( T, T_i \) is likely trivial if it is finite because it represents the exploration of a piece of code which is smaller in depth.

This abstraction of SE does not consider the difference in the cost of individual constraint solving activities. In the case of path merging, the decision making in the merged node is more expensive than the others, because constraint queries contain disjunctions. In the case where over-approximation is involved, i.e., bottom-up summarisation, there are also infeasible summaries, which increase the number of solver calls, and are not counted. Nevertheless, we can see a problem in CSE that we need to select Targets carefully, so that the lack of calling context \( (T_i \propto \uparrow T_i(c)) \) does not introduce too many infeasible summaries.
A.5. Comparing Symbolic Execution Tasks

A.5.2 An Approximation of Incremental SE’s Cost

We approximate the cost of a typical incremental SE process such as DiSE \[102\]. Suppose we use incremental SE to prove a program function by function. For each function \(f_i\), we have

\[
m_i = |T \setminus T[u_{i1}] \setminus T[u_{i2}] \setminus \ldots | + \sum_j |T_i(PC(path(T, u_{ij})))|
\]

Here, the tree set minus operation is left-associative; tree \(T\) is the whole program’s execution tree; \(T_i\) is the function \(f_i\)’s execution tree without any precondition; \(u_{ij}\) is the \(j\)th node in \(T\) that corresponds to an entry of function \(f_i\); \(PC(R)\) is the postcondition of path \(R\)\(^4\) and equivalently a precondition to the symbolic execution in the function. By this formula, we are saying the cost is the cost of directed symbolic execution to reach function \(f_i\) (\(|T \setminus T[u_{i1}] \setminus T[u_{i2}] \setminus \ldots |\) as the worst case), plus the cost of the exploration in the function (\(|T_i(PC(path(T, u_{ij})))|\)) via all call sites (the summation over \(j\)). We can see that incremental SE is more efficient in verifying particular functions than traditional SE in principle, because it is directed to them and ignores the other parts. Towards the verification of all functions (not including the entry function), the total incremental SE cost approximates

\[
m = \sum_i m_i
\]

Here, \(\sum_i \sum_j |T_i(PC(path(T, u_{ij})))|\) is simply the cost of traditional symbolic execution on those functions; \(\sum_i |T \setminus T[u_{i1}] \setminus T[u_{i2}] \setminus \ldots |\) is the cost of (repeated) search activities from the entry to the target functions.

\(^4\)Recall that a postcondition introduced in Chapter 3 is said with respect to a set of executions. It works here too considering \(R\) corresponds to a set of executions given by a path.
Appendix B

Example Usage of Stratus

This appendix uses the programs in Category (A), Section 4.5.3 to help understand part of the the evaluation process of Stratus concretely. We think it might also be a good opportunity to briefly introduce the implementation and options available in Stratus.

Implementation details Stratus is built on top of KLEE. We briefly summarise our implementation as three major modifications:

1. Modification of the execution’s direction at calls and returns. This includes identifying skippable calls, skipping (directing the execution over the calls), and resolving (directing the execution to skipped calls when a state reaches the end of a depth or reaches a return in a call being resolved.)

2. Managing constraint consistency. This includes recording the calling context, such as the variables’ states, the call stack’s states, and the address space’s states; it also includes restoring these information at the beginning of resolving and checking a state’s feasibility at the end of resolving.

3. Keeping aware of the context. This means to let a state know its current invocation depth and the instructions it covers or may-covers. A static analyser is needed to obtain such information; a searcher is needed to make use of it.

Explanation of the the options Each program and the corresponding configuration can be seen in each figure. The option explanation can be seen in Table B.1. For completeness we also include the options not used in the experiments but are available in Stratus. The other options not seen in the table are originals of KLEE. Note that the --finish-line option in the configuration is added to both of KLEE and Stratus in the experiments of proving unreachability. We made this simple modification to KLEE to ensure it receives the same information as Stratus about not to search past a line where we can assume there is no error beyond. This way symbolic execution does not get stuck in the while loops which we understand cannot cause error. We make KLEE take advantage of this option as well as Stratus.

