METHODS FOR BINARY SYMBOLIC EXECUTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Anthony Romano
December 2014
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dawson, Engler, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.
Abstract

Binary symbolic execution systems are built from complicated stacks of unreliable software components, process large program sets, and have few shallow decisions. Failure to accurately symbolically model execution produces infeasible paths which are difficult to debug and ultimately inhibits the development of new system features. This dissertation describes the design and implementation of KLEE-MC, a novel binary symbolic executor that emphasizes self-checking and bit-equivalence properties.

This thesis first presents cross-checking for detecting causes of infeasible paths. Cross-checking compares outputs from similar components for equivalence and reports mismatches at the point of divergence. This approach systematically finds errors throughout the executor stack from binary translation to expression optimization.

The second part of this thesis considers the symbolic execution of floating-point code. To support floating-point program instructions, KLEE-MC emulates floating-point operations with integer-only off-the-shelf soft floating-point libraries. Symbolically executing these libraries generates test cases where soft floating-point implementations and floating-point constraint solvers diverge from hardware results.

The third part of this thesis discusses a term rewriting system based on program path derived expression reduction rules. These reduction rules improve symbolic execution performance and are machine verifiable. Additionally, these rules generalize through further processing to optimize larger classes of expressions.

Finally, this thesis describes a flexible mechanism for symbolically dispatching memory accesses. KLEE-MC forwards target program memory accesses to symbolically executed libraries which retrieve and store memory data. These libraries simplify access policy implementation and ease the management of rich analysis metadata.
Acknowledgements

Foremost, I would like to thank my thesis advisor Dawson Engler for his often frustrating but invariably insightful guidance which ultimately made this work possible. His unwaning enthusiasm for developing neat system software is perhaps only matched by his compulsion to then break said software in new and interesting ways.

The other two-thirds of my reading committee, Alex Aiken and David Mazières, helpfully and recklessly agreed to trudge through over a hundred pages of words about binary symbolic execution.

Although the vast majority of this work was my own, several people did contribute some code in one way or another. KLEE-MC depends on a heavily modified derivative of the KLEE symbolic executor, originally developed by Daniel Dunbar and Cristi Cadar in Dawson’s lab shortly before my time at Stanford. T.J. Purtell helped develop an early version of the machine code to LLVM dynamic binary translator. James Knighton assisted in creating a public web interface for the system. David Ramos wrote a few KLEE patches that I pulled into KLEE-MC early on; he once casually remarked the only way to check these systems is mechanically.

This research was partially supported by DARPA award HR0011-12-2-009 and the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract N66001-10-2-4089. This work was also supported in part by the US Air Force through contract AFRL-FA8650-10-C-7024. Any opinions, findings, conclusions, or recommendations expressed herein are those of the author, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.
Contents

Abstract iv

Acknowledgements v

1 Introduction 1

1.1 Motivation ................................................. 1

1.2 Challenges in Symbolic Execution ............................. 3

1.2.1 Accuracy and Integrity .................................... 4

1.2.2 Floating-Point Code ....................................... 5

1.2.3 Expression Complexity ..................................... 5

1.2.4 Memory Access Analysis ................................... 6

1.3 Contributions ............................................... 7

2 The klee-mc Binary Program Symbolic Executor 10

2.1 Introduction .................................................. 10

2.2 Background .................................................. 11

2.2.1 Symbolic Execution ....................................... 11

2.2.2 Binary Symbolic Execution ................................. 14

2.3 Generating Tests with KLEE-MC ............................. 16

2.4 Design of KLEE-MC ......................................... 18

2.4.1 Program Snapshots ....................................... 18

2.4.2 Dynamic Binary Translation ............................... 21

2.4.3 Expressions and Solving .................................. 27

2.4.4 Scheduling .................................................. 32
2.4.5 Runtime Libraries ........................................ 33
2.4.6 Limitations ............................................... 39
2.5 Experiments .................................................. 41
  2.5.1 Snapshots ............................................... 41
  2.5.2 Program Tests and Bugs ............................... 43
  2.5.3 Coverage ............................................... 45
2.6 Conclusions ............................................... 47

3 Machine Cross-Checking ..................................... 49
  3.1 Introduction ................................................ 49
  3.2 Motivation .................................................. 50
    3.2.1 Bad Paths ............................................ 51
    3.2.2 System Complexity .................................... 52
  3.3 Cross-Checking in klee-mc ................................ 54
    3.3.1 Deterministic Executor Data .......................... 54
    3.3.2 Operating System Differencing on the System Model .. 59
    3.3.3 Execution Testing with Deterministic Test Replay .... 61
    3.3.4 Host CPU and the Machine Code Front-End .......... 63
    3.3.5 Constraint Validity for Expressions and Solvers .... 65
  3.4 Implementation ............................................ 68
    3.4.1 Static Non-determinism ............................... 68
    3.4.2 Dynamic Non-determinism ............................ 68
  3.5 Experiments and Evaluation ............................... 69
    3.5.1 Linux Program Test Cases ............................. 70
    3.5.2 Machine Code Front-End .............................. 73
    3.5.3 System Model ......................................... 76
    3.5.4 Expressions ........................................... 78
  3.6 Related Work .............................................. 81
  3.7 Conclusion ................................................. 82

4 Off-the-Shelf Symbolic Floating-Point ....................... 84
  4.1 Introduction .............................................. 84
5.7 Evaluation .................................................. 128
  5.7.1 Implementation ........................................ 129
  5.7.2 Test System ........................................... 129
  5.7.3 Performance .......................................... 130
  5.7.4 EquivDB ............................................... 132
  5.7.5 Rules .................................................. 134
5.8 Related Work ............................................ 136

6 Symbolically Executed Memory Management Unit 137
  6.1 Introduction ............................................. 137
  6.2 Background .............................................. 139
    6.2.1 Related Work ....................................... 139
    6.2.2 Symbolic Execution and Memory Accesses ............ 140
  6.3 A Symbolically Executed MMU .......................... 142
    6.3.1 Address Space Structures ............................ 142
    6.3.2 Soft Handlers ....................................... 142
  6.4 Access Policies and Analysis .......................... 146
    6.4.1 Symbolic Access Translation ........................ 146
    6.4.2 Shadowing Memory .................................. 150
    6.4.3 Unconstrained Pointers ............................. 154
  6.5 Evaluation ............................................... 158
    6.5.1 Implementation Complexity ........................ 158
    6.5.2 Dispatch Mechanism ................................ 158
    6.5.3 Policy Microbenchmarks ............................. 159
    6.5.4 Memory Faults in Linux Programs .................... 160
    6.5.5 Profiler ............................................ 161
    6.5.6 Heap Violations .................................... 162
    6.5.7 Unconstrained Pointers ............................. 164
  6.6 Conclusion .............................................. 167

7 Conclusion ................................................. 168
  7.1 Summary of Contributions ............................... 168
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Future Work</td>
<td>169</td>
</tr>
<tr>
<td>7.2.1 Programs and Modeling</td>
<td>170</td>
</tr>
<tr>
<td>7.2.2 Scaling</td>
<td>171</td>
</tr>
<tr>
<td>7.2.3 Coverage</td>
<td>172</td>
</tr>
<tr>
<td>7.2.4 Types of Analysis</td>
<td>173</td>
</tr>
<tr>
<td>7.3 Final Thoughts</td>
<td>174</td>
</tr>
</tbody>
</table>

A Glossary                        | 175  |

Bibliography                      | 179  |
## List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Number of experimental programs for symbolic execution systems.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Selected interpreter intrinsic extensions for runtime libraries.</td>
<td>35</td>
</tr>
<tr>
<td>2.3</td>
<td>Snapshot counts for binary program collections</td>
<td>41</td>
</tr>
<tr>
<td>2.4</td>
<td>Selected command lines triggering memory access faults</td>
<td>45</td>
</tr>
<tr>
<td>2.5</td>
<td>Selected file inputs triggering memory access faults</td>
<td>46</td>
</tr>
<tr>
<td>2.6</td>
<td>Amount of machine code covered by KLEE-MC</td>
<td>46</td>
</tr>
<tr>
<td>3.1</td>
<td>Mass checking and cross-checking results on Ubuntu Linux programs.</td>
<td>71</td>
</tr>
<tr>
<td>3.2</td>
<td>Valid x86-64 code causing VEX to panic, corrupt register state, or invoke divergent behavior</td>
<td>74</td>
</tr>
<tr>
<td>3.3</td>
<td>VEX decoder errors found with a static register file</td>
<td>75</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparison between model and operating system with LTP suite.</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Rewritten LLVM instructions with corresponding SoftFloat functions</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>Floating-point operation test cases from symbolic execution</td>
<td>96</td>
</tr>
<tr>
<td>4.3</td>
<td>JIT register log cross-checking failures on floating-point library self-tests</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>Selected library bugs from soft floating-point library consistency tests</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Hardware cross-check errors from all distinct floating-point tests</td>
<td>101</td>
</tr>
<tr>
<td>4.6</td>
<td>Conformance and library test cases applied to several FPA solvers</td>
<td>106</td>
</tr>
<tr>
<td>4.7</td>
<td>Bugs found in Linux programs following floating-point computation</td>
<td>107</td>
</tr>
<tr>
<td>5.1</td>
<td>Lines of code for expression optimization</td>
<td>129</td>
</tr>
<tr>
<td>6.1</td>
<td>Trap handler functions for a handler named $m$ with access bit-widths $w = {8, 16, 32, 64, 128}$</td>
<td>144</td>
</tr>
</tbody>
</table>
6.2 Runtime primitives for memory access .......................... 145
6.3 Lines of code for MMU components .............................. 158
6.4 Access faults found with symMMU ............................... 161
6.5 symMMU-derived Heap Violations ................................. 163
6.6 Mismatches against glibc involving unconstrained pointers 167
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A symbolically executed program and its decision tree.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Stages of translation annotated by propagation of symbolic state.</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>A solver stack with query serialization isolation.</td>
<td>30</td>
</tr>
<tr>
<td>2.4</td>
<td>A library intrinsic to enumerate predicated values</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>Function entry hooks for process exit functions</td>
<td>39</td>
</tr>
<tr>
<td>2.6</td>
<td>Snapshot storage overhead before and after deduplication</td>
<td>42</td>
</tr>
<tr>
<td>2.7</td>
<td>Test cases and paths found during mass testing</td>
<td>43</td>
</tr>
<tr>
<td>2.8</td>
<td>Bugs found during mass testing</td>
<td>44</td>
</tr>
<tr>
<td>2.9</td>
<td>Observed system call frequency over all programs</td>
<td>47</td>
</tr>
<tr>
<td>3.1</td>
<td>A user’s limited perspective of a bad path</td>
<td>51</td>
</tr>
<tr>
<td>3.2</td>
<td>A binary symbolic execution stack with multiple replay facilities.</td>
<td>53</td>
</tr>
<tr>
<td>3.3</td>
<td>Building a test log during symbolic execution</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Test case log record structures for system calls</td>
<td>58</td>
</tr>
<tr>
<td>3.5</td>
<td>The process for comparing model and operating system side effects.</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Symbolic decoder cross-checking data flow</td>
<td>63</td>
</tr>
<tr>
<td>3.7</td>
<td>Symbolic register file derived constraints to trigger a conditional jump.</td>
<td>65</td>
</tr>
<tr>
<td>3.8</td>
<td>Cross-checking two expression builders.</td>
<td>67</td>
</tr>
<tr>
<td>3.9</td>
<td>A broken floating-point division sequence</td>
<td>72</td>
</tr>
<tr>
<td>3.10</td>
<td>Cumulative time distribution to build register logs for each test case.</td>
<td>73</td>
</tr>
<tr>
<td>3.11</td>
<td>Number of basic blocks checked against register log as a function of time.</td>
<td>74</td>
</tr>
<tr>
<td>3.12</td>
<td>Time to verify machine-derived rewrite rules</td>
<td>78</td>
</tr>
<tr>
<td>3.13</td>
<td>Translation rule test with undefined result</td>
<td>79</td>
</tr>
<tr>
<td>3.14</td>
<td>Bad multiply caught by cross-checking.</td>
<td>80</td>
</tr>
</tbody>
</table>
3.15 A faulty hard-coded optimization ........................................ 80
4.1 IEEE-754 format for single and double precision floating-point data 88
4.2 Generating program tests and cross-checking soft floating-point .... 95
4.3 Floating-point test case distribution .................................... 100
4.4 Mistranslation of the \texttt{cvtsi2ss} instruction in the VexIR ........ 102
4.5 Soft floating-point library code coverage and total covered instructions 104
4.6 A Z3 test case query for checking a single-precision division result .. 106

5.1 Percent time for queries .................................................. 110
5.2 Total external solver calls ............................................... 110
5.3 \texttt{klee} Expression Language Grammar .............................. 111
5.4 Storing and checking an expression against the EquivDB .......... 115
5.5 An example translation verification query ............................. 117
5.6 Expression reduction rule grammar .................................... 119
5.7 Acceptance of constant widths on expression equivalence class hits .. 126
5.8 Percentage of queries submitted using rule sets over baseline ...... 131
5.9 Percent time for expression optimization over baseline ............ 131
5.10 Query time distribution for baseline and rule test cases .......... 133
5.11 Distribution of expressions by node count and bitwidth in EquivDB .... 134
5.12 Program rule sharing .................................................. 135
5.13 Rules for measured programs .......................................... 135

6.1 Symbolic memory access example code and its intermediate expressions. 138
6.2 A memory object shared among three states, an object state shared
    between two states, and an unshared object state. .................... 142
6.3 The symMMU pointer handling path .................................... 144
6.4 Range checking handler for 8-bit loads ............................... 147
6.5 Address concretization for 8-bit loads ................................ 148
6.6 Retrieving a disjunction of shadow values for a symbolic address .... 151
6.7 The memcheck shadow state machine ................................... 153
6.8 Unconstrained pointer structures ..................................... 155
6.9 String microbenchmark testing speed-up. .......................... 161
6.10 Profiler benchmark performance. ................................. 162
6.11 A tortuous heap read violation in cpio. ......................... 164
6.12 Simplified IP address parser from musl. ....................... 166
Chapter 1

Introduction

Complex software rarely functions as intended without going through considerable testing. Designing and implementing sufficient tests by hand is an arduous task; automated program analysis tools for finding software bugs promise to help shift this burden to machines. Unfortunately, such tools are no different from any other software; they are difficult to correctly construct, test, and debug. This work argues these challenges become tractable for at least one analysis technique, dynamic binary symbolic execution, by lacing self-testing capabilities and interchangeable components throughout the execution stack.

1.1 Motivation

Software defects, despite decades of experience and error, still continue to pose serious risk. Complex computer-controlled systems such as rockets [16] and medical devices [85] infamously demonstrate two instances of historically catastrophic software flaws. Failure to account for coding mistakes and undesirable edge cases means more than crashed computers and lost work, but also malicious destruction of nuclear centrifuges [83], loss of virtual assets [62] and global disruption of public key encryption infrastructure [112].

Standard software engineering uses a variety of processes and practices to improve code quality. These processes cover an extensive range of techniques, spanning
from unit tests for dynamically validating software component functionality to static verification that program properties unconditionally hold. Testing [56], in general, can describe errors in deployed software, detect functional regressions introduced by code updates, or assist when creating new features, such as with test-driven development [14]. Such test cases challenge the target code with a sequence of inputs then comparing the computed results against an expected result. Although testing is commonly automated, test cases are typically developed by a programmer, incurring additional development costs. Worse, testing rarely proves the absolute absence of errors. Likewise, verification systems cannot infer all intended behavior, hence requiring programmer-defined specifications or code annotations to guide analysis.

Ideally, there would be a software tool that could generate complete test case suites capable of covering all distinct behavior in any given program. A software developer would then simply review these test cases, checking for intended behavior, and repair all critical bugs. Unfortunately, Rice’s theorem [111] proves the undecidability of discovering non-trivial software defects, such as memory access faults, buffer overflows, and division by zero, for arbitrary programs. Instead, approaches to automating program analysis must compromise.

Traditionally these approaches are broadly categorized as either static or dynamic. A static analysis processes code structure, and therefore has excellent coverage, but must overapproximate program state to complete in finite time. A dynamic analysis executes code, and therefore struggles to achieve adequate coverage, but its knowledge of program state can be precise. In both cases, these algorithms must occasionally either overlook or misreport program errors.

Despite the intrinsic limits, systems built to analyze programs are effective enough to be useful in practice [17]. However, a fundamental syllogism still remains: software has bugs and program analysis systems are software, therefore these systems have bugs; a poor implementation promptly undermines any theoretical guarantees. First, if there is a bug in the system, the exact point of failure is rarely obvious. Second, poor or overspecialized optimization from tuning for small benchmarks leads to performance anomalies over diverse program sets. Third, an incomplete or partial
implementation that diverges from a precise machine representation due to technical limitations (e.g., pure source-based analysis) can ignore important code, casting aside potential sources of defects. Finally, new analysis passes must undergo a lengthy debugging phase when the system is poorly constructed or pathologically coupled.

In the past, users could be expected to file detailed tool bug reports. As program analysis systems become more sophisticated and deployments begin to analyze thousands of programs at a time, such a mindset is no longer adequate; too few people understand the system to make sense of the vast amount of data by hand. To realistically advance the state of the art, a program analysis tool should now be designed to identify and minimize its own defects.

1.2 Challenges in Symbolic Execution

Symbolic execution systems are especially well-suited to identifying their own errors. They are dynamic so it is possible to compare intermediate computations against a baseline execution. They generate their own test cases so they need minimal, if any, application-specific customization and can therefore process a large class of programs with little effort. They are sophisticated enough to be interesting.

This thesis primarily deals with correctness and performance for a binary symbolic execution. Although most of this work applies to symbolic execution in general, a binary symbolic executor has the convenient property in that its expected ground truth directly corresponds to a readily available machine specification: physical hardware. Coincidentally, machine code is a popular distribution format for programs; a great deal of software is already packaged for the executor. Analyzing very large program sets compounds the problem of dealing with the complicated inner-workings of a binary symbolic executor to such an extent that debugging and tuning such a system by hand quickly becomes impractical.

Addressing all open problems for symbolic execution is outside the scope of this dissertation; we focus on a small but tractable subset. Namely, we observe that a straightforward binary symbolic execution system on its own is naturally unreliable and rife with performance pitfalls. First, correctly interpreting code symbolically is
rather difficult; since so much state is kept symbolic, an errant computation potentially first manifests far from the point of error. Second, even running common types of code symbolically, such as floating-point instructions, remains a subject of sustained research. Third, symbolic overhead tends to create very complex expressions that ruin performance. Finally, building rich metadata describing state, necessary for many dynamic analysis algorithms, directly into the executor demands onerous changes to the core system.

1.2.1 Accuracy and Integrity

A symbolic executor runs program code over symbolic inputs to discover paths for test cases. This involves interpreting the program using symbolic data, symbolically modeling environment inputs, managing path constraints, and constructing tests by solving for symbolic variable assignments. For every test case, replacing all symbolic inputs with a concrete variable assignment should reproduce the followed path.

However, symbolic execution systems are imperfect. An executor may misinterpret the code. Its symbolic system model, whether emulating system calls or standard libraries, may diverge from the target environment. Expressions may be misoptimized and constraints may be corrupted. With this in mind, there is no guarantee a test case will accurately reflect the derived path when run through the program on a physical machine.

In light of executor flaws, there are a few options. If a test case fails to replay its path, the result may simply be thrown away as a false positive; the user never sees a bad result. If the problem is found to be in the executor itself, the tool author may be alerted to the problem. However, if the bug is only observed in one or two programs, it will be marked low priority; determining the cause of the tool bug can be prohibitive. Worse, if the false positive depends on non-determinism in the executor, it may be impossible to reliably reproduce the error.
1.2.2 Floating-Point Code

User-level programs commonly include floating-point code. If a symbolic executor supports a wide class of programs then it must handle floating-point instructions. When floating-point code is symbolically modeled, it should model floating-point data symbolically by precisely tracking floating-point path constraints. Likewise, the paths and values should match hardware.

Integer code is necessary for symbolic execution but floating-point is an enhancement. Compared to integers, floating-point operations have complicated semantics which are more difficult to model correctly. From a design standpoint, treating all data as integers simplifies the executor implementation by limiting expression types. Finally, although there are plenty of integer constraint solvers, there are comparatively fewer, let alone efficient, floating-point solvers.

Hence the challenge to supporting floating-point code involves striking a balance among performance, analysis precision, and system complexity. While concretizing floating-point data \[28\] avoids symbolic computation entirely, achieving good performance with few system modifications, the state becomes underapproximated, losing paths. Conversely, fully modeling floating-point data precisely \[6, 10, 20\] keeps all paths at the cost of additional system complexity and floating-point solver overheads. Strangely, considering floating-point’s infamous peculiarities \[61\], testing the correctness of floating-point symbolic execution itself is given little attention.

1.2.3 Expression Complexity

In order to maintain symbolic state, a symbolic executor translates operations from instructions into symbolic expressions. When there is significant live symbolic data, the executor generates a great deal of expressions. By keeping expressions small, the executor reduces overall constraint solver overhead, a main performance bottleneck. Ideally, expressions would be internally represented using the fewest nodes possible.

These expressions tend to grow quite large despite having logically equivalent, smaller representations. In essence, building expressions strictly according to the instruction stream ignores opportunities to fold operations into smaller expressions.
Binary symbolic execution magnifies this problem; the most efficient machine code can induce expression dilation under symbolic execution. Therefore it is important for a symbolic executor to include an expression optimization component.

Expression optimizations are often coded into the executor as needed. Typically, the system author observes a program symbolically executes poorly, manually inspects the expressions, then hand-codes an expression rewrite rule to fix the problem. Even if this ad-hoc approach could scale to thousands of programs, simply changing a compiler optimization level would call for a fresh set of rewrite rules. More worrying, these rules often rely on subtle two’s complement properties, but, since they are hand-coded, their correctness is difficult to verify.

1.2.4 Memory Access Analysis

Like other dynamic analysis techniques, dynamic symbolic execution can infer extra semantic information from program memory access patterns. Since pointers may be symbolic expressions in addition to classical concrete values, the symbolic executor has the opportunity to apply policy decisions that extend beyond a traditional concrete dynamic analysis approach. Such policies range from precise symbolic access tracking to symbolically shadowing program memory with rich metadata.

There is no obviously superior way to handle symbolic memory accesses. Ultimately, the access policy and workload greatly affects symbolic execution performance, making both underapproximation and overapproximation attractive options. Although supporting a multitude of policies would be advantageous, symbolic accesses introduce new edge cases that can easily corrupt state; new policies must be thoroughly tested. Likewise, built-in reasoning over symbolic state at the executor level quickly obscures the meaning of any sufficiently sophisticated access analysis.

Contemporary systems disagree on symbolic access policy, suggesting it should be configurable and tunable. These systems may underapproximate by concretizing pointers [58, 92], thus losing symbolic state, or precisely reason about accesses by maintaining symbolic state and possibly forking [28, 29, 52], incurring significant runtime overhead in the worst case. More complicated runtime policies that manipulate
or analyze accesses require deep changes to the executor’s memory subsystem [110], making development prohibitively difficult. All the while, there is no clean and isolated mechanism for dispatching memory accesses; all policies are directly coded into the interpreter, cluttering and destabilizing the core system.

1.3 Contributions

This dissertation applies methods for self-testing and validation to the issues outlined in Section 1.2. The core idea relies on the observation that executor components are interchangeable, have comparable results, rarely fail in the same way, and hence can test themselves; checking unreliable components against one another, or cross-checking, accurately narrows down intricate system bugs. Cross-checking the system establishes a solid base to build higher-order features with additional self-testing functionality.

The main contributions of this dissertation are:

1. The design, implementation, and evaluation of a cross-checked dynamic binary symbolic executor. The system uses a combination of deterministic replay, intermediate state logging, and model checking to automatically piecewise validate the correctness of symbolically executed paths. Validating correctness with cross-checking mechanically detects corrupted computation both near the point of failure in the target program path and close to the failing executor component. Cross-checking simplifies the tool debugging process by succinctly describing bugs otherwise missed in the deluge of data from analyzing programs by the thousand. Aside from detecting tool bugs, this is the first binary symbolic executor which can confirm the correctness of symbolically derived paths from the symbolic interpreter down to the hardware.

2. A self-testing system for symbolically executing floating-point code with soft floating-point libraries. To support symbolic floating-point data using only a bit-vector arithmetic constraint solver, the executor rewrites floating-point instructions to call into integer-only soft floating-point libraries. This approach
dramatically lessens the effort necessary for symbolically evaluating floating-point data over prior work by reusing code meant for emulating floating-point instructions on integer-only computer architectures. Furthermore, this approach is self-testing; since the underlying implementation is no different from any other code; symbolically executing each library with symbolic inputs produces high-coverage test cases for floating-point operations. Applying these tests against all soft floating-point libraries, floating-point constraint solvers, and hardware, uncovers serious library and floating-point constraint solver bugs.

3. An expression optimizer which automatically discovers and generates useful reduction rules. The optimizer exercises the hypothesis that programs are locally similar and therefore symbolically executing a large set of distinct programs will produce structurally different but semantically equivalent expressions. To this end, the optimizer learns reduction rules for rewriting large expressions to smaller expressions by searching a novel fingerprint based global store of expressions observed during symbolic execution. These rules are compatible with cross-checking and can be validated as they are applied at runtime. Unlike ad-hoc hand-written rewrite rules, every rule translates to a constraint satisfaction query for proving the rule’s correctness offline. Finally, the learned rules demonstrably reduce the number of constraint solver calls and total solver time when applied to a set of thousands of binary programs.

4. An efficient symbolically executed memory access mechanism and set of symbolically executed memory access policies. A novel memory dispatch mechanism, termed the symMMU, forwards target program memory accesses to special runtime code. This shifts otherwise expensive and complicated memory access policies away from executor scope to the target program scope which is better suited for reasoning about symbolic data. Policies become easier to implement and less susceptible to performance anomalies; the symMMU reimplemention of the default access policy both detects more program bugs and reports fewer false positives. Furthermore, multiple policies can be stacked to seamlessly compose new policies. Finally, new policies written against the symMMU extend the
symbolic executor’s functionality to use heavy-weight metadata without invasive executor changes; these policies include an access profiler, a heap violation checker, and lazy buffer allocation.
Chapter 2

The klee-mc Binary Program
Symbolic Executor

2.1 Introduction

This chapter outlines the background for symbolic execution along with the design of KLEE-MC, a machine code revision of the KLEE symbolic executor and the basis of this dissertation. The intent is to provide a context for the next chapters’ topics under one coherent overview. This chapter also describes and justifies important KLEE-MC features in detail which, although integral to the system’s operation as a whole, are primarily incidental to the content of other chapters.

The rest of this chapter is structured as follows. First, Section 2.2 provides a primer on symbolic execution and a survey of systems from past to present. Section 2.3 follows an example symbolic execution of a binary program using KLEE-MC. Section 2.4 highlights significant design choices made in the KLEE-MC system. Section 2.5 gives results from applying KLEE-MC to a large set of programs across three architectures. Finally, Section 2.6 makes a few concluding remarks.


2.2 Background

Conceivably, dynamic symbolic execution is a straightforward extension to normal program execution. Systems that mark inputs as symbolic then explore the feasible program paths have been known since at least the 1970s, but research stagnated, likely due to a combination of hardware limitations and high overheads. However, within the past decade there has been a flurry of developments in source-based symbolic execution systems [108]. Following this trend, many symbolic executors now target machine code (i.e., “binary”) programs, although with considerable difficulty.

2.2.1 Symbolic Execution

Symbolic execution is a dynamic analysis technique for automated test-case generation. These test cases describe paths to bugs or interesting program properties in complicated or unfamiliar software. Conceptually, inputs (e.g., file contents, network messages, command line arguments) to a program are marked as symbolic and evaluated abstractly. When program state reaches a control decision, such as an if statement, based on a symbolic condition, a satisfiability query is submitted to a theorem prover backed solver. For an if, when the symbolic condition is contingent the state forks into two states, and a corresponding predicate becomes a path constraint which is added to each state’s constraint set. Solving for the state’s constraint set creates an assignment, or test case, which follows the state’s path.

Figure 2.1 illustrates symbolic execution on a simple C program. On the left, a
program reads integers $x$ and $y$ from file descriptor 0 (conventionally, the “standard input”), then exits with a return code which depends on the input values. Assuming the \texttt{read} succeeds, the symbolic executor marks the inputs $x$ and $y$ as symbolic. Based on these symbolic inputs and given enough time, the executor follows every \textit{feasible} program path, illustrated by complete decision tree on the right. Each internal node represents a control decision, each edge describes a path constraint imposed by making a control decision, and each leaf is a path termination point. At the root of the tree is the first control decision, whether $x > 10$. Since $x$ is unconstrained, the executor forks the program into two states, adding the path constraint $\{x > 10\}$ to one and its negation, $\{x \leq 10\}$, to the other. The nodes in gray highlight a single complete path through the decision tree; the union of edges in the path define the path’s unique constraint set, $\{x \leq 10, y > 7\}$ in this case. Solving for a satisfying variable assignment of the constraint set gives concrete inputs which reproduce that path (e.g., $\{x = 10, y = 8\}$); this is the path’s test case. By completely traversing the decision tree the executor follows all possible control decisions for the program given inputs $x$ and $y$. By solving for the constraints leading to each leaf, the executor produces tests for every possible program path.

Unlike more established systems software such as databases, operating systems, or compilers, a complete design philosophy for symbolic executors remains somewhat ill-defined. Still, common themes and patterns emerge; these are reflected in the \texttt{klee-mc} description in Section 2.4. First, symbolic executors have a fundamental data type of \textit{expressions} over symbolic variables (§ 2.4.3) which precisely describe operations over symbolic inputs. The sources of these inputs, such as file or network operations, must be defined with a \textit{system model} (§ 2.4.5) to mark data symbolic when appropriate for the target program’s platform. When used in control decisions, such inputs form path constraints with solutions derived by a constraint solver based on some satisfiability decision procedure. From the solver’s solutions, a symbolic executor must generate test cases. Since the number of paths in a program may be infinite, a symbolic executor must choose, or schedule (§ 2.4.4), some paths to evaluate first before others.

Historically, the first symbolic execution systems appeared in the late 1970s [21,
The genesis of symbolic execution is most often attributed to King’s EFFIGY [77] system, perhaps due to his earlier work on program verification. On the other hand, several contemporary projects were similar in that they symbolically executed FORTRAN source code [39, 69, 109]. The SELECT [21] system, whose group equally acknowledged King (seemingly sharing preprints) but published slightly earlier, has the distinction of processing LISP. Regardless, due to the weaknesses of hardware and constraint solvers of the time, these systems were typically cast as enhanced interactive proof systems and were limited to analyzing small programs. The authors of CASEGEN [109], for instance, note execution rates of 10 statements per CPU-second and processing times of half a second for each constraint (limit 10 constraints). At a high level, all shared the basic concept of assigning states constraints by way of control decisions predicated on symbolic data. Likewise every system acknowledged modern problems in symbolic execution, such as handling language primitives, environment modeling, path explosion by loops, and indexing symbolic arrays.

Since the early 2000’s, symbolic execution has undergone a period of intense renewed interest. A large variety of new systems have emerged, most processing source code or intermediate representation code. These systems include support .NET [127], C [29, 58, 119], C++ [86], Java [75, 133], Javascript [4], LLVM [28], PHP [5], and Ruby [31] to name a few. Usually the aim of this research is divided between targeting symbolic execution of new types programs (e.g., through a different language or modeling new features), possibly detecting new types of bugs [10, 45, 87, 116, 122] and new algorithms for improving performance on expensive workloads [27, 35, 81, 121, 130, 134].

Certainly the technology behind symbolic execution has vastly improved, but it is unclear to what extent. In essence, too few programs are analyzed, it is difficult to verify the bugs in these programs, and the programs require significant manual configuration. Table 2.1 lists a small survey of the total programs tested under a variety of published symbolic execution systems. Although the average dearth of tested programs in practice may be justifiable due to type of code being tested (e.g., there are only so many operating system kernels), it raises serious questions regarding whether many techniques are effective overall or merely reflect considerable fine-tuning.
2.2.2 Binary Symbolic Execution

Since this work concerns dynamic binary symbolic execution, we pay special attention to its development. These systems hold the promise of supporting a large class of programs without the need for hand tuning. At its core, a binary symbolic executor combines the two well-established disciplines of symbolic execution and dynamic binary translation to run machine code programs. On the other hand, the extensive range of software compatible with a binary symbolic executor can make it a victim of its universality; obscure programs pave way for obscure executor bugs (often exacerbated by additional system complexity), the flood of new analysis data is unmanageable by hand, and designing optimizations on a per-program basis rapidly yields diminishing returns.

The convenience of applying dynamic analysis to unmodified program binaries led to binary symbolic execution [30, 34, 59, 92, 96, 124, 131]. Under binary symbolic execution, compiled executables are symbolically processed as-is and unmodified; there is no intrinsic need for recompilation, annotations, or special linking. If a program can run on its host system, it should be possible to subject it to binary symbolic execution provided a suitable symbolic system model to simulate host inputs.
Dynamic binary translators (DBTs) heavily influenced the design of dynamic binary symbolic executors. Unlike a static binary translator, a DBT translates machine code on demand as a target program runs; all relevant code falls under a DBT’s purview. A DBT translates machine code instructions into a simplified intermediate representation, possibly modifying the code’s semantics, before recompiling the data to the host’s instruction format. In practice, DBTs are deployed to transparently optimize programs [7] emulate code across computer architectures [15, 33], and dynamically instrument code for program analysis [89, 100, 125].

A typical binary symbolic execution system pairs a traditional symbolic executor with a dynamic binary translation front-end; the DBT converts a program’s machine code to simpler instructions suitable for symbolic execution. Many dynamic binary symbolic executors reuse off-the-shelf DBT software (machine decoding is non-trivial [103]) such as Pin [32], QEMU [34], and Valgrind [65, 92, 96]. On the other hand, it is possible to build a binary symbolic executor using static disassembly [131], but at the cost of some system flexibility. Similarly, some systems use custom decoders [30, 59], although admittedly still with some trouble [60].

The thousands of arbitrary machine code programs immediately available for binary symbolic execution presents a fresh set of challenges. A dynamic binary symbolic executor intrinsically comprises a complicated, sophisticated stack of unreliable components; it will break in unexpected ways. Worse, the complexity and depth of execution within such a system leaves little recourse for swiftly repairing observed defects. Even if the executor appears to perform as intended, understanding results from thousands of programs requires considerable expertise. Similarly, shortcuts for improving the executor, whether by focusing on several programs or piling on highly-coupled features that brazenly cut across separated concerns, either attains imperceivable gains across many programs or leaves the system incomprehensible and unworkable. This dissertation demonstrates that addressing these challenges directly makes it possible to rapidly develop a stable, novel, and mature binary symbolic executor.
2.3 Generating Tests with \texttt{klee-mc}

Although \texttt{klee-mc} works best as an automated system, using a collection of scripts for bulk processing, this section is meant to walk through the basic process for running a binary program under \texttt{klee-mc}. For this example, we will use the LLVM \texttt{opt} program with a command line argument \texttt{f}.

In most cases, \texttt{klee-mc} reads its initial guest program state and code from a program process snapshot. We will first acquire a snapshot and save it to disk; this will make the replay process straightforward. The snapshot loaded by a snapshotter, \texttt{pt\_run}. Once \texttt{opt} reaches its entry point, \texttt{pt\_run} saves the image to the file system:

\begin{verbatim}
VEXLLVM\_SAVE=1 pt\_run /usr/bin/opt f
\end{verbatim}

Next, we run the symbolic executor \texttt{klee-mc}. The command line flag \texttt{-guest-type} tells the system to read snapshot data from \texttt{guest\_last}, a symlink to the last saved snapshot in the current directory. Since \texttt{klee-mc} is primarily a research system, there are a great deal of command line arguments (391 as of writing); the default settings script uses about sixty flags to tune scheduler, solver, expression optimizer, memory management, and other aspects of the executor. The rest of the flags in the example control the state scheduler; a running state receives a five second quanta (\texttt{-batch-time=5}), states are scheduled based on fewest dispatched instructions (\texttt{-use-interleaved-MI}) and coverage of new branches (\texttt{-use-fresh-branch-search}), scheduler policy is decided by code coverage rates (\texttt{-use-ticket-interleave}), and a state’s quanta is reimbursed by covering new code (\texttt{-use-second-chance}).

\begin{verbatim}
klee-mc -guest-type=sshot -use-ticket-interleave -use-second-chance -use-batching-search -batch-time=5 -use-interleaved-MI -use-fresh-branch-search -
klee-mc
\end{verbatim}

\texttt{klee-mc} finds a memory fault on the 115th completed path. The error report includes a stack trace of the program state at the point of failure, the faulting memory address, and the nearest valid address:

\begin{verbatim}
Error: memory error: out of bound pointer
Stack:
#0 in _ZN4llvm13BitcodeReader17MaterializeModuleEPNS_6ModuleE+0x0
#1 in _ZN4llvm6Module14MaterializeAllEPSs+0x9
#2 in _ZN4llvm6Module25MaterializeAllPermanentlyEPSs+0x0
#3 in _ZN4llvm16ParseBitcodeFileEPNS_12MemoryBufferERNS_11LLVMContextEPSs+0x16
#4 in _ZN4llvm7ParseIREPNS_12MemoryBufferERNS_12SMDiagnosticERNS_11LLVMContextE+0xf4
\end{verbatim}
Next, to confirm the result independently of the symbolic executor, the test is replayed using kmc-replay. The kmc-replay utility runs the guest snapshot through a concrete dynamic binary translator, filling in system call data by reading off test data generated by klee-mc. We show the final 10 lines:

```
KMC_RECONS_FILES=1 kmc-replay 115 | tail -n10
```

```
[kmc-replay] Applying: sys=13 (rt_sigaction)
[kmc-replay] Applying: sys=2 (open)
[kmc-replay] Applying: sys=5 (fstat)
[kmc-replay] Applying: sys=17 (pread64)
Caught SIGSEGV but the mapping was a normal one @ 0x30
```

Replaying the test with the KMC_RECONS_FILES set means kmc-replay reconstructed files symbolically derived in the test case by replaying the test’s system calls back to the file system (e.g., open creates a file, pread64 writes data). This gives a file recons.0 which has contents that corresponds to a variable assignment, from a pread64 system call, in the test as given by the ktest-tool utility:

```
object 4: name: 'readbuf_1'
object 4: size: 8
object 4: data: 'BC\xc0\xde\nn\nn\nn'
```

Finally, putting the data back into opt confirms the memory fault natively on the host by using the gdb debugger to find a matching (demangled) backtrace:

```
gdb --args opt recons.0
```

```
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
.... in llvm::BitcodeReader::MaterializeModule (llvm::Module*) () from /usr/lib/libLLVM-3.4.so
(gdb) bt
#0  in llvm::BitcodeReader::MaterializeModule (llvm::Module*) () from /usr/lib/libLLVM-3.4.so
#1  in llvm::Module::MaterializeAll(std::string*) () from /usr/lib/libLLVM-3.4.so
#2  ... in llvm::Module::MaterializeAllPermanently (...*) () from /usr/lib/libLLVM-3.4.so
#3  in llvm::ParseBitcodeFile (...) () from /usr/lib/libLLVM-3.4.so
#4  in llvm::ParseIR (...) () from /usr/lib/libLLVM-3.4.so
#5  in llvm::ParseIRFile (...) () from /usr/lib/libLLVM-3.4.so
#6  in main ()
```
CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

2.4 Design of klee-mc

This section explains and justifies the major design decisions that went into KLEE-MC. On one hand, the design is partially motivated by simply having the system run at all; this highlights the changes necessary to retrofit a symbolic executor with machine code capabilities. On the other hand, the rest of the design focuses on robustness so that the system rarely breaks.

2.4.1 Program Snapshots

In order to execute a program, the symbolic executor must first have some way to load the program into an internal representation. KLEE-MC uses process snapshots, point-in-time copies of running program states, as its native initial representation of guest programs. A snapshot is a copy of all resources from a running process; the reasoning being that the symbolic executor can load a snapshot as a guest easier than starting from a bare program. A process snapshot contains the data necessary to reconstruct the state of a program running on a host system in KLEE-MC with minimal processing and inspection.

Snapshots sidestep many of the issues that would otherwise arise from building a custom program loader. If a program runs on a host, KLEE-MC can likely record and run its snapshot, ultimately giving the system a wide program reach. Furthermore, by decoupling the snapshotting process from the symbolic execution phase with independent snapshotter programs, KLEE-MC symbolically executes programs from platforms where it cannot run natively. Since snapshots are immutable, they resist side-effects and non-determinism from new libraries, different linkers, and address space randomization over multiple runs, which becomes relevant for replay cross-checking in Chapter 3.

Snapshot Structure

Snapshots save the machine configuration of a running program process from a live system for later use. The system snapshots a binary program by launching it as a process and copying out all relevant resources; the snapshot has all data necessary
CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

...to rebuild the state of a running program within KLEE-MC for symbolic execution. Likewise, to avoid complicating the executor with many special cases or making snapshots too onerous to build without sophisticated infrastructure, snapshots must be somewhat generic in to support a variety of architectures and system platforms.

Snapshot data is structured as a directory tree of files representing resources loaded from the process. The format is reasonably platform agnostic and portable; most snapshot resources describe only the machine configuration information necessary for programs to run with few, if any, strict assumptions about the host operating system. These resources include:

**Registers.** The register file contains the stack pointer, program counter, floating-point registers, vector registers, and general purpose registers. For processes with threads, user registers for each thread (sometimes called the thread’s “context”) are stored as individual files in a thread directory. Additionally, this includes system registers not directly accessible or modifiable by the program but necessary for program execution (e.g., segment registers and descriptor tables for thread local storage).

**Memory.** Code and data from the process image are stored as files in a directory containing all process memory. Each file represents a contiguous memory range from the process, thereby keeping total file count low, reducing overhead, while exposing the logical address space organization at the file system level, simplifying tooling. These ranges include all libraries, heaps, and stacks and ensures all data remains consistent from run to run. Memory ranges are disjoint, page-aligned, have access permissions, and occasionally have names (e.g., memory mapped files); this extra metadata is kept in a memory map information file.

**Symbols.** Assigning meaningful names to ranges of memory enhances the readability of the symbolic analysis. Symbols from the program binary give these names to ranges of memory. Symbol information is extracted from the process either by analyzing the backing files (e.g., the program binary and its libraries) or through system symbol facilities. The symbols are listed by symbol name and address range, and are stored in a single text file. If no symbol information is available (e.g., stripped binaries), the snapshot is still valid, but functions will be labeled by memory address.

**Platform Specifics.** Although snapshots ideally keep process details platform
agnostic, sometimes it is necessary to store data specific to a particular runtime model. Information specific to the process host’s operating system is stored to the snapshot’s platform-specific directory. For instance, the Windows pointer encoding key, which decodes function pointers in core libraries at runtime, is stored as a platform attribute. It is the responsibility of the symbolic system model to access the platform files. In the future, richer models may use additional platform details; keeping platform-specific information in one place will ease the development of such models.

**Snapshotter**

Each snapshot is collected by a snapshotting program, the *snapshotter*. Since snapshots have few dependencies on the rest of the KLEE-MC system, snapshotter code can run on more systems than KLEE-MC itself (e.g., 32-bit architectures, non-Linux operating systems), thus expanding the executor’s program reach without having to port the system to new hosts. In fact, while snapshots may be portable, the snapshotter itself is not; fortunately, the host features necessary to build a snapshotter correspond with features necessary to build a program debugger and therefore are common to modern operating systems. Additionally, since points of interest in a process’s execution may vary from program to program, the snapshotter has flexibility in how and when it takes a snapshot.

A snapshotter stops a program and reads its resources. In this sense, a snapshotter is similar to a breakpointing debugger, which must stop a program at user-defined locations and read off program state when requested. For Linux, a debugger controls the process with `ptrace` and reads process information from `procfs`. For Windows, the functions `DebugActiveProcess`, `DebugBreakProcess`, and so forth control the process like `ptrace` whereas calls to functions like `OpenProcess`, `EnumProcessModules`, and `VirtualQueryEx` access process information like `procfs`.

We found several modes for acquiring a snapshot useful in practice:

**Program Entry Point.** The first snapshot is taken immediately prior to dispatching the first system call past the program’s entry point. This captures the entire program’s system call trace starting from its entry.

**Launch and Wait.** The snapshotter launches the process and traces system calls.
Once the process dispatches the specified system call, it is snapshotted. Launching and waiting helped when snapshotting Linux programs compiled as libraries because the entry point is chosen by the runtime linker. The snapshotter waits for the first \texttt{munmap} since the linker only unmaps memory immediately before calling the library initialization function.

**Process Attach.** Begin snapshotting an already active program. Attaching is useful for tracing interactive and long-running programs (e.g., web servers, network file servers, and graphical user interfaces); the snapshot often begins at a system call where the program is blocked and waiting for some input (e.g., \texttt{select}).

**Attach and Wait.** Snapshot an already active program once it encounters a particular system call. Waiting for a system call is useful for capturing specific runtime events such as opening new files.

### 2.4.2 Dynamic Binary Translation

A dynamic binary translator converts a running program’s machine instructions to a practical intermediate representation (IR) on demand. \texttt{klee-mc} uses a DBT to translate its guest code from machine instructions, often x86-64 code, to the LLVM IR for symbolic execution. The system converts machine code to LLVM IR in two stages: first from machine code to the VEX IR using the valgrind \[100\] front-end, then from the VEX IR to LLVM IR using a custom translation pass. In addition to applying the DBT to the symbolic executor, we developed a just-in-time interpreter (JIT) which compiles the LLVM IR back to machine code using the LLVM JIT then runs the code to test the correctness of concrete execution; the JIT independently confirms symbolically derived test cases to rule out symbolic executor errors when processing LLVM IR from genuine machine code translation errors. For \texttt{klee-mc} to work with the DBT, we significantly modify \texttt{klee}’s instruction dispatch system to support jumping between dynamically generated LLVM functions that model basic blocks of machine code.
### CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

<table>
<thead>
<tr>
<th>C Machine (x86-64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(void)</td>
</tr>
<tr>
<td>{ char *bad=NULL;</td>
</tr>
<tr>
<td>int sym = 0;</td>
</tr>
<tr>
<td>read(0, &amp;sym, sizeof(sym));</td>
</tr>
<tr>
<td>if (sym != 0) *bad=1;</td>
</tr>
<tr>
<td>return 0; }</td>
</tr>
</tbody>
</table>

```c
main:
400450: sub $0x18,%rsp
400454: xor %edi,%edi
400456: mov $0x4,%edx
40045b: lea 0xc(%rsp),%rsi
400560: movl $0x0,0xc(%rsp)
40056d: callq read
40056d: mov 0xc(%rsp),%eax
400571: test %eax,%eax
400573: je 40057d
400575: movb $0x1,0x0
40057d: xor %eax,%eax
40057f: add $0x18, %rsp
400583: retq
```

**VEX**

- IMark(0x40056D, 4)
- \( t6 = 32\text{Uto64}(\text{LDle}(\text{GET}(32), 0x\text{C})) \)
- PUT(0) = t6
- IMark(0x400571, 2)
- PUT(128) = 0x13:164
- \( t12 = 32\text{Uto64}(\text{And}(64\text{to32}(t6), 64\text{to32}(t6))) \)
- PUT(136) = t12
- PUT(144) = 0x0:164
- IMark(0x400573, 2)
- PUT(168) = 0x400573:164
- if (CmpEQ64(Shl64(t12, 0x20), 0x0))
  - goto 0x40057D
  - goto 0x400575

**LLVM**

```llvm
define i64 @sb
  %0 = getelementptr %regTy*, i32 4
  %RSP = load i64* %0
  %2 = add %1, 12
  %loadP = inttoptr %2 to i32*
  %3 = load i32* %loadP, align 8
  ... store 4195699, 164* %RIP
  %9 = shl 164 %8, 32
  %10 = icmp eq %9, 0
  br i1 %10, %exit_then, %exit_else
exit_then: ret 0x40057d
exit_else: ret 0x400575
```

Figure 2.2: Stages of translation annotated by propagation of symbolic state.

**VEX-to-LLVM**

This section describes the process for translating from machine code to LLVM code in klee-mc’s VEX-to-LLVM translation layer. As a guiding example, Figure 2.2 illustrates the stages of code translation in the DBT, showing which instructions would process symbolic data under symbolic execution. The example goes from human-friendly C source, a snippet of its compiled machine code, the snippet’s VEX IR, then finally, the LLVM code suitable for symbolically execution.

The C code in the top left of Figure 2.2 is a small program that crashes given a non-zero input. The program issues a `read` system call that writes data to the `sym` variable. If the `read` succeeds and stores a non-zero value, the code dereferences
the null pointer bad, causing the program to crash. Otherwise, the program exits normally. If symbolically executed, sym is marked symbolic when applying the read system call. Next, the initial program state forks into two states at the if, one where sym is assumed zero and another where sym is assumed non-zero, to explore both feasible paths; the zero state exits normally and the non-zero state crashes.

Compiling the C code produces machine code. In this case, the C code was compiled on an x86-64 architecture KLEE-MC host machine. A call to read dispatches a system call, updating sym (stored in the stack at 0xc(%rsp)). The test instruction constructs the predicate for the if statement by checking whether the register %eax, loaded from sym, is zero. The conditional jump instruction, je, forks the state under symbolic execution; the non-zero state falls through to the illegal memory access at 0x400575 and the zero jumps to an exit. Clearly, it is less than ideal to symbolically execute code in this format due to the x86-64 instruction set’s CISC origins; there are hundreds of instructions along with many encodings to complicate matters. Section 3.5.2 demonstrates the difficulty of merely decoding machine instructions; direct symbolic execution of machine code would prove even more challenging.

The VEX IR representation maps machine instructions into a simpler machine model. The VEX code starts immediately after the read call; VEX IR decodes machine code into basic blocks that only permit jumps as exits from the code block. Like the machine code example, the VEX IR reads sym from the stack; it assigns it to a temporary register t6 that is also stored to %eax through PUT(0). Unlike machine code, the VEX IR can have arbitrary length expressions in its instructions, as demonstrated by its emulation of the test instruction, which is stored to t12. The VEX IR constructs the branch predicate with the CmpEQ64 instruction. If the VEX IR were symbolically executed, as in some systems [65, 92, 96], the state would fork on the if, one jumping to the instruction at 0x40057d and the other jumping to the instruction at 0x400575. Although the VEX IR is more wieldy than machine code, the language has over nine hundred instructions, making it an unsuitable target for a small symbolic interpreter core.

Rather than attempting to symbolically dispatch every sort of VEX IR instruction, including those that arise infrequently, the system translates VEX IR to the
simpler LLVM IR in order to provide excellent support for a handful of instructions. The translation from VEX to LLVM is straightforward; large VEX expressions are flattened to LLVM single-operation semantics. For instance, the \texttt{CmpEQ64} expression from the example lowers to \texttt{shl} and \texttt{icmp} LLVM instructions. Each VEX basic block translates to an LLVM function. An LLVMized basic block function takes one argument, the in-memory register set, and returns the program counter of the next basic block to be processed by the program interpreter. Symbolic data flows through the LLVM code like the VEX code; the \texttt{icmp} builds the condition predicate and the state forks on the conditional branch \texttt{br}.

\textbf{Just-In-Time Interpreter}

The JIT interpreter controls all concrete details for the DBT subsystem in \texttt{klee-mc}. All binary specific details that relate to concrete execution belong to the JIT; this keeps the symbolic execution portion of \texttt{klee-mc} uncontaminated by most binary minutia. These details include managing concrete program data corresponding to a target program, collectively known as the \textit{guest}, such as snapshots, translated code caches, symbols, and application binary interfaces. In addition to describing guest data, the JIT supports running guests concretely by executing basic blocks like a traditional DBT system. Furthermore, through careful design, the JIT can replay symbolically derived test cases independently of the symbolic executor.

The JIT interpreter concretely executes a snapshot by dynamically recompiling its code with the VEX to LLVM translation machinery. First, the JIT loads the snapshot’s memory regions in-place, or \textit{identity mapped}, into its process; this mapping method avoids the need for any address translation in the guest code since guest addresses are fixed from host to JIT. Identity mapping relies on linking the JIT’s executable to an unusual address and address space randomization so that the JIT’s initial address space is disjoint from guest’s, eliminating address conflicts. Next, the JIT loops, executing basic block after basic block. The JIT passes the guest’s program counter address $a$ to the DBT to retrieve an LLVM function which it compiles to a machine code function $f_a$ using the LLVM JIT. The compiled function $f_a$ has full access to the JIT’s memory but, since $f_a$ is derived from the guest and therefore only
“knows” about guest data, memory accesses rarely, if ever, touch JIT memory; either \( f_a \) manipulates the guest state or it crashes the guest. After the JIT runs \( f_a \), the function returns a new address \( b \) which is used as the base for the next basic block to run. This process of dispatching basic blocks continues until the guest crashes or exits normally and is the basis of executor cross-checking in Section 3.3.3.

Whenever a basic block calls out to the operating system with a system call, the JIT passes the arguments to a system call dispatcher. For basic execution of program snapshots, as if they were running on the host system, a pass-through system call dispatcher passes system calls straight to the host operating system. Snapshots from other architectures, but still running Linux, use a system call translator based on the QEMU [15] user-level emulation system call translation layer. The \texttt{kmc-replay} utility demonstrated in Section 2.3, which concretely replays tests with the JIT, has a system call dispatcher that fills in system call results based on a log (§3.3.1) generated along with each test case.

**Symbolic Execution Model**

\textsc{klee} in its original form made several major assumptions based on LLVM bitcode that interfere with retrofitting a DBT to the system. First, it expects that all code is loaded up front, which ignores dynamically loaded libraries, self-modifying code, and problems associated with accurately dissembling variable-length opcode instruction sets such as x86-64. Next, the memory model relies on precise variable and buffer information; it cannot generally reason about anonymous data sections from snapshots without either having expensive and excessively large objects or raising errors on page-straddling accesses. Finally, the environment assumes programs call to system model through directly linked functions, rather than an indirect system call mechanism which needs a custom dispatch path.

\textsc{klee-mc} uses a new basic block aware dispatch model to support dynamic binary translation. Since the DBT translates machine code basic blocks to LLVM functions that return the next program counter, \textsc{klee-mc} transfers control from one basic block to another when it encounters an LLVM \texttt{ret} instruction. When the function is DBT bitcode, \textsc{klee-mc} inspects the return value to determine the next code block to run
and uses the block exit code (e.g., call, return, jump, system call), as given by VEX, to decide how to update the call stack. This separate mode for DBT code is necessary because the on-demand translation of code forces the system to process basic blocks individually; DBT basic blocks cannot be stitched together into KLEE-compatible LLVM programs in general. To let DBT-generated basic blocks call LLVM bitcode, KLEE-MC defaults to KLEE behavior when the returning function is native LLVM bitcode and DBT behavior when the returning function is DBT bitcode.

Snapshots represent a program’s address space as a set of disjoint page-aligned memory regions. This format conflicts with the KLEE model which accurately tracks all memory objects (e.g., variables, arrays, heap data), at the byte level. Although analysis of debugging information and memory heap data could reveal precise memory object data could potentially provide precise memory object layout information, there is no guarantee such data will be available in a snapshot. Since totally accurate memory object modeling is a lost cause and best-effort modeling implies false positives, KLEE-MC loads memory regions page-by-page into the initial symbolic state and relaxes the KLEE memory model to permit contiguous cross-object memory accesses.

User-level binary programs communicate with the underlying operating system through system calls whereas the KLEE model assumes programs communicate with the system through library calls. While KLEE links model code with the program and lets function calls bridge the gap between program and environment model, KLEE-MC must explicitly handle system calls due to differing calling conventions. To issue a system call, per VEX semantics, a basic block function sets its exit code to indicate its intent to make a system call. The KLEE-MC basic block dispatcher intercepts this system call exit code and redirects the state to call out to a symbolically executed system call emulation bitcode library.
2.4.3 Expressions and Solving

Constraint solving on expressions is integral to symbolic execution systems from branch feasibility checks to test case generation. Adapting a symbolic executor to effectively run machine code means optimizing for new, complex expressions and configuring its solver to deal with slow-running queries from difficult constraint sets. Given the unpredictability of instructions from arbitrary binaries, solver failures should be somewhat expected, and fallbacks should be available to recover expensive paths.

Expressions

Expressions form the core symbolic data type of the symbolic executor. A symbolic expression is nearly indistinguishable from the sort of expressions found in mathematics or programming languages; an expression represents a sequence of closed operations (e.g., two’s complement arithmetic) on other expressions, often illustrated as a tree or described with a grammar (KLEE’s is in Figure 5.3), working down to a class of expression atoms, such as numbers or variables, with no further structure. Effective expression management uses rewrite rules based on operation identities to keep expressions small and efficient.

Building a new expression provides a convenient opportunity for optimization. In KLEE terminology, all expressions are constructed via one of several expression builders, depending on the system configuration. For instance, a requesting a new expression (Sub x x) from an optimizing builder may return the constant 0 if the builder knows to apply the identity $x - x = 0$. Aside from obvious constant folding, where operations on constants are folded into a single constant, the optimizing builder canonicalizes expressions and applies ad-hoc rules based on two’s-complement identities to reduce expression size when possible.

Most expressions in KLEE-MC correspond to a sequence of LLVM instructions, derived from machine code, operating on symbolic data. Due to the sophistication of modern optimizing compilers and intricacies of modern machine instruction sets, these instruction sequences can significantly differ from the original source and its intermediate bitcode representation. The compiler inserts vector instructions, collapses
conditions to arithmetic, twiddles bits, and so on to improve performance for a target architecture. These machine optimizations stress the expression builder in ways never exercised by LLVM bitcode, making new rules necessary for machine code.

To illustrate the new class of expressions common to machine code, consider a reciprocal multiplication, which replaces an expensive division operation with a cheaper multiplication operation, observed in practice:

\[(\text{extract}[127:67] \text{(bvmul bv14757395258967641293[128] x)})\]

The expression multiplies \(x\) by a “magic” constant, the 64-bit reciprocal for 10, then extracts the bits starting from bit 67 out of the result. On hardware, multipliers are significantly cheaper than dividers, making this an excellent strength reduction. On the other hand, to a constraint solver this causes queries to time out since a 128-bit multiply is much more costly than solving for a 64-bit divide. Undoing this hardware-specific strength reduction by replacing the expression with \(x/10\) halves the bit-width thereby making the problem easier for the solver so that such queries complete in reasonable time.

Although slowly adding ad-hoc rules to the expression builder would continue to work to some extent, it is far from satisfactory for scaling up to tens of thousands of programs. Early experiments with the expression builder indicated many complicated boolean expressions were in fact tautologies that could be replaced with constants, suggesting useful rewrite rules could be discovered automatically. Chapter 5 discusses a method in this vein that KLEE-MC uses to derive general rules from program traces to improve symbolic execution performance. Given that there are many more machine-generated rules than ad-hoc rules and that these rules may still have subtle bugs, KLEE-MC also supports inductive rule optimization checking at runtime to improve the overall integrity of the expression optimization system (§ 3.3.5).

Solver Stack

All path constraints, as given in Section 2.2.1, are expressions. The set of path constraints for each state is called a constraint set; every unique constraint set translates
to a set of assumptions $C$ that, along with a challenge formula expression $q$ such as a candidate path constraint, make up a solver satisfiability query $(C, q)$. The symbolic executor’s solver decides whether the conjunction of the query’s assumptions and formula is satisfiable, computing satisfying assignments as needed.

The executor organizes its solver as a sequence of processing phases pass queries down to a general purpose constraint solver. Since the constraint solver ultimately computes solutions to the boolean satisfiable problem, for which there is no known efficient algorithm, it tends to either finish quickly by exploiting the query’s structure or time out from exponential blow-up; KLEE-MC tries to both optimize for the fast path and robustly recover from failures. Although all solver queries could be directly sent to a constraint solver, the executor must inefficiently serialize a query to match the solver’s input format. By caching solutions from prior solver requests, transforming queries with optimizations to reduce overhead, and recognizing trivial solutions to certain classes of queries, the stack can often avoid making an expensive solver call. As a solver stack can become quite complicated and new stack components can subtly fail, KLEE-MC supports cross-checking multiple stacks at runtime for mismatches (§3.3.5).

As a whole, machine code programs need a robust solver stack. When running binaries, we observed KLEE’s solver stack would hang on cache lookups from traversing huge expressions, the internal third-party solver would crash with call stack overflows, and time-outs were too generous for acceptable forward progress. This experience indicated failure should be expected and that robustness is more important than various performance hacks. Figure 2.3 illustrates the default KLEE-MC solver organization as informed by this experience; queries are processed by a few essential passes, then passed to the solver through a robust, solver independent, conduit. For a query $(C, q)$, assumptions in $C$ independent from the formula $q$ are removed (as in KLEE), query results are cached using a new built-in hashing scheme with constant look up time, and the executor communicates with the solver through isolated processes.