Notes on multipass Program multipass (Figure B.4a) is used to illustrate how to apply symbolic execution in server-side verification of client behaviours [32]. The goal of symbolic execution is to tell whether the client has been modified by executing the correct client to see if it is possible to generate a message the server has now received. In this example, the send line indicates a place where a correct client sends message, and where a verification tool will
### Table B.1: Stratus options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--skip-function</code></td>
<td>Enabling skipping and resolving. Skippable (see Algorithm 4.2) functions will be automatically skipped. For example, in Figure B.1, function <code>foo</code> will be automatically recognised as skippable.</td>
</tr>
<tr>
<td><code>--targeted-resolve</code></td>
<td>Enabling targeted exploration. (See Section 4.4.2.)</td>
</tr>
<tr>
<td><code>--target-errors</code></td>
<td>In the targeted exploration mode, making any error in the program target.</td>
</tr>
<tr>
<td><code>--target-line=&lt;source&gt;:&lt;lineno&gt;</code></td>
<td>In the targeted exploration mode, specifying a line in a source code file as target. Note that it requires the program is compiled with debugging information and the line is not optimised out.</td>
</tr>
<tr>
<td><code>--finish-line=&lt;source&gt;:&lt;lineno&gt;</code></td>
<td>Specifying a line in a source code file as program exit. It can force terminate a state when it reaches such a line. Setting a line that is, e.g., after a target, as a finish line can quickly eliminate the states that can never reach the target. It has similar requirements to the above option.</td>
</tr>
<tr>
<td><code>--search=nurs:maycov</code></td>
<td>This is an option available in KLEE. The default searcher is <code>random-path interleaved with nurs: covnew</code>. Here, we can switch <code>nurs: covnew</code> to <code>nurs: maycov</code> to use the stratified searcher so as to handle may-instructions. (See Section 4.3.)</td>
</tr>
<tr>
<td><code>--skip-whitelist=&lt;fname&gt;:&lt;fname&gt;:...</code></td>
<td>Specifying what functions could be skipped. (See Algorithm 4.2.)</td>
</tr>
<tr>
<td><code>--skip-blacklist=&lt;fname&gt;:&lt;fname&gt;:...</code></td>
<td>In the case of Figure B.3, <code>g</code> is blacklisted, because after this call there is nothing to execute, so skipping it is pointless.</td>
</tr>
<tr>
<td><code>--max-skips=&lt;int&gt;</code></td>
<td>Specifying the maximum number of times a state can skip functions (default 1; 0 means no limit). In the case of Figure B.3, the states need to skip two functions (<code>foo</code> and <code>f</code>). Therefore, we make it 0.</td>
</tr>
<tr>
<td><code>--max-resolves=&lt;int&gt;</code></td>
<td>Specifying the maximum number of times a skippable call can fail being resolved before it is treated unskippable (default no limit). (See Section 4.3.3.)</td>
</tr>
<tr>
<td><code>--shallow-ivc-depth=&lt;int&gt;</code></td>
<td>This is equivalent to <code>TargetIvcDepth</code> in Algorithm 4.4.</td>
</tr>
<tr>
<td><code>--max-ivc-depth=&lt;int&gt;</code></td>
<td>Specifying the maximum invocation depth (default no limit). (See Algorithm 4.4.)</td>
</tr>
</tbody>
</table>
# Appendix B. Example Usage of Stratus

## (A) Program directedse1.

```c
#include <stdio.h>
#include <assert.h>

void foo(int x) {
    int i;
    for (i = 0; i < 32; i++) {
        if (x & (1 << i)) {}
    }
}

int main(int argc, char **argv) {
    int i, n = 0;
    int b[4] = { 0 };
    int b[10] = { 0 };
    int irrlv;
    klee_make_symbolic(&irrlv, sizeof(irrlv), "irrlv");
    for (i = 1; i < argc; i++) {
        if (*argv[i] == 'b') {
            assert(n < 4);
            assert(n < 10);
            b[n++] = 1;
        } else {
            foo(irrlv);
        }
    }
    while (1) {
        if (getchar()) {}
    }
    return 0;
}
```

(8) The configuration for program directedse1. The bolded options are the ones additionally enabled in Stratus. The rest are used in both KLEE and Stratus.

## FIGURE B.1: The experiment on program directedse1. The colourful text is used by a specific variant of the program exclusively. The orange line (line 13 and 19) is enabled to get a variant with a reachable target; the blue line (line 14 and 20) is enabled to get a safe variant. Similarly, the `--finish-line` option is enabled when doing experiment on the safe variant.
Appendix B. Example Usage of Stratus

```c
#include <assert.h>

int f(int m, int n) {
    int i, a, sum = 0;
    for (i = 0; i < 6; i++) {
        a = n % 2;
        if (a) sum += a + 1;
        n /= 2;
    }
    return sum;
}

int main() {
    int m, n, i;
    int error = 0;
    int sum = 0;
    klee_make_symbolic(&m, sizeof(m), "m");
    klee_make_symbolic(&n, sizeof(n), "n");
    for (i = 0; i < 1000; i++)
        if (m == i) sum = f(m, n);
    while(1) {
        if (sum == 0 && m == 7)
            if (sum == 20 && m == 7)
                assert(0);
            error = 0;
    }
}
```

(A) Program directedse2.