**SMTLIB Serialization.** KLEE-MC communicates with an isolated STP [55] theorem prover process through a forked copy of its own process and UNIX pipe. When the executor makes a solver call, it forks itself to create a serialization process
and launches a new STP process. The serialization process streams the query as an SMTLIB request to the STP process over an anonymous pipe. The parent process then reads off the result from STP and terminates its children. Although the parent process could write the SMTLIB query to the STP process directly without forking, serialization of large queries causes the executor to hang; forking lets the parent safely terminate the serializer if it times out. By totally separating the solver from the executor, it is easier to swap it for other solvers; the SMT pipe solver supports five solvers [13, 25, 49, 51, 55] with very little specialized code.

Although some systems propose supporting multiple solvers [105] so the best solver for a query finishes first, we found linking multiple solvers to the executable made the system difficult to build and keep up to date. Originally strong integration between the executor and solver seemed as though it would lead to improved performance. However, after writing two new custom solver interfaces, it became clear performance was too unpredictable among solvers. Furthermore each solver had its own quirks over how to best build a query, making tuning all but necessary.

**Query Hashing.** KLEE-MC caches query satisfiability and values with an array insensitive hash computed at expression build time. Unlike KLEE’s caching solvers, array insensitivity lets queries with equivalent structure but different array names share cache entries; each entry represents an equivalence class of all α-conversions. Hashing boosts the hit rate, and therefore performance, at the expense of soundness by ignoring distinct names. Although hashing is imprecise, unsound collisions were never observed when tested against an array sensitive hash during program execution. Furthermore, if an unsound path due to a hash collision were followed to completion,
then solving for its test case, which must go through the solver, would detect an inconsistent constraint set.

Array name insensitivity improves performance when a program loops, constructing the same queries, but with new array names (e.g., `readbuf_1` and `readbuf_2`). Furthermore, when only hash equality matters, cache look ups will not hang on deep comparison of large expressions. For persistence, query hashes are stored in sorted files according to query solution: sat, unsat, value, or timed out.

KLEE itself supports a caching proxy solver, `solverd`, with a persistent hash store (likely a descendent of EXE’s cache server [29]). Unfortunately, it suffers from several design issues compared to an in-process query hash cache. First, communicating with `solverd` incurs costly interprocess communication through sockets. Next, the query must be built up using STP library intrinsics, totally serialized to the SMTLIB format in memory, then sent through a socket. For large queries, this serialization may crash STP, take an inordinate amount of time, or use an excessive amount of memory. Finally, when `solverd` MD5 hashes the entire SMTLIB string it misses queries that are equivalent modulo array naming and must create a new entry.

**State Concretization**

When the solver times out on a query, there are three obvious options. One, the time limit can be boosted, dedicating disproportionate time to costly states which may not even exhibit interesting behavior. Two, the system can terminate the state, throwing away potentially interesting child paths. Three, the system can drop the state’s symbolics causing the time out, hence salvaging the remaining path. KLEE-MC pursues this third option by concretizing state symbolics into concrete data.

Since state concretization is a fall-back mechanism for failed queries, every concretization begins with a failed query \((C, q)\) on a state \(S\). Assuming the executor did not corrupt any path constraints, the constraint set has a satisfying assignment \(\sigma\). Knowing the solver already proved \(C\)’s satisfiability, it can be assumed the solver computes \(\sigma\) without timing out. Likewise, if \(C\) succeeds but \(C \land q\) causes the solver to fail, then only \(q\) needs to be concretized to make forward progress. To concretize \(q\), the executor constructs a variable assignment for only variables in \(q\), \(\sigma_q = \sigma \cap q\), then
applies this assignment to all symbolic data in \( S \), yielding the partial concretization \( S_{\sigma_q} \). Note that \( \sigma_q \) need not satisfy \( q \), only that \( \sigma_q \) must assign concrete values to all of \( q \)'s variables in addition to satisfying \( C \).

Replacing symbolic data in \( S \) with \( \sigma_q \) to get the state \( S_{\sigma_q} \) must be exhaustive in order to eliminate all references to \( q \)'s variables. The executor therefore applies \( \sigma_q \) to the following parts of the state that may have symbolic data:

- **Memory Objects** – The executor scans all state data memory for symbolic data with variables from \( q \). To speed up scanning, the executor only inspects objects which contain symbolic expressions; concrete pages are ignored.

- **LLVM Call Stack** – The LLVM call stack for \( S \) contains the values for temporary registers in LLVM code being evaluated by the executor. Usually one of these values will contain \( q \) as the result of an \texttt{icmp} expression.

- **Constraints** – The state’s constraint set must reflect the concretization of terms in \( q \) or future queries will be underconstrained. The executor applies the assignment \( \sigma_q \) to \( C \) to get \( C_{\sigma_q} \), the constraint set for \( S_{\sigma_q} \).

- **Arrays** – The constraints on elements in \( q \) vanish in \( C_{\sigma_q} \) but must be recalled to produce a correct test case. To track concretized constraints, the state \( S_{\sigma_q} \) saves \( \sigma_q \) as a set of concrete arrays along with any remaining symbolic arrays.

The idea of concretizing state during symbolic execution is common. \( \text{S}^2\text{E} \) [34] uses lazy concretization to temporarily convert symbolic data to concrete on-demand for its path corseting feature. SAGE [59] concretizes its states into tests and replays them to discover new states. \textsc{klee} concretizes symbolic data where symbolic modeling would be costly or difficult to implement. \textsc{klee-mc} is perhaps the first system that uses partial concretization to recover from solver failure.

### 2.4.4 Scheduling

A symbolic executor generates many states while exploring a path tree. The number of states almost always outnumbers the number of CPUs available to the symbolic
CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

executor; the system must choose, or schedule, a set of states to run for a given moment. Some states will cover new and interesting program behavior but others will not; choosing the best states to schedule is the state searching problem.

Constructing good search heuristics for symbolic execution remains an open problem. Although a search heuristic can conceivably maximize for any metric, the standard metric is total code coverage. Many heuristics to improve code coverage rely on overapproximations similar to those developed for static analysis. Unfortunately, static analysis techniques fare poorly on DBTs since not only is code discovered on-demand, providing a partial view of the code, but also because recovering control-flow graphs from machine code demands additional effort [8].

Instead, KLEE-MC ignores the state searching problem in favor of throughput scheduling. In this case, KLEE-MC tries to dynamically schedule states to improve total coverage based on prior observation. Two policies are worth noting:

**Second chances.** If a state covers new code, its next scheduled preemption is ignored. The reasoning is that if a state covers new code, it is likely to continue to covering new code. If a state is preempted, it is possible it will never be scheduled again, despite showing promise.

**Ticket interleaved searcher.** Given several schedulers, an interleaved searcher must choose one scheduler to query. The ticket interleaved searcher is a lottery scheduler [129] which probabilistically chooses a scheduler weighted by the number of tickets a scheduler holds. The searcher assigns tickets to a scheduler when its states cover new code and takes tickets when the states cover only old code. The reasoning is that if a scheduler is doing well, it should continue to select states to run.

### 2.4.5 Runtime Libraries

KLEE-MC extends the executor with symbolically executed runtime libraries to improve reliability by avoiding hard-coded modifications when possible. Runtime libraries are isolated from the executor’s inner-workings so they occasionally need new interpreter intrinsic calls to communicate with the underlying system. Likewise, runtime libraries call into convenience library code which builds library intrinsics from
lower level interpreter primitives. Binary guest programs invoke these libraries indirectly through intermediate code intrinsics, aspect-like function instrumentation, system calls, rewritten instructions (§ 4.3.2), or data-dependent instruction paths (§ 6.3).

**Interpreter Intrinsics**

A symbolically executed runtime library explicitly communicates with the symbolic executor through interpreter intrinsics. An interpreter intrinsic, much like an operating system call, runs in the executor’s context and safely exposes symbolic executor resources to guest code via an isolated, controlled conduit. Additionally, interpreter intrinsics serve as simple symbolic execution primitives from which runtime code can build more complicated operations that reason about symbolic state.

Choosing the right interpreter intrinsics presents challenges similar to designing any interface. There are two general guiding principles that worked in practice. First, a good intrinsic is primitive in that it exposes functionality that cannot be replicated with library code. Second, since the executor implementation is non-preemptible, a good intrinsic should complete in a reasonable amount of time. Often this reasonable time constraint implies that calling an intrinsic should execute at most one solver call. An example bad intrinsic is KLEE’s `malloc` call. The `malloc` intrinsic allocates $s$ bytes of memory, which requires executor assistance, but also lets $s$ be symbolic, issuing several solver calls in the executor context (poorly, noting “just pick a size” in the comments) which could otherwise be handled by runtime code. KLEE-MC instead has an interpreter intrinsic `malloc_fixed`, which allocates a constant number of bytes and uses $5\times$ fewer lines of code, called by a library intrinsic `malloc` that forks on symbolic allocation sizes.

Table 2.2 lists some of the more interesting new intrinsics used by KLEE-MC. None of these intrinsics inherently reason about machine code, hence the `klee` prefix, but were still integral to KLEE-MC to build library intrinsics (Figure 2.4 gives an example). Many of these intrinsics rely on a predicate $p$ argument; these predicates are constructed explicitly with the `mk_expr` intrinsic. Although instruction evaluation implicitly builds expressions, the `mk_expr` intrinsic is useful for avoiding compiler
Intrinsic Call | Description
---|---
__klee_mk_expr(op, x, y, z) | Make an op-type expression with terms x, y, z
__klee_feasible(p) | Return true when p is satisfiable
__klee_prefer(p) | Branch on p, preferring true path when feasible
klee_get_value_pred(e, p) | Get concrete value of e assuming predicate p
klee_report(t) | Create test case without terminating state
klee_indirectn(s,...) | Call intrinsic s with n arguments
klee_read_reg(s) | Return value for executor resource s

Table 2.2: Selected interpreter intrinsic extensions for runtime libraries.

optimizations (i.e., branch insertion) that might cause unintended forking or to avoid forking entirely (i.e., if-then-else expressions). The feasible intrinsic tests whether a predicate p is feasible; prior to introducing this intrinsic, runtime code could branch on p, thus collapsing p to either true or false. Likewise, prior to get_value_pred runtime code could not get a concretization of e without adding p to the state’s constraint set. Both indirectn and read_reg decouple guest code from the executor; indirect lets machine code late-bind calls to intrinsics, useful for unit tests, and read_reg lets runtime code query the executor for configuration information, an improvement over KLEE’s brittle method of scanning a library’s variable names at initialization and replacing values.

Library Intrinsics

Whereas interpreter intrinsics expose primitive functionality through a special executor interface, library intrinsics provide richer operations with runtime code. Library intrinsics may be thought of the “standard library” for the symbolic executor runtime; runtime code can independently reproduce library intrinsic functionality but is better off reusing the code already available. Furthermore, following the guidelines for interpreter intrinsics, library intrinsics define features that could conceivably be an interpreter intrinsic, but would otherwise be too costly or difficult to implement directly in the executor.

Library intrinsics primarily assist symbolic execution by managing symbolic data when multiple solver calls are needed. Two examples are the klee_max_value and
int klee_get_values_pred(
    uint64_t expr, uint64_t* buf,
    unsigned n, uint64_t pred)
{
    unsigned i;
    for (i = 0; i < n; i++) {
        /* exit if all values that satisfy 'pred' are exhausted */
        if (!__klee_feasible(pred)) break;
        /* get next value that satisfies 'pred' */
        buf[i] = klee_get_value_pred(expr, pred);
        /* remove current value from possible satisfying values */
        pred = klee_mk_and(pred, klee_mk_ne(expr, buf[i]));
    }
    return i;
}

Figure 2.4: A library intrinsic to enumerate predicated values

klee_fork_all_n intrinsics. The first computes the maximum concrete value for an expression with binary search driven by solver calls with __klee_feasible. The second forks up to n states by looping, making a solver call to get a concrete value c for an expression e and forking off a new state where c equals e. Additionally, runtime code can take on a support role for interpreter intrinsics, such as in the case of malloc: the library intrinsic processes any symbolic sizes then passes a simpler case with concrete inputs to an interpreter intrinsic.

As a concrete example for how interpreter intrinsics and library intrinsics interact, Figure 2.4 lists the library intrinsic klee_get_values_pred. This function (used in Section 6.4.1’s ite policy) enumerates up to n feasible and distinct values of expr may take assuming the calling state’s path constraints and an initial predicate pred, storing the results in buf. As the function loops, it issues a solver call testing whether the predicate is satisfiable (using __klee_feasible). issues another solver call through klee_get_value_pred to get a value c for expr, then adds a condition to the predicate that the next value cannot be c (using klee_mk_and convenience macros klee_mk_and and klee_mk_ne). The function returns once all values for expr are exhausted or n values are computed, which ever comes first. By carefully intrinsic design, this potentially long-running function is preemptible by other states and never forks or accrues additional state constraints.
Intermediate Code Intrinsics

Intermediate representations can lack appropriate expressiveness to succinctly describe complex code semantics. For instance, LLVM presently has 81 platform-independent intrinsics that describe variable arguments, garbage collection, stack management, bulk memory operations, and so forth that are not part of the LLVM machine description. Often such intrinsics affect program state and must be handled to correctly execute the program.

Like LLVM, the VEX intermediate representation has its own set of “helper” intrinsics. When VEX translates machine code to VEX IR, it emits helper calls to handle the more complicated instructions. These helpers compute the value of the eflags register, count bits, and dispatch special instructions such as cpuid and in/out. Helper calls depend on a sizeable architecture-dependent helper library (2829 lines for x86-64) that performs the computation instead of inlining the relevant code. KLEE-MC has a copy of this library compiled as LLVM bitcode. DBT basic blocks call into this runtime library like any other LLVM code.

System Models

A system model library supplies an interface between the guest program and the symbolic executor that simulates the guest’s platform. The modeled platform can range from emulated libc and POSIX calls to a simulated low-level operating system call interface; KLEE-MC’s system model replaces system call side effects with symbolic data. Specialized knowledge of platform semantics encoded in a system model library defines when and where a state takes on symbolic inputs.

Due to the variability of system libraries, modeling the binary program platform at the system call level handles the most programs with the least effort. Although there are opportunities for higher-level optimizations and insights into program behavior by modeling system libraries instead of system calls, a binary program is nevertheless free to make system calls on its own, necessitating a system call interface regardless. Furthermore, since binary programs initiate reading input through system calls, modeling inputs only through system calls is a natural cut. Finally, a model based on
CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

system calls yields test cases that are system call traces, making the executor appear from a testing view as an especially violent operating system.

The system model is a small symbolically executed runtime library which describes system call side effects. When the target program issues a system call, the executor vectors control to the model library, which marks the side effects with symbolic data. The model uses interpreter intrinsics to update state properties (e.g., new mappings for `mmap`) and to impose additional constraints on symbolic side effects. Variable assignments for this symbolic data form test cases to reproduce program paths.

The system model in klee-mc can emulate Linux and Windows. The Linux model supports both 64-bit programs (x86-64) and 32-bit programs (x86, ARM) in the same code by carefully assigning structure sizes based on the architecture bit-width. The x86 Windows model is less developed than the Linux model, primarily due to system call complexity, but is interesting from a portability standpoint (the problem of correctly modeling an operating system is investigated in Section 3.3.2). For most system calls, the guest program passes in some buffer, the model marks it symbolic, then control returns to the program. For calls that do not write to a buffer, but return some value, the model marks the return value as symbolic.

Both models attempt to overapproximate known system calls. When marking a buffer of memory passed to through a system call as symbolic, the model will occasionally permit values which disagree with expected operating system values. For instance, reads to the same position in a symbolic file will return different symbolic data, flags can be set to invalid combinations, and system times may go backwards. The system will not overapproximate when it can lead to an obvious buffer overrun, such as giving an element count that exceeds a buffer or returning a string without a nul terminator.

**Function Hooks**

Instrumenting calls to functions with runtime libraries gives the executor new drop-in program analysis features. klee-mc’s function hook support lets the user define a list of bitcode libraries to load on start up that will intercept calls to arbitrary functions within a state’s symbolic context. In practice, Section 6.4.2 extensively uses function
void __hookpre__GI__assert_fail(void* regs) {
    klee_uerror((const char*)GET_ARG0(regs), "uassert.err");
}

void __hookpre__GI_exit(void* regs) {
    exit(GET_ARG0(regs));
}

Figure 2.5: Function entry hooks for process exit functions

hooks to instrument libc memory heap functions such as malloc and free.

The function hook facility dynamically loads bitcode libraries, as given by a command line argument, which define function code to be called before entering and exiting specific functions. The loader uses the library’s function names to determine which functions to instrument and where to put the instrumentation; functions named __hookpre__f are called whenever a state enters f and functions named __hookpost__f are called whenever a state leaves f. When the function f is loaded into KLEE-MC, the calls to all relevant hook functions are inserted into f’s code to ensure the hooks are called on entry and exit.

As an example, Figure 2.5 shows two example function hooks. The functions intercept every entry to glibc’s internal __GI_assert_fail (called on an assertion failure) and __GI_exit (called on normal termination) functions. The library functions indicate the program intends to exit following some cleanup code. Instead of running this cleanup code, the function hooks immediately terminate the calling state with the failed assert producing an error report and the exit producing a normal test, saving the executor unnecessary computation.

2.4.6 Limitations

As with any research system, KLEE-MC has some limitations. These limitations reflect intentional omissions to simplify the system in the interest of expediency and tractability rather than serious architectural deficiencies. Although the lack of certain features inhibits the analysis of select programs, there are no strict reasons prohibiting the support in the future.

First, KLEE-MC lacks support for signals and threads. Bugs caused by signals and threads tend to be difficult to reproduce (and hence unconvincing without significant
CHAPTER 2. THE KLEE-MC BINARY PROGRAM SYMBOLIC EXECUTOR

The symbolic execution overhead can be expensive, and there are plenty of deterministic bugs already. Currently, the system will register signal handlers, but never trigger any signals. Similarly, the system immediately fails any thread-related system calls. However, there is partial support for threads based on selecting a single thread context from a snapshot to symbolically execute.

The system model does not perfectly simulate its target platform. In most cases, the system model can ignore specific environment details (e.g., timers, system information, users and groups, capabilities) by overapproximating with unconstrained symbolic data. However, system calls such as `ioctl` rely on program and device-specific context to define the precise interface; the system model does not know which arguments are buffers that can be marked symbolic. Ultimately, this leads to an underapproximation of the operating system which can reduce overall program coverage.

Programs with large memory footprints, such as browsers, office suites, and media editors, severely strain KLEE-MC’s memory system. Large programs do run under KLEE-MC but do not run very well. When a program uses over a hundred thousand pages, the address space structure rapidly becomes inefficient since it tracks each page as a separate object. Large programs often also have large working sets, making forked states, even when copy-on-write, cause considerable memory pressure. Conceivably, better data structures and state write-out to disk would resolve this issue.

Supervisor resources, those that require kernel-level privileges, and devices, are not supported. While these features necessary to run operating systems, hypervisors, and other software that must run close to the hardware, which are interesting in their own right, KLEE-MC focuses only on user-level programs. The symbolic executor could certainly model supervisor resources and devices (and some do), but the limited amount of software that uses these features, the difficulty of modeling obscure hardware quirks, and analysis necessary to efficiently support device interrupts, makes such a platform too specialized to pursue in the short term.
2.5 Experiments

This section demonstrates the characteristics of a single run of klee-mc’s binary symbolic execution over thousands of programs taken from three architectures (ARM, x86, x86-64), representing experimental data taken over approximately the course of a year. Linux binaries were collected from Fedora, Ubuntu, RaspberryPI, and CyanogenMod distributions. Windows binaries were collected from several malware aggregation sites. We highlight noticeable properties concerning key aspects of the system when working with bulk program sets, including snapshotting, testing, and coverage.

2.5.1 Snapshots

Snapshots let the executor easily load a program by using the program’s host platform to set up the entire process image. To take a snapshot, it must be possible to run the program; this is not always the case. Table 2.3 lists the success rates for snapshotting programs by each named collection or distribution and shows simply launching a program can be challenging.

Up to 17% of programs for each collection failed to snapshot. For x86 and x86-64 Linux, many binaries had dependencies that were difficult to resolve. Some binaries would fail when linked against libraries with the correct versions but wrong linkage flags; we set up several LD_PATHs and cycled through them in case one would launch
the binary. For ARM Linux, there were several runtime linkers, `linux-rtld`, with the same name but slightly different functionality; using the wrong linker would crash the program. Normally, only one linker and its associated set of binaries would be installed to the system at a time. To support multiple linkers on the same ARM system at once, each linker was assigned a unique path and each binary’s linker string was rewritten, cycling through linkers until the binary would launch. For Windows, programs were launched in a virtual machine, but the binaries were often compressed with runtime packers, making it difficult to extract and run the actual program.

Although snapshots are larger than regular binaries, the design can exploit shared data, making them space efficient. Figure 2.6 shows storage overhead for the system’s snapshots in logarithmic scale. Every snapshot memory region is named by hash and saved to a centralized area; regions for individual snapshots point to this centralized store with symlinks. The figure shows physical storage usage, which is roughly half unshared and shared data. The virtual data is the amount of storage that would be used if all shared data were duplicated for every snapshot. In total, deduplicating shared data gives a $4.9 \times$ reduction in overhead, amounting in 696 gigabytes of savings.

Furthermore, this shared structure helps reduce overhead on single binaries when snapshot sequencing for system model differencing (§3.3.2).
At its core, klee-mc produces tests for programs. To find tests, programs were symbolically executed for five minutes a piece. All programs were tested beginning at their entry point with symbolic command line arguments and totally symbolic files.

Figure 2.7 shows the mass checking test case results for all programs from Table 2.3. In total, the executor partially explored 24 million program paths and produced 1.8 million test cases. Only a fraction of all paths become test cases; it is still unclear how to generally select the best paths to explore. We distinguish between complete paths, a test that runs to full completion, and concrete tests, a test which may have concretized its symbolic state early; 14% of test cases were concretized, demonstrating the usefulness of state concretization and the importance graceful solver failure recovery.

Of course, finding bugs is a primary motivation to testing programs. Figure 2.8 shows the total errors found for each tested platform. By generating over a million test cases for the programs, we were able to find over ten thousand memory access faults (pointer errors) and over a thousand other faults (divide by zero and jumps to invalid code). Divide by zero errors were the rarest faults, possibly on account of divisions
being less frequent than memory accesses and indirect jumps. Likewise, hundreds of tests ran into unsupported system calls (syscall errors), demonstrating the system’s ability to detect some of its own corner cases. We also include the Window’s results to show the effects of a prototype model; the vast number of pointer errors indicates the heightened sensitivity Windows programs have toward unrealistic system call results.

The tests from KLEE-MC which trigger bugs certainly appear machine-generated. Some of the more human-readable bugs are shown in Table 2.4. These bugs are interesting symbolically derived command line inputs which cause a given program to crash, mostly due to string handling bugs. Of particular note, dc allocates a string with −1 characters. Surprisingly, many programs seem to crash with no arguments, either because they always expect arguments or because they always assume some (missing) file is present. Table 2.5 lists a few (less readable) bugs detected through symbolically derived input files. The file data is given as a hex dump taken from od program; the hexadecimal address on the left represents the file offset, the byte values follow, and * indicates the last line’s data repeats until the next offset. These bugs tend to be deeper than command line bugs since files are subject to more processing than command line strings: strings intelligently detects but crashes analyzing a malformed “srec” file, mp4info accesses an out-of-bound metadata property, ocamlrun
reads missing section descriptors, and fsck.ext2 attempts to fill out a buffer using
a negative length.

### 2.5.3 Coverage

Test counts tells little about how much distinct program behavior is exercised by the
executor. For instance, a string comparison of length \( n \) will generate \( n \) test cases but
each test case exercises the same code paths. The total amount of unique machine
code covered and the types of system calls exercised, however, give a crude measure
of code and path variety across all programs.

Table 2.6 lists basic blocks and absolute code covered. The data is divided into
a unique classification, counting the same string of code for a basic block only once
(measuring diversity), and a total classification, counting the same string of code for
every program where it was covered (measuring total coverage). Although there’s
a high degree of code sharing among programs, the KLEE-MC system still processes
hundreds of megabytes of unique code on account of easily handling large program sets
across several platforms. Of these platforms, one interesting aspect is the particularly
high rate of code sharing for Windows; this is likely due to programs spending most
of their time in the user interface libraries.

The system model can reduce code coverage if it does not accurately model its
Table 2.5: Selected file inputs triggering memory access faults

<table>
<thead>
<tr>
<th>Program</th>
<th>File Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>strings-2.24</td>
<td>0000 53 32 30 30</td>
</tr>
<tr>
<td></td>
<td>0004 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>mp4info-2.0.0</td>
<td>0000 01 00 00 82</td>
</tr>
<tr>
<td></td>
<td>0004 63 74 74 73</td>
</tr>
<tr>
<td></td>
<td>0008 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>000c 40 00 00 00</td>
</tr>
<tr>
<td></td>
<td>0010 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>1fff 00 00 00</td>
</tr>
<tr>
<td>fsck.ext2-1.42.10</td>
<td></td>
</tr>
<tr>
<td>ocamrun-4.01.0</td>
<td>0000 20 00 00 00</td>
</tr>
<tr>
<td></td>
<td>0004 43 61 6d 6c</td>
</tr>
<tr>
<td></td>
<td>0008 31 39 39 39</td>
</tr>
<tr>
<td></td>
<td>000c 58 30 30 38</td>
</tr>
</tbody>
</table>

Table 2.6: Amount of machine code covered by klee-mc

<table>
<thead>
<tr>
<th>Platform</th>
<th>Basic Blocks</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unique</td>
<td>Total</td>
</tr>
<tr>
<td>x86-64 Linux</td>
<td>7,498,415</td>
<td>82,724,878</td>
</tr>
<tr>
<td>x86 Linux</td>
<td>2,304,809</td>
<td>24,016,450</td>
</tr>
<tr>
<td>ARM Linux</td>
<td>450,340</td>
<td>5,679,779</td>
</tr>
<tr>
<td>x86 Windows</td>
<td>89,494</td>
<td>2,326,051</td>
</tr>
</tbody>
</table>
target platform. However, if a system call is used by only a few programs, the quality of its implementation has little impact on overall coverage. Figure 2.9 shows system calls sorted by total number of programs where they were used. The linear drop-off of calls on the logarithmic scale indicates system call usage follows an exponential distribution, suggesting focusing effort on more popular system calls would be a good choice. The most common calls for Windows were \texttt{NtUserCallOneParam}, a unified and complex interface for dozens of User32 system calls (modeled poorly), and \texttt{NtUserWaitMessage}, a call that suspends a thread until it receives a message (modeled easily). The most common calls for Linux were \texttt{exit\_group}, \texttt{write}, and \texttt{mmap}, which are all modeled accurately. For the long tail, 20 (ARM, Windows), 28 (x86-64), and 48 (x86) calls were only used by a single program.

2.6 Conclusions

This chapter provided a context for symbolic execution and described the design of \texttt{klee-mc}. The details to the changes necessary to support machine code programs served as a convenient tour of core system components. Although \texttt{klee-mc} can analyze huge program sets, finding thousands of unique program faults, the additional
system complexity suggests robustness and correctness are paramount when analyzing
machine code, which will remain a focus throughout the rest of this dissertation.
Chapter 3

Machine Cross-Checking

3.1 Introduction

The ease of applying dynamic analysis to unmodified program binaries spurred the development of binary symbolic execution. A binary symbolic executor symbolically evaluates compiled executables as-is; there is no need for recompilation, annotations, or special linking. This contrasts with systems [28, 58, 75, 119] built to process source code or a metadata-rich byte code. A typical system pairs a symbolic interpreter with a dynamic binary translation (DBT) front-end [30, 34, 59, 92, 96, 124]; the DBT converts a program’s machine code into instructions for the symbolic interpreter.

However, this convenience comes at the cost of complexity. The executor must support broad classes of machine code programs to be generally useful. While the system should closely match the target program as it behaves on hardware, mere inspection is inadequate to confirm accurate execution; soundness demands near-flawless interpretation. Few decisions are shallow: almost every result depends on a long line of prior computations. Mistakes cause both false positives (which at least can be seen and fixed) and, worse, invisible accumulation of false negatives as it misses paths, corrupts constraints, and more. It is often difficult or impractical to confirm these error reports by hand; instead, test cases serve as certificates against false positives.

We built an ensemble of techniques to achieve bitwise equivalence to program
evaluation on hardware with a dynamic binary symbolic executor. These techniques include co-simulation against hardware, equivalence checking, side-effect analysis, and hardware fall-backs. The process often compares unreliable data sources (cross-checking) but correctness arises by working transitively down to physical hardware.

These automated checks proved invaluable when developing klee-mc. Imprecise approximations of the program environment are detected by comparing symbolic execution traces against operating system side-effects. By checking test cases from symbolic interpreter against an independent JIT interpreter, bugs in the stack which impact execution can be detected near the point of failure in the guest program. The JIT is further checked for execution discrepancies through bi-simulation with hardware, ensuring the JIT’s intermediate results when transitioning from basic block to basic block match hardware. Expression rewrite rules are verified offline for correctness. End-to-end bugs in the expression builder in symbolic expression rewriting rules are detected at the instruction level. Detailed diagnostic reports are given to the user whenever execution accuracy becomes suspect.

The rest of this chapter is organized as follows. Section 3.2 discusses the motivation for high-integrity binary symbolic execution. Section 3.3 describes the design the klee-mc integrity checking features. Section 3.4 outlines specific implementation details behind the symbolic executor. Section 3.5 shows the results of checking the symbolic execution system across the stack. Section 3.6 discusses related work. Finally, Section 3.7 concludes.

### 3.2 Motivation

Binary symbolic execution closely matches the literal interpretation of a target program as on hardware, an advantage over source-based methods, but at the expense of clarity. Since paths become lengthy and obscured by machine semantics, the executor must accurately symbolically evaluate the program and produce sound test cases— an arduous task in light of the system’s complexity. Hence we reduce the problem to automatically checking the symbolic executor piece by piece.


CHAPTER 3. MACHINE CROSS-CHECKING

---

**Error Report**

```
Error: bad memory access!
{
  "address" : "
(ReadLSB w64
 (Add w32 320 N0:(Extract w32 0 (Shl w64
 (SExt w64 (Read w8 4621 read1)) 3)))
const_os7)
,
Stack: #1 in __IO_vfprintf_internal+0x447d
  #2 in __printf_chk+0xc8
  #3 in 0x40160b
  #4 in 0x40275d
  #5 in __libc_start_main+0xac
  #6 in 0x400e70
```

---

**Test Replay**

```
[~/]$ kmc-replay 50
Replay: test #50
[...
[kmc-replay] Applying: sys=write
[kmc-replay] Couldn’t read sclog entry #13.
[~/]$ 
```

Figure 3.1: A user’s limited perspective of a bad path.