(b) The configuration for program directedse2.

```
Klee --skip-function --targeted-resolve --target-errors \ --search=random-path --search=nurs:maycov \ --finish-line=<path>/directedse2.c:27 \ --libc=uclibc --exit-on-error --max-time=300 \ directedse2.o
```

(Figure B.2: The experiment on program directedse2. The text styles and colours have similar meanings to Figure B.1. In this case, line 24 is used by the reachable variant and line 25 by the safe one.)
#include <assert.h>

void foo(int x) {
    int i;
    for (i = 0; i < 32; i++) {
        if (x & (1 << i)) {

        }
    }
}

int f(int m, int n) {
    int i, a, sum = 0;
    for (i = 0; i < 6; i++) {
        a = n % 2;
        if (a) sum += a + 1;
        n /= 2;
    }
    return sum;
}

void g(int m, int n) {
    int i, sum = 0;
    int error;
    for (i = 0; i < 1000; i++) {
        if (m == i) sum = f(m, n);
    }
    while (1) {
        if (sum == 0 && m == 37)
            if (sum == 20 && m == 37)
                assert(0);
            error = 0;
    }
}

int main() {
    int m, n;
    int irrlv;
    klee_make_symbolic(&m, sizeof(m), "m");
    klee_make_symbolic(&n, sizeof(n), "n");
    klee_make_symbolic(&irrlv, sizeof(irrlv), "irrlv");
    foo(irrlv);
    if (m >= 30) g(m, n);
}

(A) Program directedse3.

```
  klee --skip-function --max-skips=0 --skip-blacklist=g \\
  --targeted-resolve --target-errors \\
  --search=random-path --search=nurs:maycov \\
  --finish-line=<path>/directedse3.c:31 \\
  --libc=uclibc --exit-on-error --max-time=300 \\
  directedse3.o
```

(B) The configuration for program directedse3.

Figure B.3: The experiment on program directedse3. The text styles and
colours have similar meanings to Figure B.1. In this case, line 28 is used by
the reachable variant and line 29 by the safe one.
Appendix B. Example Usage of Stratus

```c
#include <assert.h>

int AES(int iv) { /* Some computation */ }

void Client(int x, int y, int iv) {
    int p = x * y;
    if (x % 9 == 0)
        if (x % 9876 == 0)
            if (y & 1 == 1) {
                int s = AES(iv);
                int c = p ^ s;
                if (iv == 0x1234 && c == 0x9dac) // SEND(iv, c)
                    assert(0);
            }
    }
}

int main() {
    int x, y, iv;
    klee_make_symbolic(&x, sizeof(x), "x");
    klee_make_symbolic(&y, sizeof(y), "y");
    klee_make_symbolic(&iv, sizeof(iv), "iv");
    Client(x, y, iv);
}
```

(A) Program `multipass`.

```
klee --skip-function --skip-whitelist=AES 
--targeted-resolve --target-errors 
--search=random-path --search=names:maycov 
--libc=uclibc --exit-on-error --max-time=300 
multipass.o
```

(B) The configuration for program `multipass`.

Figure B.4: The experiment on program `multipass`. The text styles and colours have similar meanings to Figure B.1. In this case, line 7 is used by the reachable variant and line 8 by the safe one.
tell whether the currently received message (0x1234, 0x9dac) is a possible outcome of that line. If it is, then the message passes the check, otherwise a malicious message has been found. In other words, reaching the assertion failure means passing the check, and failing to reach it means an anomaly. AES is a cryptographic function where symbolic execution states can multiply. The original function used is not specified so we use a hash function which is sufficient for showcasing the idea. In this case, AES can generate a lot of states, while without the important information in the SEND line, most of these states are not going to reach the target. Therefore, AES is a good function for skipping and we whitelist it specifically.
Bibliography


Author/s: Lin, Yude

Title: Symbolic execution with over-approximation

Date: 2017

Persistent Link: http://hdl.handle.net/11343/197985

File Description: Thesis

Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.