### 3.2.1 Bad Paths

Whenever a program bug detection tool discovers an error, it is commenting on an aspect of the program that will likely be unfamiliar to the user. If the error is spurious, the issue is further compounded; tool code fails somewhere obscure when analyzing an unusual program. Given a binary program lacking source code, this intermediate computation becomes essentially opaque to human inspection.

Figure 3.1 presents one such baffling situation where an error report has an irreproducible path, illustrating the difficulty of understanding symbolic execution false positives. The symbolic executor tool reports a memory access error somewhere inside `printf` on a symbolic address derived from a `read` system call. When the user tries to review the test case (test 50) with the `kmc-replay` test replay utility, the test runs out of system calls, landing on a `write`, instead of crashing. Presumably an illegal memory access should have occurred after the last retired system call (`lseek`), before
the `write`, and within `printf`. It is unclear what exactly went wrong or why, only that the symbolic executor (or replay facility) is mistaken. Obviously, there must be some software defect, whether in the target program or the executor; the challenge is to determine the precise source of the problem.

### 3.2.2 System Complexity

Binary symbolic executors build from the principles of classical source-based symbolic executors. Figure 3.2 highlights the important components of a binary symbolic execution stack. These components must work perfectly in tandem to produce correct test cases; a binary symbolic executor can fail at numerous points throughout the stack. The goal of our work is to make these issues tractable. Other projects have addressed some reliability concerns, but none have combined them under a single system. In Section 3.6, we will compare KLEE-MC with this related work.

As given in Chapter 2, the basic process for symbolic execution is as follows. First, the executor begins by loading a target program. A front-end processes the target program into a simpler intermediate representation. An interpreter symbolically evaluates the intermediate representation by manipulating a mix of symbolic expressions and concrete data. A state acquires symbolic data through simulated program environment features defined by a custom system model. Evaluation forks new states on feasibly contingent branches decided by a constraint solver; each state accrues path constraints according to its sequence of branch decisions. When a state terminates, the executor constructs a test case a variable assignment that satisfies the state’s path constraints. Finally, replaying the test case reproduces the path.

The executor loads a target program similar to a native program loader. If loader details diverge from the target system, code paths depending on runtime detected architectural features will differ (e.g., string functions, bulk memory operations, codecs). Unlike a traditional program loader, the executor may support seeded tests that begin running long after the program entry point, necessitating point-in-time snapshots.

Every binary symbolic executor must handle machine code in its front-end but machine code decoders are known to be unreliable [60, 92, 104]. Modern instruction
sets are huge, complicated, and new instructions are continually being added; hardware outpaces decoder software. Additionally, the interface between the decoder and the symbolic executor’s intermediate representation adds another layer of unreliable abstraction.

The underlying symbolic interpreter evaluates the target program’s instructions. However, the interpreter can easily diverge from intended execution. It can mis-apply instructions (e.g., vector instructions). It can apply bogus optimizations on constraints. It can apply bogus optimizations on state data. It can fail to handle difficult symbolic cases (e.g., memory dereference, floating-point).

A system environment model inserts symbolic data into the program state. A model may reimplement libraries [28, 86, 116], simulate system calls [30], or emulate hardware devices [45]. Regardless of abstraction, if the model misrepresents the environment then the following path diverges from the set of feasible platform paths.

Like a traditional symbolic executor, KLEE-MC relies on specialized expression and constraint solver optimizations for good performance (§ 2.4.3). Expression optimizations, such as strength-reductions and structural simplifications, are often hand-written. Constraint solving is accelerated by light-weight query processing. The executor may also rewrite expressions to be palatable to a particular solver implementation. Keeping data symbolic throughout execution means the consequences of a broken optimization may not manifest until long after its application.
Replaying a test case should reproduce its respective program path. For binary programs, the replay mechanism can rely on the interpreter, a just-in-time compiler, or hardware. Since test cases are generated by the interpreter, replaying through other executors can expose inconsistencies. Furthermore, if the interpreter is non-deterministic, the test case can fail to replay at all.

Fortunately, the constraint solver, target environment, and target hardware all strengthen the system’s faithfulness to native execution. The constraint solver can check the solver and expressions stacks. The target environment can test the system model. The hardware can test the binary front-end and final test cases. Checking the executor with these mechanisms improves the overall integrity of the system. Furthermore, these components often help improve the robustness of the system such that it can proceed in light of failures.

### 3.3 Cross-Checking in klee-mc

In this section we discuss the design and general strategy for improving the integrity of the klee-mc binary symbolic executor.

#### 3.3.1 Deterministic Executor Data

The executor’s integrity mechanisms primarily depend on two sources of data. First, the executor needs realistic and deterministically reproducible guest states, given by program snapshots (§ 2.4.1) and extended to snapshot sequences for system call side-effect cross-checking. Second, the executor deterministically reproduces various intermediate path results for bi-simulation using register and system call logs.

**Program Process Snapshots**

As described in Section 2.4.1, a snapshot is the state of a process image taken from a live system. The system snapshots a binary program by launching it as a process and recording the resources with system debugging facilities (e.g., ptrace). The symbolic executor first loads a target program by its snapshot before symbolically executing
its code. Snapshots eliminate side effects and non-determinism from new libraries, different linkers, and address space randomization over multiple runs by persistently storing an immutable copy of process resources.

**Snapshot Structure.** A snapshotting program (the snapshotter) saves a running process’s image data to a directory. Snapshot data is structured as a simple directory tree of files representing resources loaded from the process. Three resources constitute the core process image’s machine configuration:

- **User Registers** – the set of registers for each thread is stored in a thread directory. These registers include the stack pointer, program counter, floating-point registers, and general purpose registers.

- **System Registers** – registers not directly accessible or modifiable by the program but necessary for correct program execution (e.g., segment registers and descriptor tables).

- **Memory** – all code and data in the process including libraries, heaps, and stacks.

**Snapshot Sequences.** Detecting system model differences relies on comparing program traces from the host and the executor. We represent host program traces at a system call granularity by recording sequences of snapshots. Snapshot sequences are stored as *snapshot pairs* pivoted around system calls. A snapshot pair has a *pre*-snapshot, taken immediately prior to a system call, and a *post*-snapshot, taken immediately after a system call. We write snapshot pairs as tuples \((s, s^*)\) where \(s\) is the pre-snapshot and \(s^*\) is the post-snapshot. The machine configuration difference from \(s\) to \(s^*\) is a system call side effect.

Although a snapshot may be taken at any point, only the moment immediately preceding and following a system call give fundamentally distinct snapshots. The executor only introduces symbolic data when dispatching system calls since all input derives from system calls. This controlled introduction of symbolic data means a snapshot \(s'\) between system calls is equivalent to a snapshot \(s\) immediately following the last system call because \(s\) will always eventually evaluate to \(s'\) due to snapshots being totally concrete.
Symbolic execution of a snapshot pair begins in the system call model. The pre-empted system call from the pre-snapshot is processed by the symbolic system model, symbolic data is created if necessary, the system call completes and the target program code is symbolically executed. Assuming the symbolic system model subsumes the operating system’s behavior, the post-snapshot is a concretized path of the pre-snapshot system call through the model. A post-snapshot begins with no symbolic data so symbolic execution should match native execution up to the next system call.

Test Case Logs

Recall from Section 2.3 that for every test case, the symbolic executor generates files of concrete data suitable for reconstructing a program path. A concrete test case is a variable assignment which satisfies the path constraints, computed by the solver. To replay a test, the interpreter replaces all symbolic data with concrete test case data.

Native execution makes system model effects difficult to reproduce when replaying the test data through the model code. The model cannot run directly in the target program without additional support code; the model would have to intercept system calls, use its own heap (e.g., use its own malloc), and use special assembly trampoline to reflect register updates. If the model runs outside the target program’s address space, the system must monitor the model’s memory accesses for target addresses. Regardless of where the model runs, there still must be controls for non-determinism, such as precisely reproducing addresses for memory allocation must control.

To avoid any dependency on the model code during replay, the executor logs modeled system call side effects in a test case log. In addition to maintaining model side effect information, the test case log is a general purpose logging facility for traces from the executor. This general purpose facility is useful for tracking state information when debugging the system, such as monitoring memory addresses and stack modifications. For automated interpreter integrity checking, which is this chapter’s focus, the log records registers at basic block boundaries for cross-checking against JIT-computed registers in Section 3.3.3.

Figure 3.3 illustrates the logging process during symbolic execution. Whenever the executor dispatches a machine code basic block, the executor writes the register
 CHAPTER 3. MACHINE CROSS-CHECKING

Figure 3.3: Building a test log during symbolic execution

set to the state’s log. Whenever the system model dispatches a system call, the executor writes the side-effects to the state’s log. In the example, the system call returns a symbolic value, so the return value register `%rax` has its concrete mask set to 0 to indicate the register is symbolic. The executor chooses to follow the `OK` branch, implying `%rax` must be the concrete value 0; the `%rax` mask value reflects the register is concrete. When the state terminates, the executor writes the log to the filesystem along with the concrete test case variable assignments.

System Call Logging. The model code produces a concrete log of information necessary to replay the system call side effects independent of the system model for every test case. The log records memory stores and register information similar to older Unix systems [90], as well as metadata about the system call itself. On test replay, the program runs on a DBT based on the LLVM JIT and the log is replayed to recreate the system call’s effects.

Figure 3.4 illustrates the data associated with system call log entry. On the left, the figure shows a system call record. It begins with a record header, common to
all log records, which describes the record type, flags that control attributes specific to that type, and a record length, so unrecognized record types can be skipped.

The system call record type itself holds the system call number, useful for checking whether replay has gone off course, a translated system call number, for emulation by the host platform, the call’s return value, provided it’s not symbolic, and the number of memory updates to expect. These memory updates, following the system call record, describe a base pointer or system call argument number that should be updated, along with the length of the update. The update data maps to the concrete test case’s list of array assignments. If the update length disagrees with the array length, the test replay reports a system call replay error.

**Register Logging.** The symbolic executor optionally logs the state’s machine registers (e.g., `rax`, `xmm0`) throughout execution. The logged registers record intermediate results for fine-grained integrity checking during test replay to catch interpreter errors close to the point of failure. For every dispatched basic block, the interpreter appends the emulated register file and a concrete mask, which indicates whether a register is symbolic or concrete, to the log. Test replay only checks concrete data against concrete registers; symbolic bytes are ignored.
3.3.2 Operating System Differencing on the System Model

When a program state enters the system model by making a system call, the set of states that may exit the model should precisely represent every result the operating system could possibly return. Otherwise, the system model diverges from the modeled operating system, either introducing side effects never observed in practice, producing unrealistic tests, or missing side effects, potentially missing large swaths of program code. We compare the results of symbolic execution of pre-snapshots with the side effects reflected in post-snapshots to judge the system model’s accuracy.

Model Fidelity

A system model fails to model the operating system’s side effects in possible two ways. One, the model is overconstrained when missing side effects. Two, the model is underconstrained when introducing new side effects. These failure modes are orthogonal: a model can both introduce new side effects and miss legitimate side effects.

Model quality in this sense is easily formalized. Let \( S(s) \) be the set of all possible configurations that a state \( s \) may take immediately following a system call to the operating system. Let \( M(s) \) be the set of all possible configurations that a state \( s \) may take immediately following a system call handled by the system model \( M \). The model \( M \) is overconstrained when \( \exists x \in S(s) \) such that \( x \not\in M(s) \). The model is underconstrained the operating system when \( \exists x \in M(s) \) such that \( x \not\in S(s) \). For a snapshot pair \((s, s^*)\), by definition the post-snapshot is a configuration from applying the operating system, \( s^* \in S(s) \).

System Model × Operating System

Every snapshot pair describes side effects of a system call on a process image. To compare the model with the operating system specification, the pre-snapshot seeds symbolic execution and the post-snapshot gives expected output. Differences between the symbolically derived test cases and post-snapshot determine the model’s accuracy.

Figure 3.5 diagrams the process for differencing operating system and model side effects. First, the pre-snapshot and post-snapshot are differenced to locate system
call side effects. Next, the pre-snapshot is symbolically executed for one system call. Finally, the side effects from symbolic execution are compared with the side effects derived from the snapshot pair.

We consider side effects related to register and memory contents. Side effects are found by differencing snapshot pairs (the pair difference). Pair differencing relies on the lightweight snapshot structure. Unchanged memory segments are symlinks and can be skipped. Updated memory segments are byte differenced and modified addresses are stored in a side effect summary of address ranges $\mathcal{A}$.

Symbolically executing the pre-snapshot produces the model side-effects. The pre-snapshot is loaded into the symbolic executor with the model configured to exit immediately after completing a system call. Symbolic execution produces a set of test cases $\mathcal{T}$. Each test case $t \in \mathcal{T}$ includes path constraints $C(t)$ (as a set of boolean bitvector expressions) and a set of updated memory ranges $A(t)$.

The model can be checked for underconstraining and overconstraining through side effects by $\mathcal{A}$ and $\mathcal{T}$. For overconstraining, if for every $t \in \mathcal{T}$ the operating system update set contains addresses not in the model update set, $\bigcup(\mathcal{A}) \cap \bigcup(A(t)) \neq \bigcup(\mathcal{A})$, then $t$ with the minimal missing locations $(\mathcal{A}) - \bigcup(A(t))$ represents the model overconstraining. For underconstraining, if every $t \in \mathcal{T}$ has a update address in the snapshot’s memory space but not the operating system side effect set, $(\bigcup(A(t)) - \bigcup(\mathcal{A})) \cap s^* \neq \emptyset$, then that address represents an underconstrained side effect.

Unsupported System Calls

For practical reasons, a subset of system calls are left partially or totally unsupported in the model. First, modeling system calls that are rarely used, perhaps by a handful of programs, or never observed, has little payoff. Second, there is an implementation gap as new system calls and flags as they are added host operating system. Therefore the system model gracefully handles unsupported system calls. Whenever a path reaches an unsupported system call number or feature flag, the system model generates a missing system call report and the path returns from the model with an unsupported return code.
3.3.3 Execution Testing with Deterministic Test Replay

Test cases made by the symbolic executor serve dual purposes. On the surface, these tests exercise a target program’s paths. However, treating this generated data as input for symbolic execution machinery means the executor makes tests for itself. This testing works by replaying program tests and comparing against prior executor state by alternating means of execution.

The system has three separate ways to evaluate code to test itself. First, non-determinism in the interpreter is ruled out by replaying test cases in the interpreter. Next, semantic differences between the intermediate representation and the interpretation are found by replaying on the JIT executor. Finally, direct hardware replay detects errors in the machine-code translation. If path passes each level, the interpreter transitivity matches hardware.
CHAPTER 3. MACHINE CROSS-CHECKING

**Interpreter × Interpreter**

Replaying program paths in the interpreter rules out non-determinism bugs. The replay process uses the test case as a variable assignment log. Whenever the system model creates a symbolic array, the interpreter applies the variable assignment from the test case and advances its position in the test case. Non-determinism in the interpreter can cause the current variable assignment to have a different name from the symbolic array or a different size, causing the replay to fail. Although this does not ensure the interpreter follows the path from the test case, it is a good first pass; differing paths often have separate system call sequences. Regardless, non-deterministic paths with the same system call sequences are detected through register logs in the next section.

**Interpreter × JIT**

The interpreter and LLVM JIT should have equivalent concrete evaluation semantics. Since the interpreter and JIT are independent LLVM evaluation methods, the two can be cross-checked for errors. We cross-check the two by comparing the interpreter’s register log with the JIT’s register values. The heuristic is that an interpretation error, whether on symbolic or concrete data, eventually manifests as a concrete error in the register file. On JIT replay, logged registers marked as concrete are compared with the JIT registers; a mismatch implies the interpreter or JIT is incorrect and is reported to the user.

**JIT × Hardware**

Comparing the JIT and hardware finds translation bugs. Starting from the entry point, the guest and native process run in tandem. The replay process executes code one basic block at a time: first through the JIT, then through the native process; the native process is never ahead of the JIT. Once the basic block is retired by the native process, the JIT state and native state are cross-checked for equivalence. For speed, only registers are compared at each basic block. On a mismatch, both registers and memory are differenced and reported to the user for debugging. Occasionally
mismatches are expected due to known translation issues (§ 3.4.2); these cases are ignored by exchanging state between the JIT and hardware.

Hardware is traced by first creating a native process from a program snapshot. The snapshot is loaded as a native process by forking KLEE-MC and jumping to the snapshot’s entry point address. The entry point address is breakpointed so process control with ptrace starts at the beginning of the program. KLEE-MC can jump to the snapshot’s code because it identity maps memory from snapshots; collisions are rare because the process is mapped to an uncommon address and non-fixed memory is dispersed by address space layout randomization. Although the native process has KLEE-MC in its address space as well as the snapshot, the memory from KLEE-MC is never accessed again.

3.3.4 Host CPU and the Machine Code Front-End

Cross-checking third party binaries only finds decoder bugs covered by existing code. While binaries taken from various environments are one step toward robustness, they fall short of the darker corners of the decoder — broken opcodes, sequences, and encodings either rarely or never emitted by compilers. To uncover these aspects of the decoder, we symbolically generate code fragments then cross-check their computation
between hardware and the JIT.

We generate new programs symbolically as follows. We mark an instruction buffer as symbolic, feed it to a VEX front-end (denoted guest-VEX) which is interpreted inside klee-mc. On each path klee-mc explores, the code in guest-VEX transforms symbolic bytes into constraints that exactly describe all instructions accepted or rejected by the path. We call these constraints fragments since they solve to a fragment of machine code. Fragments follow the validation pipeline in Figure 3.6.

Fragments are generated as follows. A small program, symvex, calls the VEX decoder with a symbolic buffer. symvex is a standard native binary — it is compiled with the stock system compiler (gcc in our case) and runs under klee-mc like any other program. To start fragment generation symvex reads from a symbolic file, marking a buffer symbolic. symvex then calls the VEX instruction decoder on the buffer. As the VEX decoder runs (as a guest, under klee-mc ) it mechanically decodes the buffer into every instruction sequence VEX recognizes.

The length and contents of the buffer were guided by the nature of the x86-64 instruction set. It is 64 bytes long with the first 48 bytes marked as symbolic. Since the maximum length of an x86-64 instruction is 16 bytes, the buffer fills with a minimum of three symbolically decoded instructions. To keep the decoded instructions from exceeding buffer capacity, the final 16 bytes are fixed as single-byte trap instructions; falling through to the tail causes a trap.

We use a small harness program (xchkasm) to natively run the putative code produced by solving a fragment’s constraints. An optional register file may be provided by the user to seed the computation (by default, a register file filled 0xfe is used). To protect itself from errant code, the harness establishes a sandbox by forking a ptraced process. A small assembly trampoline bootstraps native execution. It loads the register state from memory into machine registers and jumps to the code fragment. Since few fragments contain jumps, fall-through code is caught by trap opcode padding. If a fragment makes a system call, it will be trapped by ptrace. Unbounded execution caused by stray jumps is rarely observed but can be caught with a watchdog timer if the need ever arises.

Concrete register files are insufficient for testing fragments which contain jumps
Figure 3.7: Symbolic register file derived constraints to trigger a conditional jump.

or other value-dependent execution. For example, computing condition flags relies on an out-call to special, complicated, VEX library bitcode to find the flags on demand. Figure 3.7 illustrates the effect of conditional coverage. To address this issue, we also run fragments through the symbolic executor with a symbolic register file to explore the paths through these helper functions, finding solutions for register files which satisfy the conditions.

3.3.5 Constraint Validity for Expressions and Solvers

The values that symbolic data can take on a given path are represented using symbolic expressions, which describe the effects of all operations applied to the data. The constraint solver ultimately consumes these expressions as queries, typically when resolving branch feasibility and to find concrete assignments for test cases. KLEE-MC serializes queries into the SMTLIB [12] language and so expressions correspond to a subset of SMTLIB.
Solver × Solver

An efficient solver is a stack of query processing components terminated by a full theorem prover. These components include query filters, incomplete solvers, and caches. Since the stacks are non-trivial, the KLEE-MC system uses a debugging technique from KLEE to cross-check solvers during symbolic execution.

Solvers are checked with a dual solver at the top of the solver stack. The dual solver passes every query to two separate stacks, then checks the results for equality. If the results do not match, then one solver stack must be wrong.

Running two separate solvers is expensive. For instance, cross-checking a cache with a bare theorem prover would recompute every query that the caching would otherwise absorb. Additionally, solver bugs always reappear during path reconstruction, so checking for solver bugs can be deferred to path replay. In practice, the solver cross-checker detects unsound reasoning arising when developing solver components and optimizations.

KLEE-MC supports arbitrary solvers by piping SMTLIB queries to independent solver processes. This means KLEE-MC avoids specific library bindings and therefore submits text-equivalent queries to distinct SMT solvers. This is important because subtle differences in bindings and solver-specific optimizations can introduce bugs for one solver but not another. Although piping SMTLIB queries incurs an IPC penalty, using a solver as a library can ultimately impact stability of the system (e.g., STP was notorious for stack overflows), so there is impetus to use independent processes.

Expression × Expression × Solver

KLEE-MC builds expressions using an expression builder. Given the expense of interacting with the constraint solver, a significant portion of this builder code focuses on translating expressions to more efficient (typically smaller) but semantically equivalent ones, similar to peephole optimization and strength-reduction in a compiler backend. The ideal replacement produces a constant.

Expression optimizations are checked for errors at the builder level with a cross-checked expression builder. Figure 3.8 illustrates the process of cross-checking two
expression builder stacks. The cross-checked expression builder creates the desired expression (subtracting $x \& 0xff$ from $x$) once using a default, simple builder ($x - x \& 0xff$), then once again using the optimized builder ($x \& \neg 0xff$). Both expressions are wrapped in an equality query and sent to the constraint solver to verify equivalence through logical validity. If the two expressions are found to be equivalent, the result of the optimized builder is safe to return. Otherwise, the system reports an expression error and the builder returns the default expression.

For efficiency, we only cross-check “top-level” expressions, i.e., expressions not created for the purpose of another expression. In addition, before invoking the constraint solver, a syntactic check is performed to see if both expressions are identical. If so, the solver call is skipped, reducing overhead by an order of magnitude or more.

Special care must be taken to avoid recursion in the solver but maintain soundness. Recursion arises from the solver building expressions, such as by applying strength reductions or testing counterexamples. To avoid an infinite loop from the solver calling back into itself when building an expression, the cross-checked expression builder defers in-solver validations to the first expression created outside the solver.

**Expression Rules × Solver**

In addition to hard-coded expression builders, the system uses soft expression reduction rules derived from program traces [113]. These rules (e.g., $a \rightarrow b$) describe templates for translating larger expression structures ($a$) to equivalent shorter expressions ($b$). Soft rules have the advantage that the expression templates are instantiable into actual expressions. Hence, a rule $a \rightarrow b$ is verifiable with a constraint solver by testing the validity of ($a \equiv b$). Furthermore, the template expressions provide realistic seed expressions which are useful for expression fuzzing.
3.4 Implementation

This section describes implementation details necessary for cross-checking with bit-equivalent under a binary symbolic executor. Left untended, bit-equivalence breaks under non-determinism. The system handles this non-determinism through static and dynamic mechanisms. The static approach models non-deterministic features with deterministic replacements. The dynamic approach detects and repairs non-determinism as it is observed.

3.4.1 Static Non-determinism

For bit-level reproducible replay, all sources of non-determinism in a program must be ruled out. As stated in Section 3.3.1, the system call layer is totally modeled in the interpreter and side effects are logged. However, several sources of non-determinism need special care beyond register and memory side effect logging:

- **rdtsc.** Reads from a system timestamp counter. The value is hard-wired to 1 by the JIT. For cross-checking with hardware, the instruction is detected and the mismatch overwritten through fix-ups (§3.4.2).

- **mmap.** Special care is taken to rewrite mmap calls to reuse the addresses given by the interpreter. Otherwise, the operating system could allocate at a different base address and violate pointer equivalence.

- **VDSO.** A few Linux system calls (e.g., clock_gettime) have a fast-path through the VDSO library. These system calls access special read-only system pages (the vsyscall region) with memory mapped timers. No system calls are dispatched so it is difficult to account for these accesses on hardware or model the side-effects. Instead, each guest process’s VDSO library is overwritten with a custom VDSO library that uses slow-path system calls instead.

3.4.2 Dynamic Non-determinism

Machine instruction translation occasionally fails to match hardware in such a way that is either difficult or impossible to correct a priori. Although the translation
can be plainly wrong by diverging from the architectural specification, some instruction results may be undefined, micro-architecture dependent, or defined as non-deterministic. When the system encounters a problematic instruction, it corrects the mismatch with fix-ups that replace bad values with expected values. The following is a partial list of fix-ups:

- **rdtsc**: In native execution, the opcode is caught by single-stepping instructions, and a constant is injected into the native register in place of a timestamp.

- **cpuid**: stores processor information into registers. VEX returns a description that matches its feature set, but pretends to be a genuine CPU (e.g., an Intel Core i5). To fix up the native process to match the guest, the opcode is caught when stepping and overridden to use the VEX description.

- **pushf** – Stores **ptrace** control information to the stack. Single stepping the native process with **ptrace** is based on a processor feature which requires a trap flag to be set in the **eflags** register. VEX, on the other hand, is unaware. Hence, the two states will never have equal **eflags** registers. Our solution gives preference to running applications without single-stepping: the native opcode is caught and its result overridden to mask away the flag.

- **bsf, bsr** – The VEX IR overwrites the top half of the 64-bit register for the 32-bit variant of these instructions. The native register value is copied to the affected JIT register.

### 3.5 Experiments and Evaluation

This section reports measurements and faulty behaviors from running KLEE-MC. First, we show the system’s ability to cope with binary diversity by testing over ten thousand Linux binaries. Next we find system modeling differences derived by comparing operating system calls and symbolic system model side-effects from host execution traces. Finally, we check core interpreter functionality by verifying soft expression reduction rules and showing run-time bugs in hard-coded expression builders.
3.5.1 Linux Program Test Cases

KLEE-MC is designed to find bugs in user-level binary programs. The system is mature enough to construct test inputs that reliably crash programs. Regardless of this initial success, the integrity mechanisms detect critical bugs which still threaten the quality of symbolic execution.

Types of Mismatches

We recognize three error modes which roughly correspond to the symbolic interpreter, machine-code translation, and hardware respectively:

System call mismatch. Either the system call log is depleted early or the sequence of system calls diverges from the log on replay. This exercises replay determinism and coarsely detects interpreter bugs.

Register log mismatch (Interpreter × JIT). The interpreter’s intermediate register values conflict with the JIT’s register values. This detects symbolic interpretation bugs at the basic block level.

Hardware mismatch (JIT × HW). The JIT’s intermediate register values conflict with host hardware register values. This detects machine-code translation bugs otherwise missed by register logging.

Testing Ubuntu Linux

We tested program binaries from Ubuntu 13.10 for the x86-64 architecture. Each program was symbolically executed for five minutes, one to a single core of an 8-core x86-64 machine, with a five second solver timeout. The symbolic execution produced test cases which exercise many program paths, including paths leading to crashes. These paths were then checked for computational mismatches against JIT replay, register logs, and native hardware execution.

Table 3.1 lists a summary of the test results. Crash and mismatch data are given in terms of total tests along with unique programs in parentheses. In total we confirmed 4410 pointer faults (97%), all divide by zeros, and 38 (30%) decode and jump errors. Surprisingly, the overlap between mismatch types was very low—3.8% for hardware
CHAPTER 3. MACHINE CROSS-CHECKING

<table>
<thead>
<tr>
<th>Programs</th>
<th>14866</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>500617</td>
</tr>
<tr>
<td>Solver Queries</td>
<td>44.8M</td>
</tr>
<tr>
<td>Test Case Size</td>
<td>114MB</td>
</tr>
<tr>
<td>Register Log Size</td>
<td>235GB</td>
</tr>
<tr>
<td>Pointer Faults</td>
<td>4551 (2540)</td>
</tr>
<tr>
<td>Divide By Zero</td>
<td>109 (84)</td>
</tr>
<tr>
<td>Decode / Bad jump</td>
<td>126 (39)</td>
</tr>
<tr>
<td>Unsupported Syscalls</td>
<td>80</td>
</tr>
<tr>
<td>Fix-Ups</td>
<td>742 (48)</td>
</tr>
<tr>
<td>Syscall Mismatch</td>
<td>4315 (390)</td>
</tr>
<tr>
<td>Interpreter × JIT</td>
<td>2267 (214)</td>
</tr>
<tr>
<td>JIT × HW</td>
<td>4143 (201)</td>
</tr>
</tbody>
</table>

Table 3.1: Mass checking and cross-checking results on Ubuntu Linux programs.

and register logs and 11% for register log and system call mismatches. This suggests each checking mechanism has its own niche for detecting executor bugs.

Example Mismatch

Figure 3.9 gives an example deep translation bug with a small assembly code snippet detected through a register log mismatch. The assembly code clears the 128-bit xmm0 register with xorpd, computes the double-precision value the division 0.0/0.0, storing the result to the lower 64-bit position in xmm0. The machine code front-end translates and optimizes the instructions to putting the value (0, 0.0/0.0) into the xmm0 register. Next, converting to LLVM causes the LLVM builder to constant-fold 0.0/0.0 into 0x7ff80..0 (nan). Symbolically executing the LLVM code gives (nan, 0), running the LLVM code with the JIT gives (0, nan), and the original instructions give (0, ~nan) natively. The symbolic executor diverges from the JIT because it builds the LLVM constant data sequence (0, nan) backwards. The JIT diverges from hardware because LLVM constant-folding computes 0.0/0.0 as nan instead of ~nan. Fixing the symbolic executor simply requires swapping the concatenation order. Since LLVM causes the JIT × hardware divergence, the divsd %xmm0, %xmm0 is included as a safe fix-up so the mismatch is repaired during cross-checking.
Assembly

\[
\begin{align*}
xorpd & \%xmm0, \%xmm0 \\
divsd & \%xmm0, \%xmm0
\end{align*}
\]

VexIR

\[
\text{Put}(224) \leftarrow \text{Iop\_Div64F0x2}(0:V128, 0:V128)
\]

LLVM Code

\[
\begin{align*}
\%ctx &= \text{bitcast} \%\text{guestCtxTy}* \%0 \text{ to } <2 \times \text{double}>* \\
\%\text{XMM}[0]^* &= \text{getelementptr} <2 \times \text{double}>* \%\text{ctx}, \%i32 14 \\
\text{store} <2 \times \text{double}>* <\text{double }0x7ff8000000000000, \text{double }0.000000e+00>, \\
\text{<2 x double>}>%%\text{XMM}[0]^*
\end{align*}
\]

Register Values

<table>
<thead>
<tr>
<th>SymEx</th>
<th>XMM0: (0x7ff8000000000000, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT</td>
<td>XMM0: (0, 0x7ff8000000000000)</td>
</tr>
<tr>
<td>Hardware</td>
<td>XMM0: (0, 0xfff8000000000000)</td>
</tr>
</tbody>
</table>

Interpreter Bug

\[
\begin{align*}
\text{ref<Expr> ConstantExpr::createSeqData}( \\
\text{llvm::ConstantDataSequential}* v)
\{ \ldots \\
\text{for (unsigned i = 0; i < elem_c; i++)} \\
\text{ref<ConstantExpr> ce; } \\
\ldots \\
\text{cur_v} = (i == 0) ? ce : cur_v->Concat(ce); } \}
\text{return cur_v; } \}
\]

Figure 3.9: A broken floating-point division sequence.

Register Logs

Register log can consume excessive memory because the executor stores a new register set for every basic block. In the interpreter, the log is stored in memory until path termination, where it is written, compressed, to the file system. Since the memory overhead makes logging impractical during the symbolic execution run, register logs are recreated through test case replay. Figure 3.10 shows the time it takes to reconstruct a register log from a test case. Since paths tend to be short, approximately 90% of test cases can have their register logs reconstructed in under ten seconds. Since a few missing logs are of little importance, the process memory was limited so the
interceptor would terminate, thus losing the log, without exhausting all total memory. Offline reconstruction is a good trade-off; complicated efficient online logging is unnecessary because the data can be quickly rebuilt from test cases.

Figure 3.11 plots the number of basic blocks checked for each test case. As expected, there is a linear relationship between number of basic blocks checked and time to check the entire test case. The most frequent tests have fewer basic blocks. The longest logs check more than a million basic blocks, approaching the executor memory limit with more than a gigabyte of log data.

### 3.5.2 Machine Code Front-End

We ran klee-mc on 39 cores of a cluster for two hours, where each core explored a different partial path in the VEX x86-64 instruction decoder. This method generated a total of 40,270 test fragments (both decodable and undecodable). Conservatively, this gives 17 code fragments per CPU-minute. Generated code fragments are fed into the JIT and cross-checked with the native machine using a fixed concrete register file.

The system solves constraints to generate fragments of concrete code, then runs them natively. If VEX rejects a fragment, no solution should fault. Conversely, if
Figure 3.11: Number of basic blocks checked against register log as a function of time.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop esp</td>
<td>0f 19 cc</td>
<td>vex amd64-&gt;IR: unhandled instruction bytes</td>
</tr>
<tr>
<td>rex.w popfq</td>
<td>48 9d</td>
<td>Assertion ‘sz == 2</td>
</tr>
<tr>
<td>fs stosb</td>
<td>66 aa</td>
<td>disAMode(amd64): not an addr</td>
</tr>
<tr>
<td>rex.w ss add eax, 0xcccccccc</td>
<td>48 36 1d cc cc cc cc</td>
<td>Keeps top half of rax</td>
</tr>
<tr>
<td>rex.w ss mov ch, 0xcc</td>
<td>48 36 b5 cc</td>
<td>Stores to first byte of rbp</td>
</tr>
<tr>
<td>rol rax, 0x11</td>
<td>48 c1 c0 11</td>
<td>Sets undefined overflow flag</td>
</tr>
<tr>
<td>shl cl, r9</td>
<td>49 d3 e1</td>
<td>Sets adjust flag</td>
</tr>
</tbody>
</table>

Table 3.2: Valid x86-64 code causing VEX to panic, corrupt register state, or invoke divergent behavior.
accepting, every solution should run natively without raising an illegal instruction violation. Further, any non-faulting sequence should yield the same register values at the end. As expected, we found numerous places where these checks failed, indicating bugs in VEX. Table 3.2 shows the flavor of opcodes and bugs which come from symbolically generated fragments.

No special effort is made to support memory accesses, so most trap. When a fragment makes an access to an invalid pointer, the harness would catch the access, log the address and value (if a write), patch in a return value (if a read), then return control to the fragment. To keep results pure, we discard from consideration any code fragment which causes a memory access violation. Despite this drawback we find many decoding errors.

Cross-checking symbolically generated code with a concrete register file was surprisingly effective: 8,274 out of the forty-thousand code fragments produced mismatches. This number is after filtering errors from bad jumps, bad memory accesses, and obvious unhandled cases in our JIT dispatcher. We crudely culled duplicate bugs by binning all errors that had the same unique register values (but different instructions), which brought the total number down to 86 errors. A gross classification of the bugs uncovered by the concrete register file is listed in Table 3.3. The bugs are classified into four categories:

1. **Corrupted Registers**: A register in the native state disagrees with its guest counterpart after running the fragment. Mismatches stem from overzealous truncation, mixed up operands, and nondeterminism.

2. **Assert Failures**: `assert` failures thrown within the VEX decoder. The binned
number is very conservative since we only count unique assertion messages (i.e., the same error message twice is one error counted); VEX has few messages for rejections, so distinct opcodes are rejected with the same message.

3. **False Reject**: VEX rejects an instruction sequence which native hardware accepts. This number is conservative. Unique occurrences are counted based on the signal returned by the `ptrace`’d process (e.g., SIGBUS, SIGTRAP, SIGSEGV); a correctly rejected instruction would return the illegal instruction signal, SIGILL. We excluded 1,607 code fragments from the false reject count because raise SIGSEGV as a consequence of the incomplete memory model.

4. **False Accept**: VEX accepts an instruction which is illegal on the host machine. Only SIGILL is expected; this test yields one “unique” mismatch.

Of the four classes, we believe corrupted register errors are most serious. If a corrupted register is live after completion of the instruction, it will introduce garbage into the computation. Occasionally the bogus computation will fail fast (e.g., the corrupted register was a pointer). Without cross-checking, the computation may proceed unhindered for some time, making the root cause uncertain.

We next evaluate the propensity toward false-negatives in concrete tests by rechecking matching fragments with symbolically derived concrete register files. From 11,379 fragments that cross-checked successfully, KLEE-MC generated 570,977 register files designed to exercise all paths in the fragments. Of the 11,379 fragments passed by concrete checking, 346 failed under symbolic registers. Errors stemmed from new paths, condition codes, and a mishandled or opcode masked by our choice of concrete register values.

### 3.5.3 System Model

Systematically applying model differencing detects both underconstrained and overconstrained system calls. To strenuously test the symbolic model, we used the system call test cases from the Linux Test Program [88] project (LTP). Although the system model occasionally diverged from the host, the differences were modest.
Table 3.4: Comparison between model and operating system with LTP suite.

<table>
<thead>
<tr>
<th>Type</th>
<th>System Calls</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underconstrained</td>
<td>11</td>
<td>133</td>
</tr>
<tr>
<td>Overconstrained</td>
<td>34</td>
<td>1586</td>
</tr>
<tr>
<td>No difference</td>
<td>152</td>
<td>39360</td>
</tr>
<tr>
<td>Total</td>
<td>201</td>
<td>41079</td>
</tr>
</tbody>
</table>

Each LTP program was snapshotted for up to one hundred system calls. LTP program demanding root access were ignored. To test the executor’s system model, each pre-call snapshot was symbolically executed to path completion for the call. This covers all possible results the system model could give for the snapshot’s system call. Next, the side-effects, from the pre and post snapshots and the system model, were compared for overconstrained and underconstrained modeling. Table 3.4 summarizes the model differencing data from the LTP programs. The results suggest that the model tends to be conservative, opting to leave memory untouched. It should be noted that LTP is not a complete test suite and therefore does not represent a full set of possible program-operating system interactions. For instance, LTP makes no attempt to seriously test the notoriously overloaded ioctl call (likewise, KLEE-MC makes little attempt to accurately model ioctl).

Overconstrained system model calls, those that left some data untouched compared to the host, typically flagged obviously missing modeling. Most calls were unimplemented or incomplete with remarks in the comments. In one case (getdents), the complete system call was commented out in favor of an incomplete version, for reasons unknown. Five system calls (not counted in the table) were accurately modeled (e.g., getpid), but were occasionally flagged as overconstrained because of system modifications to the read-only vsyscall page.

Underconstrained system calls, those that touched excess data, flagged environmental differences between the host and the symbolic system model. Often the system model fills in structures for resources that missing from the host system. As an example, the model (as a heuristic) permits failures of lstat only occasionally; otherwise, the stat structure is unconditionally updated.
3.5.4 Expressions

Testing Rules

Thousands of expression reduction rules in klee-mc are verified using the constraint solver. Although rule verification is offline, it is still important that soft rules can be verified within a reasonable about of time. Figure 3.12 shows the time to test the soft rule set currently used by klee-mc. The entire set of 19444 rules took under half an hour to verify with the STP constraint solver. All rules are verifiable in under thirty seconds with a median time of less than a tenth of a second.

On the other hand, testing the rules with Z3 [49] led to other conclusions. Individual rules could take more than a minute to confirm, far longer than STP’s worst case; in total, Z3 took 11.6× as long as STP to verify the rule set. Furthermore, Z3 handles undefined division edge cases differently from STP, causing it to reject 27 otherwise acceptable rules. Figure 3.13 shows an example rule verification expression that is valid in STP but contingent under Z3. The trace-derived rule states the 5th byte of four 64-bit divides of a zero-extended 32-bit value \(x\) by any value \(c\) will always be zero. Technically, dividing by zero could result in a value greater than the
CHAPTER 3. MACHINE CROSS-CHECKING

Figure 3.13: Translation rule test with undefined result.

numerator, but this seems unnecessarily pedantic.

**Hard-coded Builders**

We generated expressions to cross check both using actual programs and by using a simple “fuzzer” to construct them synthetically. Cross-checking the former found errors in both the optimizing and in the naïve builders, cross checking fuzz-produced expressions found an error in LLVM itself. We discuss a few in detail.

**Basic Builder.** Although the cross-checker is intended to find bugs in the optimizing expression builder, it still found an error in the simpler builder. Both builders attempted to replace an equality check of two identical constant values with \(1\) (true), the optimizing builder correctly, the simple builder incorrectly. The latter generated the boolean constant using an optimistically-named helper function, \(\text{True}()\). Unfortunately, \(\text{True}()\) was an exact copy of its negated counterpart and presumed paste progenitor, \(\text{False}()\). One might assume defining true as false would lead to immediate errors, but only one line called the function, allowing its effects to go unnoticed.

**Optimizing Builder.** The cross-checker immediately found several bugs, including a truncation error in the constraint marshaler and unsupported expression combinations. One of the most serious was a buggy routine to replace multiply operations (expensive in constraint solving) with shifts and adds. Figure 3.14 shows a subset of the information the cross checker reports — the expression that failed cross-checking, a concrete assignment which will make both routines differ, and the concrete values they produce. The broken multiplication algorithm relied on complicated bit tricks broken beyond repair; it was rewritten from scratch to use Booth’s
CHAPTER 3. MACHINE CROSS-CHECKING

\[
\text{Eq (NotOptimized w128 }
\begin{align*}
&\text{(Mul w128 0x1845c8a0ce512957} \\
&\text{N0:(SExt w128 (ReadLSB w64 88 statbuf8)))} \\
&\text{(Sub w128 0} \\
&\text{(Add w128 (Add w128 (Add w128 (Add w128} \\
&\text{ (Shl w128 N0 59) } \\
&\text{ (Shl w128 N0 46)) } \\
&\text{ (Shl w128 N0 30)) } \\
&\text{ (Shl w128 N0 25)) } \\
&\text{ (Shl w128 N0 0))}))
\end{align*}
\]

Counterexample: statbuf8[95] = 0x1 (N0 = 0x1000000000000000)
Oracle Cex Eval: 0x001845c8a0ce512957000000000000000
Test Cex Eval: 0xfff7ffbfffbdffffff00000000000000

Figure 3.14: Bad multiply caught by cross-checking.

\[
\text{const ZExtExpr } \ast zr, \ast zl; \\
\text{if } ((zr = dyn_cast<const ZExtExpr*>(r)) &&} \\
(zl = dyn_cast<const ZExtExpr*>(l)) && \\
zr->getKid(0)->getWidth() == \\
zl->getKid(0)->getWidth()) \{ \\
\text{return MK_SEXT(} \\
\text{MK_SUB(zl->getKid(0), zr->getKid(0)),} \\
zl->getWidth()); \}
\]

Figure 3.15: A faulty hard-coded optimization method.

Faulty optimizations occasionally creep into the code, but the cross-checker helps flag the contamination. Figure 3.15 lists one such optimization. The optimization is meant to optimize subtraction expressions. When the arguments \(x\) and \(y\) are zero-extended to \(n\) bits, \((- (\text{ZExt } x \ n) (\text{ZExt } y \ n))\), the code replaces the two zero-extensions with a single \(n\)-bit sign-extension, \((\text{SExt } (- x \ y) \ n)\). This reasoning is clearly incorrect (e.g., for 8-bit values \(x = 0x80, y = 0,\) and \(n = 16\)). An obvious mistake, but only arising in very specific circumstances. After being introduced to the code base, the bug was detected and fixed in time for the next commit.

LLVM. Cross-checking fuzz-produced expressions found a broken corner case in
LLVM's arbitrary precision integer code. The arithmetic right shift operation is mishandled for the single-bit case; LLVM incorrectly claims $1 >> 1 = 0$. This is wrong for arithmetic right shift because the shift is defined to round to negative infinity (i.e. 1). The SMT solver, however, gets it right. Fuzzing expressions loosely resembles fuzzing constraint solvers [26], although we assume the solver to be correct but not the expression builder. Since there are multiple SMT solvers, this assumption may be lifted by submitting queries to several solvers and holding an election.

## 3.6 Related Work

Much recent work uses symbolic execution to find bugs both on source code [28, 58, 75, 119] and on binaries [30, 34, 59, 65]. In the latter category, S$^2$E uses a QEMU-to-LLVM translator to convert x86 code into fragments for KLEE. Another system [65] uses the valgrind DBT, symbolically running the VEX IR directly. KLEE-MC shares code with both: KLEE is the symbolic executor core and VEX translates. We view our work as complementary to past efforts, which have been largely unreflective. Success meant bugs in other code, rather than turning the tool on itself for trustworthy results. We believe these other tools would benefit from the techniques in this chapter.

FuzzBALL [92] uses a symbolic binary executor to find edge cases in system-level emulation features and instruction decoding. KLEE-MC focuses on user-level code so the checks in KLEE-MC support self-hosting, whereas FuzzBALL excels at manipulating system registers. Finally, KLEE-MC tests its own decoder; FuzzBALL uses VEX as an internal decoder, but ignores VEX as a target because VEX only models a user-mode environment.

Validation, or on-demand verification of individual paths, is a practical alternative to full-fledged verification of program code. Translation validation has been used to validate compilation by verifying target code for a given source program [107], intermediate forms in optimizer passes in the GNU C compiler [98], LLVM optimizer passes [126], and program patches [94]. Instead of targeting compilation, we validate expressions used during symbolic execution; optimized expressions are validated against unoptimized expressions by reusing the constraint solver infrastructure from
symbolic resolution.

For hardware, co-simulation [48] is a verification heuristic which validates experimental hardware against a simulator oracle. KLEE-MC uses co-simulation with a reversal of roles: hardware is an oracle for software.

CSmith [135] generates well-defined C programs to find defects in C compilers. Both this work and ours share the use of cross-checking to dispense with needing an oracle for correctness. They use pure concrete execution rather than symbolic, targeting bugs in compilers rather than their infrastructure. Unlike KLEE-MC, which validates against hardware, CSmith lacks a ground truth C compiler (verified C compilers [84] remain experimental).

The verified L4 operating system micro-kernel [120] uses translation validation to verify C sources match compiled code. L4 translation validation uses a processor model to decode compiled code and checks it against a specification modeled by the C code. KLEE-MC is weaker in that it validates code semantics through tests instead of verification, but it targets more code; the binary translation models an x86-64 processor which is checked through native execution on the host hardware.

3.7 Conclusion

In our work on building and validating a dynamic binary symbolic execution stack, we developed cross-checks at points where manual debugging would be intricate and frustrating. Validating paths in the system boosted our confidence when implementing new, experimental features. Furthermore, flagging mismatches at the basic block level virtually eliminated many challenges with reproducing runtime bugs.

Engineering a robust bug checker is the next logical step for symbolic execution. Much progress has been made in systematically finding bugs in third party programs. However, it is time to detect bugs in the checker for better confidence in the process itself. We have presented cross-checking in a symbolic executor as a method to find errors in program evaluation given a complex dynamic program analysis system. The symbolic executor detected thousands of serious program errors in commodity third party binaries, all of which can be mechanically confirmed through a high-integrity
test replay system. Likewise, we found many errors in the analysis tool itself by testing with a generous program set. Considering the results, we believe cross-checking is the easiest and fastest way to find binary symbolic interpreter bugs and develop a high-quality binary program analysis tool.
Chapter 4

Off-the-Shelf Symbolic Floating-Point

4.1 Introduction

To obtain a satisfying variable assignment for a constraint set, a symbolic executor usually represents constraints as expressions over a theory of bit-vectors and arrays which is solved by some decision procedure or theorem prover. Deciding satisfiability modulo the theory of bit-vectors is meant for integer workloads; expressions consist of two’s complement arithmetic, integer comparisons, and bitwise operators.

Solving for expressions over floating-point operations requires additional effort and is considered a significant challenge in model checking [1]. There is little agreement on how to best handle symbolic floating-point data in a symbolic executor; in fact, several classes of floating-point support have been proposed. The simplest support evaluates only concrete data [28], which is fast and sound, but incomplete. Another approach, but still incomplete, applies taint analysis [57] and floating-point expression matching [41] to detect suspicious paths. The most challenging, complete and accurate symbolic floating-point semantics, relies on the flawless reasoning of a floating-point solver [6, 10, 20].

Accurate floating-point is essential for checking software. The de facto floating-point standard, IEEE-754 [71], fully describes a floating-point arithmetic model; it is
subtle and complicated. A cautious software author must account for lurid details\cite{61} such as infinities, not-a-numbers (NaNs), denormals, and rounding modes. Ignoring these details is convenient once the software appears to work but defects then arise from malicious or unintended inputs as a consequence.

Fortunately, IEEE-754 emulation libraries already encode the specifics of floating-point in software. Namely, a soft floating-point library emulates IEEE-754 operations with integer instructions. These libraries are a fixture in operating systems; unimplemented floating-point instructions trap into software handlers. Elsewhere, soft floating-point shared libraries assist when the instruction set lacks floating-point instructions.

This work presents an integer-only binary symbolic executor augmented to support floating-point instructions through soft floating-point libraries. Five off-the-shelf soft floating-point libraries are adapted to the symbolic executor by mapping soft floating-point library operations to a unified runtime interface. Floating-point instructions are mapped to integer code by transparently replacing program instructions with calls to soft floating-point runtime functions, an idea that can apply to nearly any symbolic executor. Aside from testing floating-point paths in general purpose programs, binary symbolic execution with soft floating-point provides a novel means for testing floating-point emulation libraries and floating-point theorem provers.

The rest of this chapter is structured as follows. Section 2 discusses related work, relevant background, and the motivation behind soft floating-point in a symbolic executor. Section 3 describes the operation and implementation of soft floating-point in a binary symbolic executor. Section 4 analyzes operation integrity through symbolic execution of soft floating-point libraries with symbolic operands by comparing the generated test cases against native evaluation on host hardware. Section 5 continues by testing floating-point SMT solvers for inconsistencies with hardware. Section 6 considers general purpose applications by examining errors in Linux binaries found through symbolic execution with soft floating-point. Finally, Section 7 concludes.
CHAPTER 4. OFF-THE-SHELF SYMBOLIC FLOATING-POINT

4.2 Related Work

Testing and verification of floating-point software is the topic of much study. At the most primitive level, finite-width floating-point variables are reconciled with real and rational arithmetic and are the focus of floating-point decision procedures. Static analysis of source code leverages these primitives to find floating-point bugs but is limited to only an approximation of execution. A symbolic executor of floating-point code dynamically executes programs and must balance performance, completeness, and soundness with different workload needs. If floating-point data is concrete or uninteresting symbolically, symbolic execution of floating-point operations may be exclusively concrete [28]. Fast, but unsound, symbolic floating-point, useful for bug finding, applies canonicalization and matching on floating-point expressions [41, 57, 87]. When complete semantics are necessary, precise symbolic floating-point integrates a floating-point solver into the symbolic execution stack [6, 10, 20, 82]. This work introduces a new point in this progression: symbolically executed soft floating-point with integer code.

Many techniques have been developed to handle abstract floating-point data. Abstract domains [43], interval propagation [47], and abstraction refinement [38] are some influential approaches for computing solutions to value constraints. Such concepts have been extended, improved, and refined for floating-point values through exact projections [20], filtering by maximum units in the last place [6], interval linear forms [95], monitor variables [72], saturation with simplex bounding [42], conflict analysis over lattice abstractions [64], and guided approximation transformations of formulas [22], among others. For ease of use, decision procedures based on these strategies may be integrated into a solver back-end [6, 20, 22, 42, 64]. For hardware, formal floating-point specifications have been used to verify the correctness of a gate-level description of a floating-point unit [102].

Static analysis of source code to find floating-point bugs includes a broad class of notable abstract interpretation systems. The FLUCTUAT [63] system models floating-point values in C programs with affine arithmetic over noise symbols to locate sources of rounding error with respect to real numbers. ASTRÉE [19] is based on an interval
abstraction that uses a multi-abstraction, specializable, domain-aware analysis to prove the absence of overflows and other errors for source programs written in a subset of the C language.

In the context of symbolic execution, the simplest floating-point tactic discards symbolic data in favor of processing floating-point operations through concretization [28]. To dispatch a floating-point operation on symbolic data, each operand is constrained to a satisfying variable assignment (concretized) and the operation is evaluated. As an example, if \( x \) is unconstrained and 0 is the default assignment, computing \( x + 1.0 \) concretizes to \( 0.0 + 1.0 \). Concretization is fast but it overconstrains the variable term \( x \) and discards most feasible values (in this case, \( x \neq 0.0 \)).

A symbolic executor with expression matching also avoids the difficulty of supporting full floating-point semantics. These symbolic executors broaden the expression language to include floating-point operators but only impose additional structure on expressions; the operations are purely superficial. Taint tracking, where data becomes “contaminated” according to a taint propagation policy and operations on tainted data are flagged according to a detection policy, is one instance of expression analysis on symbolic floating-point; floating-point operations tag expressions as tainted and dereferences of tagged pointers are flagged [57]. Beyond tainting, comparison of expression structure [41, 87] demonstrates equivalence between algorithm implementations. This analysis is often unsound; spurious errors are possible.

Precise symbolic floating-point reasons about the underlying semantics by using a floating-point solver. Although explicit reasoning is accurate and often complete by design, it demands a careful solver implementation (tested in Section 4.5) and invasive executor modifications [10]. In some cases, authors of these systems use the symbolic executor as a platform to test their own floating-point decision algorithms in lieu of a third-party IEEE-754 solver [6, 20, 82].

This chapter proposes symbolically executed soft floating-point, a compromise between concretization and full symbolic floating-point. Where a floating-point solver may model a formula as disjunctions of several feasible subformulas, soft floating-point models that formula with many states. A soft floating-point operation partially concretizes on program control at contingent branches by forking off multiple states.
These states partition floating-point values by disjunction but together represent the set of all feasible floating-point values. Additional states are costly but soft floating-point remains attractive because the complicated floating-point component is off-the-shelf software which requires little support code and no changes to the core symbolic execution system. Aside from its simplicity, soft floating-point is self-testing in that tests can be efficiently derived directly from the libraries using symbolic executor, a property explored in Section 4.4 and applied to floating-point solvers in Section 4.5.

### 4.3 Soft Floating-Point

This section details the implementation of a soft floating-point extension for an integer-only symbolic binary executor. First, an abstract soft floating-point library is defined by its set of floating-point operations. Next, the components for the base binary symbolic executor are briefly outlined. Finally, a description of a runtime interface and implementation establishes the connection between the symbolic executor and several soft floating-point libraries.

#### 4.3.1 Floating-Point Operations

A soft floating-point library is a collection of idempotent integer-only operation functions which model an IEEE-754-1985 [71] compliant floating-point unit. The client code bitcasts floating-point data (single or double precision) into integer operands; the library unpacks the sign, mantissa, and exponent components (Figure 4.1) with bitwise operators into distinct integers. Operations evaluate the components, then pack and return floating-point results in IEEE-754 format.
Arithmetic

IEEE-754 defines the four-function arithmetic operations and remainder: +, −, ∗, /, and %. Arithmetic operations are complete floating-point valued functions over single and double-precision pairs. A major repercussion of floating-point arithmetic is many desirable invariants from real numbers and two’s-complement are lost: addition is non-associative, subtraction has cancellation error, and division by zero is well-defined.

Comparisons

Conditions on floating-point values are computed with comparison functions. Comparisons are defined for all pairs of 32-bit and 64-bit floating-point values and are represented with the usual symbols (i.e., =, ≠, >, <, ≥, and ≤). Evaluation returns the integer 1 when true, and 0 when false.

Comparisons take either an ordered or unordered mode. The mode determines the behavior of the comparison on non-number values. An ordered comparison may only be true when neither operand is a \(\text{NaN}\). An unordered comparison is true if either operand is a \(\text{NaN}\). During testing, only ordered comparisons were observed in code, so the two were never confused when evaluating floating-point code.

Type-Conversion

Type conversion translates a value from one type to another; floating-point values may be rounded to integers and back, or between single and double precision. In general, rounding is necessary for type conversion. Additionally, values may be rounded to zero, down to \(-\infty\), up to \(\infty\), or to nearest, depending on the rounding mode. However, only the round nearest mode appeared in program code during testing. There are several ways a floating-point computation may be rounded for type conversion:

- **Truncation and Expansion** (↔). Data is translated between single and double precision. Mantissa bits may be lost and values can overflow.

- **Integer Source** (\(f\leftarrow i\)). Conversion from integer to float. The integer may exceed the mantissa precision.
• **Integer Target** \((f \rightarrow i)\). Conversion from float to integer; \(\text{NaN}\) and \(\infty\) values may be converted.

**Elementary Functions**

The only elementary function required by IEEE-754-1985 is the square root function. Hence, all soft floating-point libraries support it. Transcendental functions, on the other hand, were deemed too costly to due to the precision necessary for correctness to the half-unit in the last place (the table-maker’s dilemma [73]); few soft floating-point libraries support these functions because they are not part of the standard. Although present in some instruction sets (e.g., x86), transcendental functions are treated as unsupported non-standard extensions; these extensions can be implemented as intermediate code intrinsics (§ 2.4.5) using standard floating-point operations if necessary for instruction emulation.

### 4.3.2 Runtime Libraries

The soft floating-point extended symbolic interpreter handles floating-point operations by calling out to a runtime library with a standard interface. All floating-point instructions are replaced with runtime function calls that manipulate floating-point data with integer instructions; the target program is essentially dynamically linked to the floating point library by way of its floating-point instructions. Internally, the runtime libraries differ on a fundamental level by the data encoding used for computation. Floating-point emulation code is readily available and easily portable; open-source operating systems supply many distinct library implementations which need few modifications to work with the symbolic executor.

A soft floating-point library, loaded as part of the klee LLVM bytecode runtime, encodes data operations in integer terms with an idempotent function call interface. If the library is correct and completely specified, then every floating-point operation is fully modeled; there is no need to re-encode the details. If the library is wrong or incomplete, it can be detected by testing for faulty inputs and cross-testing with other implementations (§ 4.4). To map floating-point code to the integer-only interpreter,
the program code is rewritten with soft floating-point calls. To validate the design, the runtime supports five off-the-shelf soft floating-point implementations: bsdhppa (PA-RISC from NetBSD), bsdppc (PowerPC from NetBSD), linmips (MIPS from Linux), softfloat (the SoftFloat library [67]), and shotgun (from an emulator [74]).

**Instruction Rewriting**

A function pass rewrites program code to call soft floating-point runtime functions in place of LLVM floating-point instructions. The pass replaces every floating-point instruction in the program code with a call to a type thunking stub. The thunk function bitcasts the operands to integer types (no bits change) and jumps to the corresponding runtime library function. At execution time, these integer-only runtime functions compute all floating-point operations.

**Interface**

For basic functionality, the standard interface uses a strict subset of the SoftFloat [67] library. SoftFloat features a set of functions which take floats and doubles bitcast to unsigned integers and return bitcast results. All other soft floating-point libraries require small custom SoftFloat interface adapters. This standardization simplifies instruction rewriting with a single target interface.

LLVM instructions are rewritten as function calls to their SoftFloat counterparts. Table 4.1 lists the functions which replace LLVM instructions for symbolic interpretation. The instructions encode arithmetic, comparisons, and rounding which are handled by the soft floating-point functions.

Floating-point operation handlers are stored as interchangeable libraries for the interpreter. Depending on the emulation code, each library is compiled from C to LLVM bitcode (a binary representation of an LLVM assembly listing). Since a bitcode library is native to the KLEE LLVM machine, it can not support hand-coded assembly which is found in some soft floating-point implementations. Although KLEE-MC can process machine instructions, handling the additional context necessary for machine code runtime libraries appears to require considerable restructuring and gross
Table 4.1: Rewritten LLVM instructions with corresponding SoftFloat functions

overengineering of both the JIT and symbolic executor basic block dispatch code.

Software Encodings

Floating-point unpacking policies are either selective or unselective. SoftFloat selectively masks components out as needed to local variables. Both bsdhppa and bsdppc, on the other hand, completely unpack floating-point values into a data structure before every operation and repack the result into the IEEE-754 format. The best representation depends on the circumstance. SoftFloat has a single compilation unit; the representation likely benefits from interprocedural analysis. BSD has multiple compilation units; mask calculations like SoftFloat’s would repeat at function boundaries.

Operating System Handlers as Libraries

The runtime uses off-the-shelf operating system code ported from one Linux and two BSD floating-point emulators. The soft floating-point implementations in operating systems are often quite good and at least heavily tested. This is because on several machine architectures (e.g., x86, ARM, MIPS), an operating system may be expected to emulate a hardware floating-point unit through software. Correctness and reproducibility demand that the software emulation closely matches hardware, hence operating systems should have accurate soft floating-point implementations.

In many cases, soft floating-point code can be compiled into LLVM bitcode for the runtime bytecode library. The floating-point emulation layer is intended to run on the
target operating system architecture and is usually written in C. After adjusting a few header files, the code can be compiled independently into a bitcode library. Handlers accelerated with assembly code (e.g., x86 Linux, ARM BSD), however, must compile to native machine code.

The soft floating-point library must extract the relevant functionality from the rest of the operating system code. In contrast to the symbolic executor’s instruction rewriting pass, an operating system’s native emulation mechanism traps and emulates missing floating-point instructions. Whenever a user program issues a floating-point instruction, control is trapped and vectored to the operating system. The trapped instruction is decoded into a soft floating-point computation; the computation uses only integer instructions and stores the result to the machine state. Finally, control returns to the next program instruction. Since the symbolic executor rewrites floating-point instructions instead of trapping, the executor bypasses the decoding logic and directly calls the floating-point operations.

Library-specific SoftFloat glue code translates internal floating-point calls to the standardized SoftFloat interface. The internal functions never trap or decode instructions, so those stages are ignored and inaccessible. Porting requires relatively few lines of code; glue code for linmips, bsdhppa, and bsdppc is between 50 and 150 lines of C source.

### 4.4 Operation Integrity

Once a floating-point library is in place, it is possible to test the library. Test cases are gathered by symbolically executing a binary program with a given soft floating-point library for each floating-point operation. Cross-checking intermediate path data with the JIT over the host’s hardware floating-point unit detects interpreter and soft floating-point inconsistencies. Pooling tests by operation across all libraries addresses the problem of single library underspecification; tests for a single library may cover all paths with correct results but still have inputs that do not match hardware. The JIT, with soft floating-point libraries and without, is checked against hardware execution of the binary on the pool of all test cases. Every library disagrees with hardware; some
patterns emerge as common pitfalls. Finally, significant library coverage confirms the thoroughness of testing.

When comparing soft floating-point libraries with hardware, it is useful to distinguish between consistency and verification. If tests derived from a library $\mathcal{L}$ all match hardware, then $\mathcal{L}$ is consistent with hardware under a symbolic executor: for every library path there is a test case which matches hardware. When $\mathcal{L}$ is consistent on an operation $\circ$ it is $\circ$-consistent. Verification against hardware is stronger than consistency: all possible inputs for $\mathcal{L}$ must match hardware. Consistency without verification arises when an underspecified program misses edge cases which describe inconsistent test cases. Consequentially, tests from a library $\mathcal{L}^*$ may induce a mismatch on a consistent but underspecified library $\mathcal{L}$.

The soft floating-point code is tested in two phases. In the first phase, operation programs are symbolically executed to produce test cases for each library. To determine consistency, the symbolic executor is cross-checked with the LLVM JIT’s machine code through a log replay mechanism that compares intermediate concrete values (§ 3.3.3). If the emulation is wrong, symbolic interpretation diverges from the JIT values, potentially causing false positives and negatives with respect to native execution. In the second phase, to handle underspecification, operations are cross-tested on a test case pool and checked against an Intel Core2 processor. Further, the pool tests the JIT, which compiles floating-point LLVM instructions to machine code, and raises several translation errors.

Figure 4.2 illustrates the process of cross-checking floating-point symbolic interpretation with a JIT and native hardware. To generate test cases, an x86-64 program binary is symbolically executed with KLEE-MC and a soft floating-point bitcode library. The test’s register log (§ 3.3.1) then replay cross-checks the library results against the LLVM JIT engine executing native floating-point instructions (Interpreter × JIT). Test case values are inputs for hardware cross-checking (JIT × Hardware); the soft floating-point JIT is cross-checked, basic code block by basic block, against the registers from native execution of the binary program.
4.4.1 Gathering Test Cases

Each floating-point operation is modeled through a distinct test program binary. These test programs, which we call operation programs, are compiled from small C source files into x86-64 Linux programs which read IEEE-754 formatted operands from the standard input. Only binary programs are evaluated, rather than C source or LLVM bitcode, to keep any compilation artifacts [97] the same among the symbolic executor, JIT, and hardware. Likewise, the operation programs test the machine code interface of the symbolic executor with compiled C, so some functionality is masked or inaccessible. This masking is particularly noticeable for the remainder (%) program which must use the math library fmod function because the C language only defines integer modulus operations.

The operation program reads operands from a symbolic standard input stream under symbolic execution. When the symbolic executor completes a path, it creates a test case by selecting a feasible input bit-string which satisfies the path constraints imposed by the soft floating-point runtime library. Feeding the bit-string into the operand program reproduces the floating-point library path.

Table 4.2 lists the number of test cases produced by the symbolic executor for each floating-point library. Test programs are split by precision: 32-bit single-precision (f32) and 64-bit double-precision (f64).
operation is a \textit{strict} upper bound on the number of states that will fork on a floating-point instruction in a symbolically executed program. Operations marked † and ‡ timed out exploring paths and solving queries respectively. Implementation differences lead to unique test case counts for each operation across libraries; some libraries fork more (bsdppc) than others (softfloat).

The test case counts wildly vary across libraries but some implementation details may be surmised from the data. Several patterns from the generated test cases qualitatively stand out. Operations are sometimes insensitive to precision; softfloat’s addition operation and the linmips less-than operation show the same total tests for both single-precision and double-precision. These equal test counts suggest the operations use essentially the same code for both precisions, only changing the bit-width. Libraries which have different test case counts could have additional, but unnecessary paths, for each precision. Likewise, equal tests across operations suggest very similar code; the = and ≠ operations may perform the same computation, then negate the

<table>
<thead>
<tr>
<th>Op.</th>
<th>bsdhppa</th>
<th>bsdppc</th>
<th>linmips</th>
<th>softfloat</th>
<th>softgun</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f32</td>
<td>f64</td>
<td>f32</td>
<td>f64</td>
<td>f32</td>
</tr>
<tr>
<td>+</td>
<td>122</td>
<td>1017</td>
<td>1670†</td>
<td>2016†</td>
<td>3741</td>
</tr>
<tr>
<td>-</td>
<td>122</td>
<td>1017</td>
<td>1615†</td>
<td>1717†</td>
<td>3738</td>
</tr>
<tr>
<td>*</td>
<td>3700‡</td>
<td>520‡</td>
<td>368‡</td>
<td>349‡</td>
<td>1945‡</td>
</tr>
<tr>
<td>/</td>
<td>6109‡</td>
<td>6268‡</td>
<td>2932‡</td>
<td>3520‡</td>
<td>4694‡</td>
</tr>
<tr>
<td>%</td>
<td>3247‡</td>
<td>3359‡</td>
<td>2</td>
<td>3680‡</td>
<td>3156‡</td>
</tr>
<tr>
<td>&lt;</td>
<td>28</td>
<td>32</td>
<td>1341†</td>
<td>1661†</td>
<td>91</td>
</tr>
<tr>
<td>≤</td>
<td>34</td>
<td>38</td>
<td>2034†</td>
<td>2353†</td>
<td>91</td>
</tr>
<tr>
<td>=</td>
<td>2890</td>
<td>42</td>
<td>1359†</td>
<td>1140†</td>
<td>2905</td>
</tr>
<tr>
<td>≠</td>
<td>2890</td>
<td>42</td>
<td>1402†</td>
<td>1413†</td>
<td>2905</td>
</tr>
<tr>
<td>↔i32</td>
<td>29</td>
<td>31</td>
<td>29</td>
<td>91</td>
<td>29</td>
</tr>
<tr>
<td>→i64</td>
<td>81</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>65</td>
</tr>
<tr>
<td>←i32</td>
<td>14</td>
<td>6</td>
<td>75</td>
<td>34</td>
<td>51</td>
</tr>
<tr>
<td>←i64</td>
<td>118</td>
<td>46</td>
<td>1466†</td>
<td>491</td>
<td>191</td>
</tr>
<tr>
<td>√x</td>
<td>3451†</td>
<td>2979†</td>
<td>2583†</td>
<td>4500†</td>
<td>4856†</td>
</tr>
<tr>
<td>Total</td>
<td>20087</td>
<td>15606</td>
<td>17115</td>
<td>23179</td>
<td>28555</td>
</tr>
</tbody>
</table>

Table 4.2: Floating-point operation test cases from symbolic execution.
solution if necessary. Single-precision surprisingly can have more paths than double-precision, such as for \( = \) and \( \rightarrow \text{i32} \), which may be due to compilation artifacts or optimizations inserted by the compiler.

A complete symbolic execution of an operation test program exhausts all paths. Every feasible path becomes a test input (e.g., pairs of floating-point values for binary operations) which satisfies the path constraints. Path exhaustion can be costly, however, so each program runs for a maximum time of one hour with a two minute solver timeout in order to enforce a reasonable limit on total execution time; most operations ran to completion. Relatively complicated operations, such as division, timed out across all libraries. The symbolic executor is sensitive to the branch organization of the library code so some simpler operations (e.g., + for \text{bsdppc} and \text{linmips}) time out from excessive state forking.

4.4.2 Cross-Checking for Consistency: Interpreter × JIT

Symbolically executing an operation \( \circ \) produces a set of test inputs which exercise distinct paths through the floating-point emulation library. Ideally, the symbolic interpreter exhausts all paths on \( \circ \), leading to a test case for every possible path on \( \circ \). The test cases are replayed concretely on the LLVM JIT and cross-checked with the symbolic interpreter’s concrete values to find bugs in the symbolic interpreter. Testing with cross-checking determines consistency; when all of \( \circ \)'s test cases for a library cross-check as matching, the library is \( \circ \)-consistent.

The symbolic interpreter is a custom LLVM interpreter which may diverge from bitcode semantics. To find divergences, the interpreter’s soft floating-point library results (in emulated hardware registers) are checked against natively executed LLVM JIT machine code for bit-equivalence at every dispatched basic block. These checks replay a test case on the JIT and cross-check with the interpreter’s concrete register log.

The number of failures for symbolically generated test cases is given in Table 4.3. There are three reasons for failure to cross-check: 1) the path terminated early (log runs out), 2) the floating-point library is wrong, or 3) the LLVM JIT engine emits
CHAPTER 4. OFF-THE-SHELF SYMBOLIC FLOATING-POINT

Table 4.3: JIT register log cross-checking failures on floating-point library self-tests

<table>
<thead>
<tr>
<th>Op.</th>
<th>bsdhppa f32</th>
<th>bsdhpc f32</th>
<th>linmips f32</th>
<th>softfloat f32</th>
<th>softgun f32</th>
<th>bsdhppa f64</th>
<th>bsdhpc f64</th>
<th>linmips f64</th>
<th>softfloat f64</th>
<th>softgun f64</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1 1 0 76</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1 1 0 5</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>2 30 29 28</td>
<td>0 8 0 0</td>
<td>0 0 0 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>2 1 2 0</td>
<td>0 0 0 29</td>
<td>0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>7 3 0 828</td>
<td>30 2 13 3</td>
<td>67 207 178</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>0 0 0 148</td>
<td>0 0 0 0</td>
<td>0 1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>≤</td>
<td>0 0 0 368</td>
<td>0 0 0 0</td>
<td>0 2 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>0 0 174 161</td>
<td>0 0 0 0</td>
<td>0 718 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>≠</td>
<td>0 0 233 213</td>
<td>0 0 0 0</td>
<td>718 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↔</td>
<td>0 0 6 1</td>
<td>1 2 0 1</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→i32</td>
<td>2 2 12 0</td>
<td>0 0 0 0</td>
<td>2 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>→i64</td>
<td>2 2 86 5</td>
<td>0 0 0 0</td>
<td>2 0 169 152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>←i32</td>
<td>0 0 30 19</td>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>←i64</td>
<td>0 1 312 76</td>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>√x</td>
<td>9 92 8 4</td>
<td>5 1 8 10</td>
<td>84 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>26 133 892 1931</td>
<td>36 13 25 42 1761 422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

wrong instructions. Failure to complete paths is demonstrated by softfloat division and linmips multiplication. All libraries fail cross-checking as highlighted in Table 4.4. It is likely the libraries have never been systematically cross-checked, so inconsistency is expected. Furthermore, JIT engine errors arise in √x and % for bsdhpc and softfloat, but are discussed in the next section because they require systematic hardware cross-checking for confirmation.

### 4.4.3 Cross-Testing for Underspecification Bugs

A consistent floating-point library may appear correct with path-exhaustive testing but it is still unverified over all values. Some components could be underspecified by missing edge cases, causing broken inputs to never materialize as paths. Underspecification may stem from the library (and therefore its tests) partially describing IEEE-754 semantics or the decoder mistranslating machine instructions. Additionally, consistency alone merely demonstrates interpreter and JIT equivalence over all
Table 4.4: Selected library bugs from soft floating-point library consistency tests

<table>
<thead>
<tr>
<th>Library</th>
<th>Operation</th>
<th>Library Soft FP</th>
<th>Hardware FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsdhppo</td>
<td>$\infty \times 0.0$</td>
<td>NaN</td>
<td>-NaN</td>
</tr>
<tr>
<td>bsdppc</td>
<td>1.1125...6e-308 + 1.1125...7e-308</td>
<td>5e-324</td>
<td>2.2250...1e-308</td>
</tr>
<tr>
<td>linmips</td>
<td>0x7ff00000001000000 (NaN) $\rightarrow$ i32</td>
<td>0x7fbfffffff (NaN)</td>
<td>0x7fc000000 (NaN)</td>
</tr>
<tr>
<td>softfloat</td>
<td>NaN $\rightarrow$ i32</td>
<td>0x7fffffff</td>
<td>0x80000000</td>
</tr>
<tr>
<td>softgun</td>
<td>0.0 / 0.0</td>
<td>0.0</td>
<td>-NaN</td>
</tr>
</tbody>
</table>

discovered paths; both can still possibly disagree with hardware. Pooling all tests for all discovered library paths and testing concrete JIT replay against native processes, on the other hand, finds bugs from underspecification and mistranslation.

Pooled Tests

Cross-checking detects only mismatches on known inputs; the symbolic executor must provide the input test cases. If a floating-point library is underspecified, then symbolic execution may not generate a test case for a hardware mismatch because no code necessarily describes the value. An underspecified library is incorrect despite being consistent. In fact, many operations cross-check as consistent; softfloat and linmips seem to perfectly convert floating-point to and from integers because the libraries are \{→i32,→i64, ←i32, ←i64\}-consistent. However, these consistent operations are not correct but underspecified; no path describes a bug under all feasible solutions, despite bugs being present when compared with hardware.

The underspecification problem can be partially mitigated by cross-testing across libraries. All tests are pooled and hardware cross-checked on all floating-point libraries. The test pool is applied to each library to cover values or conditions missed by underspecification.

Figure 4.3 illustrates the distinct value coverage for each library. The bsdppc library stands out as flooding comparison operations with unique values, agreeing with high fork rate in Table 4.2. Similarly, the linmips and bsdhppo libraries dominate the unique values for arithmetic operations. Heavy skewing essentially reflects implementation quirks. More single-precision values are shared among libraries than double-precision, possibly because there are fewer possible values for single-precision,
although there are more single-precision tests (54425) than double-precision (49164).
The unevenness of unique values among libraries and across operations suggests that
no one library produces a complete test suite of values.

There are two ways to account for the uneven test case distribution among the
libraries. One, even if distinct libraries compute the same result, the values for
test cases can differ due to how each implementation structures its code; different
decision trees lead to different partitions of values. Two, given incorrect emulation
of floating-point, the test case values from incorrect code may cover a divergence
between library and hardware floating-point algorithms. Ideally, the test cases would
uncover no divergences because no library would be underspecified.

In total, there were 103624 distinct test cases. The test count is 34 orders of
magnitude smaller than brute force testing \([(6(2^{32}+2^{64})+9(2^{64}+2^{128}))\] tests) because
there are fewer unique paths than unique inputs. However, so many tests may be
still relatively inefficient; one hand-designed suite [80] for \(\exp(x)\) uses 2554 tests.
When test count is a concern (e.g., tests are expensive), non-exhaustive execution
can give useful results. We found randomly dropping forked states on hot branches
still covered incorrect values for all libraries with 19560 distinct tests.
Table 4.5: Hardware cross-check errors from all distinct floating-point tests

<table>
<thead>
<tr>
<th>Op.</th>
<th>bsdhppa f32</th>
<th>bsdppc f32</th>
<th>linmips f32</th>
<th>softfloat f32</th>
<th>softgun f32</th>
<th>JIT f32</th>
<th>softfloat f64</th>
<th>softgun f64</th>
<th>JIT f64</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>603 135</td>
<td>7 8034</td>
<td>45 90</td>
<td>6 12</td>
<td>80 92</td>
<td>0 0</td>
<td>603 135</td>
<td>7 8034</td>
<td>45 90</td>
</tr>
<tr>
<td>-</td>
<td>623 109</td>
<td>47 5354</td>
<td>45 63</td>
<td>6 9</td>
<td>62 81</td>
<td>0 0</td>
<td>623 109</td>
<td>47 5354</td>
<td>45 63</td>
</tr>
<tr>
<td>*</td>
<td>50 53</td>
<td>8 1295</td>
<td>21 23</td>
<td>8 7</td>
<td>23 28</td>
<td>0 0</td>
<td>50 53</td>
<td>8 1295</td>
<td>21 23</td>
</tr>
<tr>
<td>/</td>
<td>56 52</td>
<td>2 831</td>
<td>32 28</td>
<td>2 4</td>
<td>37 41</td>
<td>0 0</td>
<td>56 52</td>
<td>2 831</td>
<td>32 28</td>
</tr>
<tr>
<td>%</td>
<td>176 123</td>
<td>134 13</td>
<td>176 58</td>
<td>9 7</td>
<td>4263 4638</td>
<td>35 3</td>
<td>176 123</td>
<td>134 13</td>
<td>176 58</td>
</tr>
<tr>
<td>&lt;</td>
<td>0 0</td>
<td>0 270</td>
<td>0 0</td>
<td>0 0</td>
<td>52 402</td>
<td>0 0</td>
<td>0 0</td>
<td>0 270</td>
<td>0 0</td>
</tr>
<tr>
<td>≤</td>
<td>0 0</td>
<td>0 405</td>
<td>0 0</td>
<td>0 0</td>
<td>72 609</td>
<td>0 0</td>
<td>0 0</td>
<td>0 405</td>
<td>0 0</td>
</tr>
<tr>
<td>=</td>
<td>0 0</td>
<td>650 7</td>
<td>0 0</td>
<td>0 0</td>
<td>4665 125</td>
<td>0 0</td>
<td>0 0</td>
<td>650 7</td>
<td>0 0</td>
</tr>
<tr>
<td>≠</td>
<td>0 0</td>
<td>669 6</td>
<td>0 0</td>
<td>0 0</td>
<td>4736 204</td>
<td>0 0</td>
<td>0 0</td>
<td>669 6</td>
<td>0 0</td>
</tr>
<tr>
<td>↔</td>
<td>5 84</td>
<td>49 19</td>
<td>4 7</td>
<td>0 0</td>
<td>2 4</td>
<td>0 0</td>
<td>5 84</td>
<td>49 19</td>
<td>4 7</td>
</tr>
<tr>
<td>→i32</td>
<td>40 32</td>
<td>86 38</td>
<td>25 26</td>
<td>40 31</td>
<td>25 26</td>
<td>0 0</td>
<td>40 32</td>
<td>86 38</td>
<td>25 26</td>
</tr>
<tr>
<td>→i64</td>
<td>153 76</td>
<td>304 121</td>
<td>24 27</td>
<td>47 36</td>
<td>257 264</td>
<td>0 0</td>
<td>153 76</td>
<td>304 121</td>
<td>24 27</td>
</tr>
<tr>
<td>←i32</td>
<td>147 75</td>
<td>147 75</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>147 75</td>
<td>147 75</td>
<td>0 0</td>
</tr>
<tr>
<td>←i64</td>
<td>1668 650</td>
<td>1671 651</td>
<td>55 0</td>
<td>55 0</td>
<td>55 0</td>
<td>55 0</td>
<td>1668 650</td>
<td>1671 651</td>
<td>55 0</td>
</tr>
<tr>
<td>√x</td>
<td>36 6</td>
<td>39 18</td>
<td>36 0</td>
<td>0 0</td>
<td>36 6</td>
<td>36 6</td>
<td>36 6</td>
<td>39 18</td>
<td>36 0</td>
</tr>
<tr>
<td>Total</td>
<td>3557 1395</td>
<td>3813 17137</td>
<td>463 322</td>
<td>173 106</td>
<td>14365 6520</td>
<td>126 9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Checking JIT × Hardware

Table 4.5 shows cross-checking errors found from cross-testing with a pool of foreign test cases. The JIT is cross-tested with a ptraced native shadow process; for every dispatched decoded instruction block, ptrace hardware registers are checked against the JIT’s emulated registers. A single bit difference is an error for a test case. Consistent operations, such as softfloat’s −, are found to be incorrect, because they do not match hardware, and underspecified, because only tests from other floating-point libraries reveal the defect. Some operations under some libraries show fewer errors for one precision than the other (or none at all); this discrepancy likely owes to these libraries having fundamentally different implementations for each precision and the uneven distribution of test values.

Applying the tests to the JIT engine, which is never symbolically executed, and comparing the registers with hardware execution revealed bugs on interesting edge cases. For instance, \( \sqrt{x} \) on a negative single-precision value returns NaN in the JIT,
but -NaN for hardware and softfloat. Errors in the JIT from translating machine code (a known problem for machine code interpreters on integer workloads [60, 92]) to LLVM bitcode appear across all libraries (e.g., f32 ← i64). Mismatches which appear solely for the JIT could be due to instruction selection.

For an in-depth example, Figure 4.4 reproduces the code involved for a f32 ← i64 conversion error. The operation program begins as x86-64 machine code which is translated by the VEX decoder into VEX IR, then from VEX IR into LLVM bitcode; if VEX mistranslates the machine code, the symbolic interpretation will be wrong. In the case of f32 ← i64, the x86-64 instruction cvtsi2ss converts a 64-bit signed integer to a single precision floating-point number. The corresponding VEX IR converts the signed integer into a double precision number, then to single precision. This induces rounding errors (confirmed by the VEX maintainer) that cause certain inputs (e.g., rbx = 72057598332895233) to evaluate one way natively (7.20576e+16) and another way through VEX (7.2057594e+16).

**4.4.4 Common Pitfalls**

Some effort was put into improving the library code’s cross-checking results before finalizing the data. The softfloat and linmips libraries were intended to be consistent whereas the consistency of bsdhppa and bsdppc was a lower priority. Most improvements concentrated on a few frequent problems.

**Endianness.** Architecture byte-order can conflict with the hardware byte-order. The PA-RISC (bsdhppa), PowerPC (bsdppc), and MIPS (linmips) architectures are big-endian but the host x86-64 machine is little-endian. MIPS is dual-endian so linmips has a #define to enable little-endian mode. For bsdhppa, the glue code must swap double-precision operands and results. bsdppc, evolved from SPARC and m68k
code bases, is staunchly big-endian. For instance, a 32-bit function for converting
double precision values expects the most significant half of a 64-bit value as its 32-bit
return result, a big-endian convention.

**Default NaN.** Certain operations, such as division by zero, produce a default
“quiet” NaN. Unfortunately, hardware is free to deterministically choose a QNaN from
$2^{23}$ bit-patterns. Both single and double precision QNaNs differed from the host ma-
chine for every library. The single-precision QNaN was 0x7ff00000, as opposed to
x86-64 hardware which uses 0xffc00000. Manual inspection of the Linux x87 emu-
lator confirmed the values on x86-64 matched the expected QNaN.

**NaN operands.** The x86-64 floating-point unit encodes extra diagnostic informa-
tion into the mantissa of its NaNs which disagrees with emulation. There are numerous
ways to mishandle a NaN operation so the bits do not match. For operations between
NaNs, a signaling NaN would sometimes be converted into the wrong QNaN. Arithmetic
between a NaN and number would use the default NaN, missing the mantissa bits.
Likewise, operands returning the left-hand NaN in hardware instead used the library
default NaN. Although this extra information is optional, it is still desirable to stay
bit-identical to the host hardware.

### 4.4.5 Coverage

Operation test cases raise plenty of mismatches but the depth of testing remains
unclear. Code coverage for each library is a simple metric for overall testing quality;
high coverage implies thorough testing. Figure 4.5 shows the coverage of each floating-
point implementation from symbolic execution. The instruction coverage percentages
are calculated from visited functions. The set of instructions is limited to visited
functions because soft floating-point libraries often have additional features which
are inaccessible through the interpreter (e.g., trapped instruction decoding). Total
covered instructions gauges the complexity, although not necessarily the correctness,
of the library implementation.

Between 79%–95% of instructions were covered by test cases for each library,
which leaves 5%–21% of instructions uncovered. There are several justifiable reasons
for missing instructions. All libraries support all rounding modes but only round-nearest is tested because it is the default mode. Compiler optimizations mask paths; when converting $x$ to floating-point, one optimization tests if $x$ is 0 using an integer comparison to avoid the floating-point instruction, leaving the 0 library path unexplored. Finally, compiled-in assertions, such as checking for bad type tags, often remain uncovered because the code never contradicts the assertion predicate.

### 4.5 Floating-Point SMT Solvers

This section evaluates the accuracy of floating-point solvers with respect to floating-point hardware. A floating-point solver decides the satisfiability of formulas over a theory of floating-point and therefore must at least encode floating-point semantics like those found in soft floating-point libraries. Prior work [114] suggests testing floating-point solvers with randomly generated conformance formulas. Unlike randomized conformance queries, test cases derived from soft floating-point target interesting edge cases defined by the emulation code. Despite the importance of accuracy,
testing a selection of solvers reveals divergent results in every solver. Furthermore, each floating-point solver has implementation quirks which impede testing with all operations and values.

Several freely available contemporary floating-point solvers support a theory of IEEE-754 floating-point. These solvers include mathsat5-2.8 [64], sonolar-2013-05-15 [106], and Z3 [49] (current stable and unstable FPA versions from the git repository). Such specialized floating-point solvers back the only complete symbolic execution alternative to soft floating-point. However, these solvers only occasionally conform to a standard interface, have complicated internals and, despite formal proofs of correctness, are clearly wrong when compared to hardware in many cases.

SMTLIB2-FPA [115] (a proposed standard) conformance tests [114] provide a useful baseline test suite for SMTLIB2-FPA compliant solvers (mathsat and Z3). The conformance tests cover a range of features in SMTLIB2-FPA. These tests exercise the front-end and floating-point theory for a floating-point solver. A solver implements SMTLIB2-FPA in its front-end by translating floating-point arithmetic (FPA) theory primitives into an internal floating-point representation. The tests were generated from the reference implementation with random floating point numbers for a total of 20320 SMTLIB2-FPA queries.

The automatically generated test cases from soft floating-point libraries are novel in that they work as semantically derived tests for third-party floating-point solvers. Each test includes an operation and operands (e.g., \((+ a b)\)) and a scalar result \(r\) computed through hardware. Each solver is tested against the hardware result by checking that the operation feasibly evaluates to \(r\) with a bit-vector equality query as in Figure 4.6. The library tests are based on the smaller fork-inhibited data set from Section 4.4.3 to reduce testing overhead. These tests are simple satisfiability tests on concrete expressions; they neither impose symbolic constraints on operands nor examine counter-examples.

Table 4.6 lists the test results for the floating-point solvers. Even though the tests are shallow, every solver fails some test (fail), indicating actual bugs, or gives no answer after five minutes or refusing certain inputs (unknown). Each row only has a subset of the tests because the front-end and library interfaces lack particular
(set-logic QF_FPA)
(set-info :status sat)
(assert (= (/ roundNearestTiesToEven ; a = 4.96875

; b = 1.9469125e−38

; r = 2.5521178e+38

(check-sat)

; a / b = 2.5521178e+38 is sat, but model claims a/b=plusInfinity

Figure 4.6: A Z3 test case query for checking a single-precision division result

<table>
<thead>
<tr>
<th>Solver</th>
<th>Conformance</th>
<th>Library Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pass</td>
<td>Fail</td>
</tr>
<tr>
<td>mathsat</td>
<td>14487</td>
<td>5833</td>
</tr>
<tr>
<td>sonolar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z3-stable</td>
<td>20023</td>
<td>297</td>
</tr>
<tr>
<td>z3-fpa</td>
<td>20090</td>
<td>230</td>
</tr>
</tbody>
</table>

Table 4.6: Conformance and library test cases applied to several FPA solvers

operations (e.g., ≠). Overall, the rate of failure indicates these solvers are currently more appropriate for domain-specific applications than general program testing.

Each solver has its own quirks. mathsat’s front-end accepts the latest SMTLIB2-FPA proposal but misses a rounding mode in the conformance tests. For concrete tests mathsat is consistent but rejects NaN inputs and often times out on division operations. Sonolar only supports floating-point with library bindings. The Z3 solver accepts obsolete SMTLIB2-FPA and lacks some type conversions. Furthermore, the stable branch of Z3 is nearly a year old; the current unstable Z3 FPA branch is a significant improvement but still diverges from hardware.

4.6 Bugs in Linux programs

The previous sections focus on custom testing with soft floating-point; in this section, the symbolic executor is applied to a large general program set. Test cases were collected from five minutes of symbolic execution on 27795 program binaries from
### Table 4.7: Bugs found in Linux programs following floating-point computation

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide By Zero</td>
<td>26</td>
</tr>
<tr>
<td>Bad Write</td>
<td>57</td>
</tr>
<tr>
<td>Bad Read</td>
<td>259</td>
</tr>
<tr>
<td>Total Programs</td>
<td>314</td>
</tr>
</tbody>
</table>

Ubuntu 13.10 (x86-64) and Fedora 19 (x86) using the SoftFloat library. Floating-point arithmetic appears in hundreds of program faults from these test cases. However, the influence of floating-point operations can be subtle and independent of the crash. Additionally, few paths use the symbolic floating-point capabilities but the overhead from concrete evaluation with soft floating-point is often negligible compared to concrete evaluation using the host’s floating-point instructions.

Table 4.7 presents a summary of flagged floating-point programs. Although 4979 binaries raise some kind of error, only paths covering floating-point instructions on replay (837 test cases) are considered to isolate floating-point tests. Test cases are classified as errors by the type of program fault they cause. The errors are validated with a replay mechanism based on the JIT (568 test cases); fault-free test cases in the JIT replay are ignored. The largest class, bad memory reads, frequently accessed lower addresses which presumably stem from `NULL` pointers.

Floating-point is often independent of the actual bug; programs such as `gifrsize`, `scs2ps`, and `pngcrush` all access `NULL` pointers solely through integer constraints. Regardless, floating-point numbers may subtly influence symbolic integers. The `unicoverage` program (crashes on a buffer overflow) lends an example expression: $100.0 \times \text{nglyphs}/(1+cend-cstart)$. Terms `cend` and `cstart` are symbolic integers (read from `scanf`) and `nglyphs` is a concrete integer. The floating-point 100.0 term coerces the symbolic integer expression into a double precision floating-point value. The floating-point multiplication therefore imposes floating-point constraints (from the i32→f64 operation) on integer-only terms. Curiously, manual inspection of many reports yielded no direct floating-point crashes (e.g., a dereference with a floating-point derived index [57]) but this may be a symptom of briefly symbolic execution time per program.
The majority of floating-point test cases rely solely on concrete floating-point data. Only 94 programs (18%) forked on soft floating-point library code and hence processed any symbolic floating-point values at all. Programs which process only concrete floating-point data incur overhead from dispatching extra instructions for floating-point emulation. The instruction overhead from emulating concrete floating-point with integer code, compared to the default klee concrete floating-point dispatch, is negligible. Soft floating-point tests incur 242058 extra instructions on average (1.04× overhead) with a 135474 instruction standard deviation (1.61× overhead) and 613 instruction median (1.0007× overhead). Floating-point heavy programs skew the average: a program for processing triangulated meshes, admesh, suffered the maximum instruction overhead of 6.98×, followed by bristol, an audio synthesizer emulator, with 2.68× overhead.

4.7 Conclusion

The best software analysis tools must soundly model floating-point. Floating-point as a runtime library is perhaps the simplest worthwhile way to model high-quality floating-point semantics in a symbolic binary executor. This quality comes from soft floating-point being testable within reason through a combination of symbolic execution and cross-checking against hardware. Integer-only symbolic execution produces concrete test files with floating-point information which can be directly confirmed by hardware. If important tests are missed from underspecification, they may be found by testing with multiple floating-point libraries. Finally, although the library testing is incomplete in some cases, the results have demonstrated that a symbolic soft floating-point unit is sufficient for finding many verifiable test cases for bugs in commodity binary programs.
Chapter 5

Expression Optimization from Program Paths

5.1 Introduction

Shifting from specialized symbolic interpretation to general binary translation imposes a performance challenge. First, compilers target physical machine microarchitectures. Fast code on hardware may be slow to interpret due to expensive constraint solver queries. Next, resource mismatches from translation incur some overhead. For instance, register access may map to an intermediate value access plus a read into memory emulating a register file. Furthermore, type information, control structures, and other metadata from source code, which is useful for inferring execution traits, is often unavailable at the binary level. Finally, simple ad-hoc tuning fails to scale to the variety of programs, compilers, and optimization configurations in the wild.

In practice, solver requests dominate symbolic execution running time. Figure 5.1 illustrates query overhead; of several hundred programs after running five minutes, 80% spend more time solving for satisfiability than dispatching instructions. Hence, reducing or eliminating queries can be expected to yield gains.

There is ample opportunity to optimize queries. Based on Figure 5.2, the total number of external calls made to the solver for the same programs from Figure 5.1, a typical symbolic execution session may submit thousands of queries to the solver.
These solver calls primarily determine branch satisfiability for symbolic conditionals (e.g., whether a state should fork) and are inherent to symbolic execution.

The expressions which define queries often have redundancies or unnecessarily verbose terms suitable for optimizations. As an example, a loop that increments an expression $x$ every iteration might produce the bulky expression $(+ ... (+ (+ x 1) 1) ... 1)$ which should fold into a svelte $(+ x c)$. Binary code worsens the problem because translation artifacts leads to more operations and hence larger expressions.

We present an expression optimizer which learns reduction rules across symbolic evaluation of large program sets. During the learning phase, expressions produced at run time are keyed by a hash of samples and stored to a global database. Candidate rules are constructed by matching the sample hash against the hashes of shorter expressions in the database. Each candidate rule is verified using the theorem prover, applied in the interpreter, and saved for future use. Rules are further processed into generalized rules suitable for matching larger classes of expressions.

We implement and evaluate the expression optimizer in klee-mc. Although this system uses binary symbolic execution, the optimizer will work for a traditional source-based symbolic execution system provided such a system scales to support large and diverse programs sets. The optimizer is evaluated at scale with over two thousand commodity programs from a stock desktop Linux system. Rules collected from the programs during a brief learning period are shown to lessen the total number of queries dispatched during symbolic execution by 18% on average and improve
\[ \langle expr \rangle ::= \langle \text{bit-vec-num} \rangle \\
| (\langle \text{unary-op} \rangle \langle expr \rangle) \\
| (\langle \text{binary-op} \rangle \langle expr \rangle \langle expr \rangle) \\
| (\langle \text{extend-op} \rangle \langle expr \rangle \langle num \rangle) \\
| (\text{Select} \langle expr \rangle \langle expr \rangle \langle expr \rangle) \\
| (\text{Extract} \langle num \rangle \langle num \rangle \langle expr \rangle) \\
| (\text{Read} \langle array \rangle \langle expr \rangle) \\
| \langle \text{bit-vec-num} \rangle \langle num \rangle ::= [0-9]+ \]

\[ \langle \text{bit-vec-num} \rangle ::= \langle \text{num} \rangle \langle \text{w} \rangle \langle \text{num} \rangle \]

\[ \langle \text{array} \rangle ::= [a-z]\{a-z0-9\}* \mid (\text{Store} \langle array \rangle \langle expr \rangle \langle expr \rangle) \]

\[ \langle \text{unary-op} \rangle ::= \text{Not} \mid \text{Neg} \mid \text{NotOptimized} \]

\[ \langle \text{binary-op} \rangle ::= (\langle \text{arith-op} \rangle \mid \langle \text{bit-op} \rangle \mid \langle \text{compare-op} \rangle \mid \text{Concat} \]

\[ \langle \text{arith-op} \rangle ::= \text{Add} \mid \text{Sub} \mid \text{Mul} \mid \text{UDiv} \mid \text{SDiv} \mid \text{URem} \mid \text{SRem} \]

\[ \langle \text{bit-op} \rangle ::= \text{Shl} \mid \text{LShr} \mid \text{AShr} \mid \text{Or} \mid \text{And} \mid \text{Xor} \]

\[ \langle \text{extend-op} \rangle ::= \text{ZExt} \mid \text{SExt} \]

\[ \langle \text{compare-op} \rangle ::= \text{Eq} \mid \text{Ne} \mid \text{Ult} \mid \text{Ule} \mid \text{Ugt} \mid \text{Uge} \mid \text{Slt} \mid \text{Sle} \mid \text{Sgt} \mid \text{Sge} \]

Figure 5.3: klee Expression Language Grammar

running and solver times by 10% on average.

5.2 Symbolic Execution and Expressions

Recall symbolic expressions are a byproduct of symbolic execution. The symbolic executor runs a program by marking inputs as symbolic and evaluating the program’s instructions. This evaluation emits expressions to represent operations on symbolic data.

The symbolic executor manipulates a mix of concrete values and symbolic expression data when evaluating LLVM operations. LLVM operations are those defined by the LLVM IR, such as arithmetic, logical, and memory operations, as well as a
handful of specialized LLVM intrinsics. \texttt{klee} maps the results of these operations to expressions with the grammar in Figure 5.3; this language is isomorphic to a subset of the SMTLIB [11] language used by the constraint solver. When convenient, operators will be written in shorthand notation. For instance, the expression to add two bit-vector expressions $a$ and $b$ is $(+ \ a \ b)$.

Large, redundant expressions are expensive. A large expression slows query serialization to the solver and is costly to evaluate into a constant on variable assignment. Expanded tautologies (e.g., $(x \lor \neg x)$), or expressions that evaluate to one value for all interpretations, pollute solver caches and incur unnecessary calls to the theorem prover. Worse, a large expression may linger as a path constraint, slowing future queries that must use the constraint set.

There are common two strategies for shedding expression bloat: expression optimization and concretization. Expression optimization applies sound identities to reduce an expression to fewer terms. For instance, the bit-wise \texttt{or} expression $(\texttt{or} \ 0 \ x)$ is identical to $x$. The drawback is that vital identities are program-dependent; it is infeasible to encode all useful reductions by hand. Alternatively, concretizing symbolic terms reduces expressions to constants at the cost of completeness; $x$ becomes $\{c\}$ instead of $x \in \mathbb{N}$.

The original \texttt{klee} interpreter uses both expression optimization and concretization. A hand-written optimizer folds constants and reduces excess terms from dozens of common redundancies. For concretization, path constraints are inspected through implied value concretization (IVC) to find and concretize variables which are constant for all valid interpretations. For instance, under IVC the constraint $(= \ x \ 1)$ replaces $x$ with 1 in the program state.

We consider the problem of expression node minimization. Given an expression $e$, we wish to find a semantically equivalent expression $e'$ with the fewest possible terms. Formally, we define expression node minimization as follows: given an expression $e$ the minimized expression $e'$ is such that for all tautological expressions $(= \ e \ e_i)$, the number of nodes in $e'$, $|e'|$, satisfies $|e'| \leq |e_i|$. It is worth noting $e'$ is not unique under this definition. To ensure $e'$ is unique, first define an ordering operator on expressions $\leq$ where $e_i \leq e_j$ when there are fewer nodes, $|e_i| < |e_j|$, or by lexical
comparison, $|e_i| = |e_j| \land \text{lex}(e_i) \leq \text{lex}(e_j)$. Uniqueness is then given by the expression minimization problem on $e$ when finding $e'$ where $e' \leq e_i$ for all $e_i$.

A node minimizing optimizer has several advantages over non-minimizing optimization. Theoretically, it is bounded when greedy; optimization stops once the expression stops shrinking. If a conditional expression reduces to a constant, a solver call may be avoided. Contingent conditionals also benefit, such as through better independence analysis since fewer terms must be analyzed. Furthermore, smaller expressions improve the likelihood that IVC will discover concretizing implications because the expressions are simpler. On the other hand, smaller expressions may lead to slower queries because some operations (e.g., integer division) are slow; however, we observe performance improvements from our implementation in Section 5.7.

### 5.3 Rules from Programs

Expression minimization forms a reduction relation, $\rightarrow$. Reductions are central to the theory of contractions, a classic formalization for converting lambda calculus terms [36]. This theory was further developed in abstract rewriting systems as reduction relations under the notion of confluence [70]. We use reduction relations as a framework for reasoning about properties of the expression rewriting system.

The problem of discovering elements of $\rightarrow$ is handled with a database of reduction rule targets (“reducts”). The database, referred to as the EquivDB, is globally populated by expressions made during symbolic execution of binary programs. As the executor creates expressions, they are matched against the EquivDB with assistance from a theorem prover to find smaller, but semantically equivalent, expressions. If equivalent, the expression reduces to the smaller expression and is related under $\rightarrow$.

Information about $\rightarrow$ is maintained as a set of rewrite rules. Each rule has a from-pattern and a to-pattern which describe classes of elements in $\rightarrow$ through expression templates. Once shown to be sound by the theorem prover, rules are used as the expression optimization directive format in the symbolic interpreter.
5.3.1 Reductions on Expressions

A reduction from one expression to another is cast in terms of reduction relations. A reduction relation describes sound translation of $\lambda$-terms to other $\lambda$-terms. We omit the trivial proof of the existence of the correspondence from expressions to $\lambda$-terms, $\Lambda_E : \text{Expr} \to \Lambda$, but note all expressions for our purposes are in the set of closed sentences $\Lambda^0$. All variables are bound; arguments to $\Lambda_E(e)$ are De Bruijn indexes of 8-bit symbolic array read operations (select in SMTLIB) from $e$.

The reduction relation $\rightarrow$ is defined as the binary relation

$$\rightarrow = \{ (\Lambda_E(e), \Lambda_E(e')) \mid (e, e') \in \text{Expr}^2, e' \leq e \land (= e' e) \}$$

The optimal reduction relation $\rightarrow$, the subset of $\rightarrow$ containing only optimally minimizing reductions, is defined as

$$\rightarrow = \{ (e, e') \in \rightarrow \mid \forall (e, e'') \in \rightarrow. \Lambda^{-1}_E(e') \leq \Lambda^{-1}_E(e'') \}$$

An expression is reduced by $\rightarrow$ through $\beta$-reduction. The reduction $a \rightarrow b$ is said to reduce the expression $e$ when there exists an index assignment $\sigma$ for $\Lambda_E(e)$ where $\Lambda_E(e)_\sigma$ is syntactically equal to $a$. $\beta$-reducing $b$ with the terms in $e$ substituted by $\sigma$ on matching variable indices yields the shorter expression $[a \rightarrow b][e]$. The new $[a \rightarrow b][e]$ is guaranteed by referential transparency to be semantically equivalent to $e$ and can safely substitute occurrences of $e$.

As an example of a reduction in action, consider the following 8-bit SMTLIB expressions $e$ and $e'$ that were observed in practice. Both expressions return the value 127 when the contents of the first element in a symbolic array is zero or the value 0 otherwise:

$$e = (\text{bvand } \text{bv128}[8] (\text{sign\_extend}[7] (= \text{bv0}[8] (\text{select } a \text{bv0}[32])))$$

$$e' = (\text{concat } (= \text{bv0}[8] (\text{select } b \text{bv0}[32])) \text{bv0}[7])$$
Expressions $e$ and $e'$ are semantically equivalent following $\alpha$-conversion to the variables. Applying $\Lambda_E$ yields the De Bruijn notation $\lambda$-terms,

$$\Lambda_E(e) = (\lambda x_1. (\text{and} \ 128 \ (\text{sgnext7} \ (= \ 0 \ x_1)))$$

$$\Lambda_E(e') = (\lambda x_1. (\text{concat} \ (= \ 0 \ x_1) \ 0_\tau))$$

Any expression syntactically equivalent to $e$ up to the variable select term is reducible by $\Lambda_E(e) \rightarrow \Lambda_E(e')$. For instance, suppose the variable term were replaced with $(\ast \ 3 \ (\text{select} \ c \ 1))$. Applying the reduction rule with a $\beta$-reduction replaces the variable with the new term,

$$\Lambda_E(e')(\ast \ 3 \ (\text{select} \ c \ 1)) \rightarrow_\beta (\text{concat} \ (= \ 0 \ (\ast \ 3 \ (\text{select} \ c \ 1))) \ 0_\tau)$$

Finally, the new $\lambda$-term becomes an expression for symbolic interpretation,

$$(\text{concat} \ (= \ \text{bv0}[8] \ (\text{bvmul} \ \text{bv3}[8] \ (\text{select} \ c \ \text{bv1}[32])) \ \text{bv0}[7])$$

5.3.2 EquivDB

![Figure 5.4: Storing and checking an expression against the EquivDB](image)

Elements of $\rightarrow$ are discovered by observing expressions made during symbolic execution. Each expression is stored to a file in a directory tree, the EquivDB, to facilitate a fast semantic lookup of expression history across programs. The stored expressions are shorter candidate reducts. The expression and reduct are checked for semantic equivalence, then saved as a legal reduction rule.
Generating Candidate Reducts

The expressions generated by programs are clues for reduction candidates. The intuition is several programs may share local behavior once a constant specialization triggers a compiler optimization. Following a path in symbolic execution reintroduces specializations on general code; terms in expressions are either implicitly concretized by path constraints or masked by the particular operation sequence from the path. These (larger) expressions from paths then eventually match (shorter) expressions generated by shorter instruction sequences.

In the rule learning phase, candidate reducts are collected by the interpreter’s expression builder and submitted to the EquivDB. Only top-level expressions are considered to avoid excess overhead from intermediate expressions which are generated by optimization rewrites during construction. To store an expression into the EquivDB, it is sampled, the values are hashed, and is written to the file path `<bit-width>/<number of nodes>/<sample hash>`. Reduct entries are capped at 64 nodes maximum to avoid excessive space utilization; Section 5.3.3 addresses how reducts can exceed this limit to reduce large expressions through pattern matching.

Samples from expressions are computed by assigning constant values to all array `select` accesses. The set of array assignments include all 8-bit values (e.g., for 1, all symbolic bytes are set to 1), non-zero values strided by up to 17 bytes (i.e., $> 2 \times$ the 64-bit architecture word width to reduce aliasing), and zero strings strided by up to 17 bytes. The expression is evaluated for each array assignment and the sequence of samples is combined with a fast hashing algorithm [2]. It is worth noting this has poor collision properties; for instance, the 32-bit comparisons ($= x \text{12345678}$) and ($= x \text{12345679}$) would have the same sample hashes because neither constant appears in the assignment set. Presumably, more samples would improve hash hit rates at the expense of additional computation.

The EquivDB storage and lookup facility is illustrated by Figure 5.4. At the top of the diagram, an expression from the interpreter is sampled with a set of assignments and the values are hashed. The expression is looked up by the sample hash in the EquivDB and saved for future reference. The match from the look up is checked against the starting expression for semantic equality. Finally, the equality is found
CHAPTER 5. EXPRESSION OPTIMIZATION FROM PROGRAM PATHS

:extrafuns ((x Array[32:8]))
:formula
(= (ite (= bv0xffffffff00[40] (extract [63:24]
 (bvadd bv0xffffffff00000001[64]
 (zero_extend[56] (select x bv0[32])))))
 bv1[1] bv0[1]) bv0[1])

Figure 5.5: An example translation verification query

valid and the corresponding rule is stored into the rule set.

Reduction by Candidates

Before storing an expression $e$ to the EquivDB, the learning phase attempts to construct a reduction rule. Based on $e$’s sample hash, the EquivDB is scanned for matching hashes with the same bit-width and fewer terms. Expression equivalence is checked using the theorem prover and, if valid and contracting, a rule is saved and applied to the running program.

The learning phase loads a smaller expression from the EquivDB based on matching sample hash and bit-width. Loading a candidate reduct $e^*$ parses an SMT file from the EquivDB into an expression with temporary arrays for each symbolic read. The temporary arrays in $e^*$ are replaced by index in $\Lambda_E(e^*)$ with the matching index terms from $\Lambda_E(e)$ to get $e'$. If the reduct is contracting, $e' < e$, and $(= e e')$ is valid by the solver, then $(e, e') \in \rightarrow$ and $e \rightarrow e'$ is saved as a rule for future reference. If $e \rightarrow e'$ is valid, the shorter $e'$ is produced instead of $e$.

An example query for a candidate rule validity check is given in Figure 5.5. An equality query on an arithmetic expression $(e)$ and a constant $(e')$ is sent to the solver and a “sat” or “unsat” string is returned; the from-expression $extract(...)$ is compared against the to-expression $0xffffffff00_{40}$. To determine soundness of the rule $e \rightarrow e'$ with one query, the equality is negated so validity follows from unsatisfiability. Negation of the example translation equality is unsatisfiable, hence the translation is valid.
Care is taken to handle several edge cases. Expressions $e$ often reduce to a constant, but storing and accessing constant values from the EquivDB is needless overhead. Instead, the sample hash predicts $e$ is constant by observing unchanging sample values $c$; the constant $c$ serves as a candidate reduct $e \rightarrow c$ before falling back to the EquivDB. Whenever the solver code builds an expression, to avoid infinite recursion from equivalence checks, that expression is queued for rule checking until the interpreter dispatches the next program instruction. For reliability, if the solver fails or takes too long to check a rule’s validity, then the rule is dropped and $e$ is returned for symbolic interpretation.

### 5.3.3 Rewrite Rules

Rewrite rules represent sets of reductions in $\rightarrow$. Every rewrite rule traces back to a sound primordial expression equality originating from the EquivDB. Each rewrite rule covers a class of reductions with expression template patterns that match and materialize classes of expressions. These rules direct expression optimization and are managed as a rule set kept on persistent storage.

Every candidate rule built during the learning phase is verified with the solver. The equivalent expressions $e$, from the interpreter, and $e'$, from the EquivDB, are converted into a from-pattern $a$ and to-pattern $b$ which are combined to make a rule $a \rightarrow b$. Once $a \rightarrow b$ is verified, all future applications of the rule $a \rightarrow b$ are conducted without invoking the solver.

A rule pattern is a flattened expression with extra support for replacement slots. Figure 5.6 lists the grammar for patterns and rules in general. Symbolic array reads are labeled as 8-bit slots ($\langle label \rangle$) which correspond to the variables from the expression transform $\Lambda_E$. These slots are used in pattern matching which is described in Section 5.4 and establish variable correspondence between the from-pattern and to-pattern. Dummy variables ($\langle dummy-var \rangle$), which are only matched on expression width and ignore structure, replace useless subexpressions through subtree elimination (Section 5.6.1). Likewise, there are constrained slots ($\langle cconst \rangle$) for constants, introduced through constant relaxation (Section 5.6.2), which match when a constant
Figure 5.6: Expression reduction rule grammar
satisfies constraints bundled with the rule \((\langle \text{const-constraints} \rangle)\).

Rules are written to persistent storage in a binary format, serialized into files, and read in by the interpreter. Serialization flattens expressions into patterns by a pre-order traversal of all nodes. On storage, each rule has a header which lets the rule loader gracefully recover from corrupted rules, specify version features, and ignore deactivated tombstone rules.

Manipulating rules, such as for generalization or other analysis, often requires materialization of patterns. A pattern, which represents a class of expressions, is materialized by building an expression belonging to the class. Rules can be materialized into a validity check or by individual pattern into expressions. The validity check is a query which may be sent to the solver to verify that the relation \(a \rightarrow b\) holds. Each materialized expression uses independent temporary arrays for symbolic data to avoid assuming properties from state constraint sets.

### 5.4 Rule-Directed Optimizer

The optimizer applies rules to a target expression to produce smaller, equivalent expressions. A rule-directed expression builder loads a set of reduction rules from persistent storage at interpreter initialization. The expression builder applies reduction rules in two phases. First, efficient pattern matching searches the rule set for a rule with a from-pattern that matches the target expression. If there is such a rule, the target expression \(\beta\)-reduces on the rule’s to-pattern to make a smaller expression which is equivalent to the target.

#### 5.4.1 Pattern Matching

Over the length of a program path, a collection of rules is applied to every observed expression. The optimizer analyzes every expression seen by the interpreter, so finding a rule must be fast and never call out to the solver. Furthermore, thousands of rules may be active at any time, so matching rules must be efficient.

The optimizer has three ways to find a rule \(r\) which reduces an expression \(e\). The
simplest, linear scan, matches \( e \) against one rule at a time until reaching \( r \). The next method hashes \( e \) ignoring constants and \texttt{selects} (skeletal hashing) then matches some \( r \) with the same hash for its from-expression. Flexible matching on the entire rule set, which includes subexpression replacement, is handled with a backtracking trie traversed in step with \( e \). Both skeletal hashing and the trie are used by default to mitigate unintended rule shadowing.

**Linear Scan.** The expression and from-pattern are scanned and pre-order traversed with lexical tokens checked for equality. Every pattern variable token assigns its label to the current subexpression and skips its children. If a label has already been assigned, the present subexpression is checked for syntactic equivalence to the labeled subexpression. If distinct, the variable assignment is inconsistent and the rule is rejected. All rules must match through linear scan; it is always applied after rule lookup to double-check the result.

**Skeletal hashing.** Expressions and from-patterns are skeletal hashed [46] by ignoring \texttt{selects} and constants. A rule is chosen from the set by the target expression’s skeletal hash. The hash is invariant with respect to array indexes and is imprecise; a hash matched rule will not necessarily reduce the expression. Lookup is kept sound by checking a potential match by linear scanning.

**Backtracking Trie.** A trie stores the tokenization of every from-pattern. The target expression is scanned with the trie matching the traversal. As nodes are matched to pattern tokens, subexpressions are collected to label the symbolic read slots. Choices between labeling or following subexpressions are tracked with a stack for backtracking on match failure. On average, an expression is scanned about 1.1 times per lookup, so the cost of backtracking is negligible.

Many expressions never match a rule because they are already optimal or there is no known optimization. Since few unique expressions match on the rule set relative to all expressions built by at runtime, rejected expressions are fast-pathed to avoid unnecessary lookups. Constants are the most common type expression and are obviously optimal; they are ignored by the optimizer. Misses are memoized; each non-constant expression is hashed and only processed if no expression with that hash failed to match a rule.
5.4.2 $\beta$-reduction

Given a rule $a \rightarrow b$ which reduces expression $e$, a $\beta$-reduction contracts $e$ to the $b$ pattern structure. Subexpressions labeled by $a$ on the linear scan of $e$ serve as the variable index and term for substitution in $b$. There may be more labels in $a$ than variables in $b$; superfluous labels are useless terms which do not affect the value of $a$. On the other hand, more variables in $b$ than labels in $a$ indicates an inconsistent rule. To get the $\beta$-reduced, contracted expression, the $b$ pattern is materialized and its selects on temporary arrays are substituted by label with subexpressions from $e$.

5.5 Building Rule Sets

Rules are organized by program into rule set files for offline refinement. Rule set files are processed by kopt, an independent program which uses expression and solver infrastructure from the interpreter. The kopt program checks rules for integrity and builds new rules by reapplying the rule set to materializations.

There is no guarantee the optimizer code is perfect; it is important to have multiple checks for rule integrity and correctness. Without integrity, the expression optimizer could be directed by a faulty rule to corrupt the symbolic computation. Worse, if a bogus rule is used to make more rules, such as by transitive closure, the error propagates, poisoning the entire rule set. All integrity checks are constraint satisfaction queries that are verified by the solver. As already discussed, rules are checked for correctness in the building phase. Rules are checked as a complete set, to ensure the rules are applied correctly under composition. Finally, rules may be checked at runtime when building new expressions in case a rule failure can only be triggered by a certain program.

Additional processing refines a rule set’s translations when building expressions. When rules are applied in aggregate, rather than in isolation, one rule may cause another rule’s materialization to disagree its pattern; this introduces new structures unrecognized by the rule set. These new structures are recognized by creating new
rules to transitive \( \Rightarrow \) the rule set. Further, to-patterns are normalized to improve rule set matching by canonicalizing production templates.

### 5.5.1 Integrity

Rules are only applied to a program after the solver verifies they are correct. Output from the learning phase is marked as pending and is verified by the solver independently. Rules are further refined past the pending stage into new rules which are checked as well. At program runtime the rule set translations can be cross-checked against the baseline builder for testing composition.

Rule sets are processed for correctness. \texttt{kopt} loads a rule set and materializes each rule into an equivalence query. The external theorem prover verifies the equivalence is valid. Syntactic tests follow; components of the rule are constructed and analyzed. If the rule does not reduce its from-pattern when materialized through the optimizer, the rule is ineffective and thrown away.

A rule must be contracting to make forward progress. When expressions making up a rule are heavily processed, such as serialization to and from SMT or rebuilding with several rules, the to-expression may eventually have more nodes than the from-expression. In this case, although the rule is valid, it is non-contracting and therefore removed. However, the rule can be recovered by swapping the patterns and checking validity, which is similar to the Knuth-Bendix algorithm [78].

As an end-to-end check, rule integrity is optionally verified at runtime for a program under the symbolic interpreter. The rule-directed expression builder is \textit{cross-checked} against the default expression builder. Whenever a new expression is created from an operator \( \circ \) and arguments \( \overline{x} \), the expression \( (\circ \overline{x}) \) is built under both builders for \( e \) and \( e' \) respectively. If \( (\overline{=} \ e \ e') \) is not valid according to the solver, then one builder is wrong and the symbolic state is terminated with an error and expression debugging information. Cross-checking also works with a fuzzer to build random expressions which trigger broken translations. This testing proved useful when developing the \( \beta \)-reduction portion of the system; in most cases it is the last resort option since rule equivalence queries tend to catch problems at the \texttt{kopt} stage.
5.5.2 Transitive Closure

Rules for large expressions may be masked by rules from smaller expressions. Once rules are applied to a program’s expressions, updated rules may be necessary to optimize the new term arrangement. Fortunately, rules are contracting, and therefore expression size monotonically decreases; generating more rules through transitivity converges to a minima.

An example of how bottom-up building masks rules: consider the rules \( r_1 = [(+ \ a \ b) \rightarrow \ 1] \) and \( r_2 = [a \rightarrow c] \). Expressions are built bottom-up, so \( a \) in \( (+ \ a \ b) \) reduces to \( c \) by \( r_2 \), yielding \( (+ \ c \ b) \). Rule \( r_1 \) no longer applies since \( r_2 \) eagerly rewrote a subexpression. However, all rules are contracting, so \( |(+ \ c \ b)| < |(+ \ a \ b)| \). Hence, new rules may be generated by applying known rules, then added to the system with the expectation of convergence to a fixed point.

A seemingly optimal solution is to embed rules within rules. For instance \( (+ \ a \ b) \) could be rewritten as \( (+ \ D \ b) \) where \( D \) is an equivalence class that corresponds to both \( a \) and \( c \). However, \( a \) and \( c \) may be syntactically different, and a rule can only match one parse at a time. Embedding rules would needlessly complicate the pattern matcher because simply adding a new rule already suffices to handle both \( a \) and \( c \).

New rules inline new patterns as they are observed. For every instance of pattern materialization not matching the pattern itself (as above), a new rule is created from the new from-pattern materialization. Following the example, the rule \( r_1 \) must now match \( (+ \ c \ b) \), so define a new rule \( r_3 = [(+ \ c \ b) \rightarrow 1] \).

The EquivDB influences the convergence rate. The database may hold inferior translations which bubble up into learned rules. Since smaller expressions store to the database as the rules improve, the database conveniently improves along with the rules. Hence, a database of rule derived expressions continues to have good reductions even after discarding the initial rule set.

5.5.3 Normal Form Canonicalization

Expressions of same size may take different forms. Consider, \((= \ 0 \ a)\) and \((= \ 0 \ b)\) where \( a = a_1 \ldots a_n \) and \( b = a_n \ldots a_1 \). Both are equivalent and have the same number
of nodes but will not be reducible under the same rule because of syntactic mismatch. Instead, a normal form condition is imposed by selecting for the minimum of the expression ordering operator $\leq$ on semantic partitions on to-patterns. With normal forms, fewer rules are necessary because semantically equivalent to-patterns must materialize to one minimal syntactic representation.

The to-pattern materializations are collected from the rule set and partitioned by sample hashes. Each partition $P$ of to-expressions is further divided by semantic equivalence by choosing the minimum expression $e_\perp \in P$, querying for valid equality over every pair $(e_\perp, e)$ where $e \in P$. If the pair is equivalent, the expression $e$ is added to the semantic partition $P(e_\perp)$. Once $P(e_\perp)$ is built, a new $e'_\perp$ is chosen from $P \setminus P(e_\perp)$ and the process is repeated until $P$ is exhausted.

Normal forms replace noncanonical rule to-patterns. After partitioning the to-expressions, the rule set is scanned for noncanonical rules with to-expressions $e$ where there is some $P(e_\perp)$ with $e \in P(e_\perp)$ where $e_\perp \neq e$. Every noncanonical rule to-pattern $e$ is replaced with $e_\perp$ and outdated rules are removed from the rule set file.

### 5.6 Rule Generalizations

The class of expressions a rule matches may be extended by selectively relaxing from-pattern terms. The process of generalization goes beyond transitive closure by inserting new variables into expressions. Useless subterms are relaxed with dummy variables, which match and discard any term, by applying subtree elimination to find terms that have no effect on expression values. Constants with a set of equisatisfiable values are relaxed by assigning constraints to a constant label.

#### 5.6.1 Subtree Elimination

Subtree elimination marks useless from-expression terms as dummy variables in the from-pattern. A rule’s from-expression $e$ has its subexpressions post-order replaced with dummy unconstrained variables. For each new expression $e'$, the solver finds for the validity of ($= e e'$). If $e'$ is equivalent, the rule’s from-pattern is rewritten with
$e'$ so that it has the dummy variable.

For example, let $e = (\uparrow 0 \oplus 1023 (\text{concat} (\text{select} 0 x) (\text{select} 1 x)))$ be a from-expression. This expression, observed in practice, tests whether a 16-bit value bitwise conjoined with 1023 is equal to 0. Since the $\oplus$ term is always non-zero, the equality never holds, implying $e \rightarrow 0$. A good rule should match similar expressions with any 16-bit subterm rather than the concatenation of two 8-bit reads. Traversal first marks the 0, $\oplus$, and 1023 terms as dummy variables but the solver rejects equivalence. The $\text{concat}$ term, however, may take any value so it is marked as a 16-bit dummy variable $v_{16}$, yielding the pattern $(\uparrow 0 \oplus 1023 v_{16})$, which matches any 16-bit term.

### 5.6.2 Constant Relaxation

A large class of expressions generalize from a single expression by perturbing the constants. In a rule, constant slots serve as constraints on the expression. Consider the 16-bit expression $e$, $(\text{and} 0x8000 (\text{or} 0x7ffe (\text{ite} (x) 0 1)))$. The values of the $\text{ite}$ if-then-else term never set the 15th bit, so $e \rightarrow 0$. By marking 0x8000 as a labeled constant $c$, this reduction generalizes to the rule $(\text{and} 0x8000 (\text{or} c$}
(ite (x) 0 1)) where \( c < 0x8000 \) is the \emph{constant constraint}, which expands the rule’s reach from one to thousands of elements in \( \rightarrow \). We refer to this process of slotting out constants with weaker constraints to match more expressions as \emph{constant relaxation}.

To find candidates for constant relaxation, rules are partitioned by from-pattern expression materialization into constant-free equivalence classes. The constant-free syntactic equivalence between expressions \( e \) and \( e' \) is written as \( e \equiv_c e' \). Let the function \( \alpha_c : \text{Expr} \rightarrow \text{Expr} \) \( \alpha \)-substitute all constants with a fixed sequence of distinct free variables. When the syntactic equivalence \( \alpha_c(e) \equiv \alpha_c(e') \) holds, then constant-free equivalence \( e \equiv_c e' \) follows.

A cumulative distribution of equivalence class sizes in \( \equiv_c \) from hundreds of rules is given in Figure 5.7. Constants in rules are \( \alpha \)-substituted with a dummy variable by bit-width from 64-bit only to all byte multiples. Singleton equivalence classes hold rules that are syntactically unique and therefore likely poor candidates for constant relaxation; there are no structurally similar rules with slightly different constants. In contrast, rules in large classes are syntactically common modulo constants. Aside from admitting more rules total, the distribution is insensitive to constant width past 64-bits; few rules are distinct in \( \equiv_c \) and one large class holds nearly a majority of rules.

Constants are selected from a rule one at a time. The constant term \( t \) is replaced by a unique variable \( c \). The variable \( c \) is subjected to various constraints to find a new rule which matches a set of constants on \( c \). This generalizes the base rule where the implicit constraint is \( (= c t) \).

**Constant Disjunction**

The simplest way to relax a constant is to constrain the constant by all values seen for its position in a class of rules in \( \equiv_c \). A constant is labeled and the constraint is defined as the disjunction of a set of observed values for all similar rules. The resulting rule is a union of observed rules with similar parse trees pivoted on a specific constant slot.

The disjunction is built by greedily augmenting a constant set. The first in the set of values \( S \) is the constant \( c \) from the base rule. A new constant value \( v \) is taken
from the next rule and a query is sent to the solver to check if $v$ can be substituted into the base rule over $c$. If the validity check fails, $v$ is discarded. If $v$ is a valid substitution, it is added to $S$. When all candidate values from the rule equivalence class are exhausted, the constraint on the labeled constant slot $c$ is $\bigvee_{s \in S}(= c s)$.

**Constant Ranges**

Range constraints restrict a constant to a contiguous region of values. The values for the range $[a, b]$ on the constant substitution $x$ are computed through binary search in the solver. The constant $c$ from the base rule is the initial pivot for the search since $c \in [a, b]$. Starting from $c$, one binary search finds $a$ from $[0, c]$ and another finds $b$ from $[c, 2^n - 1]$. The constraint $a \leq x \leq b$ is placed on the new rule and the solver verifies equivalence to the base rule’s from-expression.

**Constant Bitmasks**

A constant in a rule may only depend on a few bits being set or zeroed, leaving all other bits unconstrained. Ranges on constants only support contiguous ranges, so it is necessary to introduce additional constraint analysis. Constant constraints on a constant $x$’s bits are found by creating a mask $m$ and value $c$ which is valid for a predicate of the form $x \& m = c$.

The solver is used to find the mask $m$ bit by bit. Since the base rule is valid, the rule’s constant value $a$ must satisfy $a \& m = c$. Bit $k$ of the mask is computed by solving for the validity of $(= x (a \& 2^k))$ when $x$ is constrained by the base rule. Each set bit $k$ implies bit $k$ of $x$ must match bit $k$ of $a$.

### 5.7 Evaluation

The expression optimizer is evaluated in terms of performance, effects on queries, and system characteristics on two thousand programs. Foremost, rules improve running time and solver performance on average. Total queries are reduced on average from baseline by the optimizer. The space overhead and expression distribution of the
EquivDB illustrate properties of the learning phase. Rule effectiveness is measured by number of rules used and rate of sharing.

### 5.7.1 Implementation

The optimizer system is written for KLEE-MC as a drop-in replacement for the existing optimizing expression builder. Table 5.1 shows the lines of C++ code for major components of the expression handling system. Qualitatively, the code for the new optimizer represents a modest effort compared to the ad-hoc version. The largest component is \texttt{kopt}, the offline rule analysis program, where the cost of complexity is low. The rule builder, which applies the vetted rules as expressions are built inside the interpreter, is primarily focused on the fast matching trie. The EquivDB learning builder uses the least code since creating candidate rules is relatively simple.

### 5.7.2 Test System

**Programs**

All experiments are performed over a set of approximately 2300 programs. The programs are from the system binary directories `/\texttt{usr/},\{\texttt{sbin,bin}\}` of an up-to-date x86-64 Gentoo Linux system. Each program is breakpointed at its entry point and snapshotted. All future accesses to the program reuse the snapshot for reproducibility purposes; every snapshot contains the full process memory image, including linked shared libraries, such as the C library \texttt{glibc}.

Programs are set to run under the symbolic interpreter for at most five minutes. Each program is allocated five minutes on one core of an 8-core desktop chip with

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EquivDB/Learning</td>
<td>645</td>
</tr>
<tr>
<td>Rule Builder</td>
<td>1900</td>
</tr>
<tr>
<td>\texttt{kopt}</td>
<td>2519</td>
</tr>
<tr>
<td>Original Builder (Hand-written)</td>
<td>1315</td>
</tr>
</tbody>
</table>

Table 5.1: Lines of code for expression optimization
16GB of memory. There is minor additional processing and book keeping; overall, one symbolic run of the program set takes slightly more than a day.

**Path Replay**

Two sets of runs are taken as a basis of comparison: one with only the ad-hoc optimizer and the other with the rule-directed optimizer as well. The same paths must be followed to give an accurate comparison between the baseline symbolic interpreter and the optimizer. Since paths are known a priori, persistent query caches are disabled to avoid distorted times.

KLEE-MC supports two kinds of path replay: concrete tests and branch paths. A concrete test is a solution to path constraints which replaces symbolic values with constants to duplicate the path. A branch path is a list of branch decisions.

Each branch path is a log of taken branch indexes (e.g., true, false) for some completed state. Branch replay reads an entry from the log for every branch decision and directs the replaying state toward the desired path. As an optimization, if a branch replay forks off a state with a branch log which is a prefix for another branch path, the branch path replay begins at the forked state.

Branch path equivalence is not guaranteed between equivalent paths using different rule. Mismatched branch paths arise between distinct rule sets when the interpreter syntactically checks for constant expressions to avoid extra work; a decision is elided on a constant for one rule set, but recorded for a non-constant on another set, so the logs are no longer synchronized. A concrete test, on the other hand, is a value assignment, and therefore insensitive to expression structure. Concrete tests preserve paths across rule sets so they are used to rebuild branch paths.

### 5.7.3 Performance

The effect of rules on running time and solver time is given sorted in Figure 5.9. Overall, there are significant performance gains made on average with the rule directed optimizer. Additionally, the correlation between run and solver time is evident by solver improvements closely following run time gains.
Figure 5.8: Percentage of queries submitted using rule sets over baseline

Figure 5.9: Percent time for expression optimization over baseline
On average, the optimizer improves performance of the symbolic interpreter on a wide variety of programs. The optimizer improved times by producing shorter expressions and syntactic structures favorable to solver optimizations. The fastest 50th percentile decreases running time by at least 10%. Limited to the fastest 25th percentile, programs see decreased running time of at least 27%.

A few programs do not benefit from the optimizer. Either no improvement or a performance loss were observed in slightly fewer than 13% of all programs. Only five programs (0.2%) took more than $2 \times$ of baseline execution time. There is no requirement, however, to run the optimizer, so applications which exhibit a performance penalty with rules can simply go without and retain baseline speed.

Ultimately, less time is spent in the solver. A 94% majority of the programs spent less time in the solver by using the optimizer. A sizeable 36% of all programs are at least twice as fast in the solver. The 6% minority of programs, like for running time, incurred additional solver overhead. Solver time improvement and running time improvement appear related; only 5% of faster programs had running time decrease more than solver time.

The percent change in queries submitted to the solver is shown ordered in Figure 5.8. On average, the total number of solver queries dispatched for consideration is lower with the optimizer than without. Within the best 50th percentile, at least 17% of queries submitted to the solver were eliminated. In total, fewer queries were dispatched for 87% of the programs. The query histogram in Figure 5.10 illustrates a shift toward faster queries from slower queries.

### 5.7.4 EquivDB

The EquivDB reduct distribution is given in Figure 5.11. Data for the EquivDB was collected from a single run of all programs with a five minute learning period. On the file system, the EquivDB uses approximately 4GB of storage and contains 1.2 million expressions, a modest overhead. Ridges appear at 8 bit multiples, indicating expressions are often byte aligned; possibly because symbolic arrays have byte granularity and most machine instructions are byte-oriented. Some ridges appear to extend past
Figure 5.10: Query time distribution for baseline and rule test cases
the node limit, suggesting the cut-off could be raised. Blank areas, such as those between 32 and 40 bits indicate no entries. As an outlier, there are 63485 expressions with seven nodes at 64 bits.

5.7.5 Rules

A large number of rules suggests the possibility of poor rule efficiency. Highly efficient rules are frequently used whereas low efficiency rules apply only a few times or never at all. If rules are one-off translations with low efficiency, then it is unlikely a fixed rule set will be effective in general. However, as illustrated in Figure 5.13, most programs have fewer than a few thousand rules and fewer than a hundred generalized rules. Rule explosion is from only a few programs interacting poorly with the optimizer.

Code sharing is common through shared libraries, so it is reasonable to expect rules to be shared as well. Figure 5.12 counts the frequency of rules found across programs and confirms sharing. There are 41795 shared rules out of a total of 240052 rules; 17% of rules are shared. The most common rule, with 2035 programs, is an equality lifting rule, \((= \cdot 1_{32} (\text{concat} \cdot 1_{24} \cdot x_8)) \rightarrow (= \cdot 1_{8} \cdot x_8)\), which appears to stem from system call error condition handling code.
Figure 5.12: Program rule sharing.

Figure 5.13: Rules for measured programs.
5.8 Related Work

Peephole optimizers using a SAT solver to find optimal short instruction sequences on machine code are well-known [9, 93]. Expression optimization with reduction rules shares a similar goal by seeking minimal operations. The benefit of applying optimization at the expression level over the instruction level is rules can path-specialize expressions regardless of the underlying code.

For compilers, HOP [46] automatically generated peephole rules by using a formal specification of the target machine. PO [54] simulates runs of register transfers symbolically, using primitive abstract interpretation. It speculatively translates the combined effects back into assembly code. If successful, it replaces the original with the shorter code segment. HOP improved PO by using skeletal hashes to memoize the rewrite rules for a faster peephole optimizer.

STOKE [118] randomly mutates instruction sequences for high-performance code kernels. As in this system, candidate optimizations are sampled as a quick sanity check, then verified using an SMT solver. Like sampling with the EquivDB, the approach is incomplete in that does not guarantee an optimal instruction sequence will be found. However, on account of synthesis and optimization costs, the STOKE system works best when optimizing a small set of important benchmarks rather than optimizing a wide classes of expressions as they appear in a symbolic executor.

Prior work on symbolic execution uses rule-based term rewriting for optimizing solver queries. F-Soft [123] applies a term writing system [40] to expressions generated through symbolic interpretation of C sources. The term rewriting system is seeded with a hundreds of handwritten rules from formal systems (e.g., Presburger arithmetic, equational axiomatization), was applied to a handful of programs, and found improved solver times. However, from the total rules observed in our system and comparatively poor performance of hand-written rules in KLEE, we believe manual rule entry alone is best suited to carefully selected workloads.
Chapter 6

Symbolically Executed Memory Management Unit

6.1 Introduction

Generally symbolic executors explore a program by applying the target’s instructions to a mix of symbolic and concrete data. Symbolically executed instructions manipulate expressions (e.g., 5, \((x + 4)\), \((y + z - 1)\)) as data. These expressions require semantics beyond a traditional concrete execution model. Notably, an executor must process memory accesses on both symbolic and concrete pointers.

A memory access is symbolic if its address expression contains a variable. To illustrate, Figure 6.1 demonstrates some symbolic accesses in the C language. Under symbolic execution, the function \(f\) accesses memory data with two symbolic addresses, \(s[w]\) and \(s[r]\), both of which can resolve to any location in the array \(s\); symbolic values can resolve to multiple values and therefore symbolic pointers can point to multiple addresses. Initially, \(s\) is zeroed by line 1. Assigning 1 to the symbolic \(s[w]\) on line 5 makes all values in \(s\) depend on \(w\) since any element may absorb the symbolic write. The following symbolic read on line 6, \(s[r]\), returns an expression containing the symbolic read index \(r\) and the symbolic write index \(w\). Continuing to build on \(s\) with symbolic accesses creates large expressions which eventually overwhelm the executor’s satisfiability solver.
Chapter 6. Symbolically Executed Memory Management Unit

A large body of work discusses symbolic execution [28, 30, 34, 58, 59, 77, 92, 119, 131] and related memory access techniques. All symbolic executors must handle symbolic accesses; an access policy applies constraints, optimizations, or analysis on the memory access. Such policies are built into the executor and must contend with complicated low-level details. We claim access policy belongs in a symbolically executed runtime library.

This chapter presents a symMMU, a runtime memory access dispatcher for symbolic execution. A symMMU reroutes accesses to a runtime library which evaluates symbolic state in situ. Runtime libraries meet desirable criteria:

- **Variety** of access policies, such as which addresses are accessed and how those accesses are structured, and analysis policies, such as heap violation checking and execution on demand-allocated symbolic buffers.

- **Simplicity** of implementation. Library code is short, high-level policies are free of low-level executor details, and core system changes are non-invasive.

- **Performance** equal to or better than a hard-coded symbolic executor memory access dispatcher.

This system makes several advances over the current state of the art. At a design level, it cleanly separates mechanism (library call-out for memory accesses) and policy (library implementation of access resolution); the symMMU supports six well-known symbolic access policies with few lines of code. To our knowledge the symMMU’s profiler, which symbolically tracks address hits, and heap analysis, which finds over

```c
char s[256] = {0, ..., 0};
char f(void) {
    int r = sym_range(0, 255);
    int w = sym_range(0, 255);
    s[w] = 1;
    return s[r];
}
```

Figure 6.1: Symbolic memory access example code and its intermediate expressions.

<table>
<thead>
<tr>
<th>Line</th>
<th>New Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0, ..., 0)</td>
</tr>
<tr>
<td>5</td>
<td>((if (= w 0) 1 0), ..., (if (= w 255) 1 0))</td>
</tr>
<tr>
<td>6</td>
<td>(read r ((if (= w 0) 1 0), ..., (if (= w 255) 1 0))</td>
</tr>
</tbody>
</table>

...
a thousand heap violations, are the first applications of purely symbolically executed shadow memory. Additionally, it is the first system to test unmodified machine code libraries with demand allocated symbolic buffers, which we use to find functional differences among several distinct libc implementations. Finally, results are tested independently of the symMMU to confirm the symbolic execution agrees with physical hardware; this produces thousands of high-confidence faulting test cases by automatically pruning false positives.

The rest of this chapter is structured as follows. Section 6.2 discusses background for memory accesses and symbolic execution. Section 6.3 describes the symMMU memory access dispatch mechanism in the context of a symbolic executor’s state memory subsystem. Section 6.4 continues by detailing several policies and analysis techniques supported by the symMMU system. Section 6.5 evaluates the implementation of the dispatch and policies with benchmarks and a large set of Linux programs. Finally, Section 6.6 concludes.

6.2 Background

This section provides background information on handling memory accesses, symbolic execution systems, and a formal description of symbolic memory accesses.

6.2.1 Related Work

The symMMU touches a wide range of memory access translation topics. At a low level, address translation in hardware memory management units (MMUs) and operating systems (virtual memory) have software controlled, adjustable policies [3, 53, 128]. For software testing, memory access instrumentation plays an important role in dynamic analysis [100, 101, 117, 136]. All symbolic executors handle symbolic accesses but memory subsystems and analysis approaches differ across systems.

The usefulness of software-defined memory translation has long been acknowledged. RISC architectures, such as MIPS and SPARC, provide mechanisms for software translation lookaside buffer (TLB) handling; accesses fault into software handlers
CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

which map virtual addresses to physical addresses. Software TLBs have a large design space and varying performance characteristics [128]. Operating systems designers also recognize the importance of customization with virtual memory page-faults [3, 53]. The symMMU TLB mechanism (§6.3.2) shares a similar interface to software TLBs.

Many dynamic program analysis algorithms rely on custom memory access handling. Metadata for these algorithms is often stored in shadow memory [99, 137]. This metadata includes lock sets [117], vector clocks [136], heap allocation [100], and information flow [101]. Section 6.4.2 describes a symbolic extension of shadow memory for partial heap checking.

The symMMU supports and extends prior work on symbolic execution. Despite running in a symbolic environment, a natural setting for expressing symbolic policies, symbolically executed runtime code has primarily only modeled specific system environments [86, 116], rather than core system functionality. Meanwhile, the symbolic access policies (§6.4.1) in contemporary symbolic execution systems are quite diverse but also hard-coded. These policies include pointer concretization [58, 92], prioritized concretization [30], symbolic indexing on objects [28, 29, 52], and variable creation [59, 131]. Another policy tags floating-point expressions to report errors on dereference [57]. Other systems lazily initialize or demand allocate accesses on symbolic pointers [75, 110] (§6.4.3). Woodpecker [44] can instrument accesses like a symMMU but only with built-in checkers. Likewise, FIE [45]’s memory smudging and special memory IO regions could be expressed as symMMU policies, but are built into the executor.

6.2.2 Symbolic Execution and Memory Accesses

A dynamic symbolic executor dispatches instructions to advance symbolic program states. Each state has an address space, data memory, and other resources. When an instruction must dereference state memory for a load or store, the executor issues a memory access to a software memory management unit that controls the state’s address space.

A memory access is defined by a 5-tuple \((S, s, w, i, v)\). The access tuple consists
of a state $S$ (including its address space and constraints), a base pointer $s$, the access width $w$ in bytes, the access instruction $i$, and an optional write value $v$. The access width $w$ and instruction $i$ are single-valued concrete data. The write value $v$ and pointer $s$ are expressions which may be symbolic.

Symbolic accesses require formal attention in order to have a precise discussion. Let a symbolic pointer $s$ be an element of the set of pointer-width expressions on bit-vectors. Accessing an address space, a set of disjoint address ranges, through $s$ assumes the constraints of the issuing state $S$. When $s$ is a symbolic pointer, $p \in s[S]$ denotes $p$ is a feasible concrete value for $s$ (i.e., $(s = p)$ is satisfiable given $S$ with $p \in \mathbb{N}$). If $p$ feasibly maps into $S$’s address space, then $p \in S$. For a range of pointers $[p, p + n)$, the subset relation $[p, p + n) \subseteq S$ is shorthand for the existence of a feasible solution for the conjunction $p \in S \land \ldots \land p + n - 1 \in S$.

Any address not mapped in a state $S$’s address space is a bad address for $S$. Typically an access to a bad address is flagged as an error with an accompanying concrete test case. If the test raises a memory access fault under native execution, then the test describes a true bug. For our purposes, we ignore access permissions (e.g., read-only) on particular regions of memory since distinct permissions can be modeled with multiple address spaces used according to the access type $i$.

The executor issues solver queries to determine whether a symbolic access is in $S$. An access $(S, s, w, i, v)$ belongs to exactly one of three categories:

- **Valid.** No faults; $p \in s[S] \implies [p, p + w) \in S$.
- **Invalid.** All faults; $p \in s[S] \implies [p, p + w) \not\subseteq S$.
- **Contingent.** Some faults; $\exists p_1, p_2 \in s[S]$ where $[p_1, p_1 + w) \subseteq S$, $[p_2, p_2 + w) \not\subseteq S$.

Contingent accesses tend to impose the most overhead. A complete symbolic executor finds all permissible accesses and generates at least one test case for a bad access. To find non-faulting accesses, a solver call is submitted for every feasible memory object to check for pointer inclusion. Remaining feasible pointer values imply a bad access.
Figure 6.2: A memory object shared among three states, an object state shared between two states, and an unshared object state.

6.3 A Symbolically Executed MMU

This section outlines the design and implementation of a symMMU for the klee-mc symbolic executor. First, address space and state memory structures are given. Next, the symMMU’s dispatch mechanism and interface are discussed.

6.3.1 Address Space Structures

The symbolic executor stores target program information as a collection of states. A state represents the process image of a partially executed path. The state memory organization is borrowed from klee and is a typical design.

Figure 6.2 illustrates the relationship among states, address spaces, and state data memory. A concrete address space structure maps 64-bit memory addresses to object pairs for each state. Object pairs consist of an object state and a memory object. An object state is a copy-on-write structure that holds the state’s memory data values. Every read and write accesses some object state. A memory object is an immutable structure which tracks the concrete address and length of an object state. The address space manages concrete addresses and lies below the symMMU; symbolic addresses require special policies which issue solver calls.

6.3.2 Soft Handlers

The symbolic executor dispatches memory data accesses through its memory management unit (MMU) subsystem. The original klee interpreter uses a hard-coded MMU that finds access violations, resolves concrete pointers, forks on multi-object
CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

pointers, and issues object-wide symbolically indexed accesses. Symbolic data and queries must be explicitly managed in the executor, however, so augmenting such a policy is a delicate procedure. In contrast, the symMMU in klee-mc bypasses these details by forwarding memory accesses to symbolically executed runtime soft handlers.

SymMMU soft handlers are selected by a command line argument. The executor loads a selected handler (written in C, compiled to LLVM intermediate representation bitcode) from a LLVM bitcode file. A handler translates either symbolic or concrete accesses to direct memory operations on program state; translating both symbolic and concrete accesses with the symMMU uses at least two separate handlers. Multiple handlers can be configurably stacked based on an input file; a handler passes control down the stack by calling executor-controlled function pointers. Example code for a stack of two soft handlers is discussed in Section 6.4.1.

Library Interface

The executor forwards accesses to a library-provided handler through a standardized function interface. Forwarding directs program state to handler bitcode with an implicit function call; the interface is listed in Table 6.1. For a handler \( m \), the interface functions are suffixed with \( _m \) to distinguish among multiple handlers. The handler initializes internal structures with an optional function \( \text{mmu\_init} \) which runs prior to executing the target program. Accesses forward to the \( \text{mmu\_load\_w} \) and \( \text{mmu\_store\_w} \) functions based on the access width \( w \). The handler defines \( \text{mmu\_cleanup} \) to impose additional constraints on a state prior to generating its test case. When marking memory as symbolic, the executor notifies the handler through \( \text{mmu\_signal} \). Special intrinsic functions (Table 6.2) expose executor resources to the handler; each intrinsic makes at most one solver call to minimize time spent in non-preemptible executor code.
Figure 6.3: The symMMU pointer handling path.

### Soft Handlers

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmu_init_m</td>
<td>Initialization.</td>
</tr>
<tr>
<td>mmu_cleanup_m</td>
<td>Add final constraints to test case.</td>
</tr>
<tr>
<td>mmu_load_{8,16,32,64,128}_m(p)</td>
<td>Load from pointer p.</td>
</tr>
<tr>
<td>mmu_store_{8,16,32,64,128}_m(p,v)</td>
<td>Store v into pointer p.</td>
</tr>
<tr>
<td>mmu_signal_m</td>
<td>Signals an extent was made symbolic.</td>
</tr>
</tbody>
</table>

Table 6.1: Trap handler functions for a handler named m with access bit-widths $w = \{8, 16, 32, 64, 128\}$.

### Access Forwarding

Figure 6.3 shows the dispatch process for accessing memory with a symMMU. When the executor evaluates a memory access instruction, it issues an access to the pointer dispatcher. Whether the pointer is a concrete (i.e., numeric) constant or a symbolic expression determines the access path; the executor forwards the state to a handler depending on the instruction and pointer type. Symbolic addresses always forward to a symbolically executed runtime handler. Concrete addresses, if ignored or explicitly masked, follow a built-in fast-path.

### Concrete Addresses

Some memory analysis policies, such as heap access checking, must track accesses on concrete pointers alongside symbolic pointers. However, concrete accesses are necessary to symbolically execute symMMU handler code, so care must be taken to avoid...
CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

### Table 6.2: Runtime primitives for memory access

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>klee_sym_hash(s)</code></td>
<td>Hash expression <code>s</code> to constant.</td>
</tr>
<tr>
<td><code>klee_wide_load_w(s)</code></td>
<td>Load with symbolic index <code>s</code>.</td>
</tr>
<tr>
<td><code>klee_wide_store_w(s,v)</code></td>
<td>Store <code>v</code> to symbolic index <code>s</code>.</td>
</tr>
<tr>
<td><code>klee_enable_softmmu</code></td>
<td>Enable concrete redirection.</td>
</tr>
<tr>
<td><code>klee_tlb_ins(p)</code></td>
<td>Default accesses to object at <code>p</code>.</td>
</tr>
<tr>
<td><code>klee_tlb_inv(p)</code></td>
<td>Drop defaulting on object at <code>p</code>.</td>
</tr>
</tbody>
</table>

Infinite recursion. Concrete translation through the symMMU is temporarily disabled on concrete accesses, reverting to the default built-in concrete resolution, to limit recursion. For each state, a translation enabled bit controls soft handler activation. If the bit is set, concrete accesses forward to the runtime. The translation enabled bit is unset upon entering the handler, forcing fast path translation for subsequent accesses. Prior to returning, the handler re-enables symMMU translation by setting the translation bit with the `klee_enable_softmmu` intrinsic.

**Translation Lookaside Buffer**

Calling handlers for every concrete access is slow. Fortunately, if most concrete accesses are irrelevant to the handler (i.e., processed no differently than the default concrete path), then the symMMU overhead is amortizable. If concrete accesses to an entire memory object can bypass the symMMU, the handler can explicitly insert the object’s range into a software TLB so subsequent accesses follow the concrete fast path for better performance.

A runtime programmed TLB controls concrete fast-path forwarding. The concrete TLB maintains a fixed number of address ranges to pass to the fast path. A concrete handler ignores accesses by registering address ranges with the TLB. The handler reclaims accesses by removing address ranges from the TLB. Each state has its own private TLB because state interleaving interferes with reproducing paths; flushing a global TLB on state reschedule alters the instruction trace past the preemption point from extra TLB misses.
6.4 Access Policies and Analysis

Our symMMU supports a variety of access policies. For symbolic accesses, the choice of policy affects solver overhead, state dilation, and testing completeness. We use the symMMU’s capability to observe all accesses to extend the executor with new modes of analysis on programs. Two new extensions, symbolic access profiling and heap checking, use shadow memory to track metadata for every access. Another extension, unconstrained execution, demand allocates memory buffers by access to dynamically infer argument structure from program function behavior.

6.4.1 Symbolic Access Translation

A symbolic access may resolve to multiple addresses. The choice of access policy influences the performance and completeness of symbolic accesses. We consider two types of policies: partial and full. Partial policies transform pointers and check for properties, but only occasionally dispatch an access and must be stacked. Full policies, situated at the bottom of a handler stack, always dispatch an access.

Excessive Range Checking

A symbolic pointer $s$ potentially generates many states because it may be unbound. Following through on every precise address where an unbound pointer access lands is expensive and usually uninformative; it suffices to know the set of feasible pointers $\{p \mid p \in s[S]\}$ is large. Range checking finds excessively stray accesses; if $s$ feasibly exceeds a sensible distance (we use 256MB) from some $p \in s[S]$, then the access is flagged, otherwise the access continues down the stack.

Figure 6.4 lists a partial symbolic 8-bit load soft-handler, `mmu_load_8_rangechk`, which checks the range of a symbolic pointer $s$. First, the handler issues a solver request in `testptr_invalid` to find a $p \in s[S]$, then builds a symbolic range check for $s$. When $s$ covers an extreme range, the state forks on the `if` into an out-of-range and an in-range state because `testptr_invalid(s)` is feasibly both true and false. The out-of-range state is assumed to fault and is reported. The in-range state proceeds down the stack to the next handler via the function pointer `mo_load_8`. 
CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

```c
MMUOPS_S_EXTERN(rangechk); /* setup stack structures */
#define MAXRANGE ((ptrdiff_t)0x10000000)

static int testptr_invalid(intptr_t s) {
    intptr_t p = klee_get_value(s);
    ptrdiff_t d = s - p;
    return (s<0x1000) | (d > MAXRANGE) | (-d > MAXRANGE);
}

uint8_t mmu_load_8_rangechk(void* s) {
    /* test for excessive range */
    if (testptr_invalid((intptr_t)s))
        klee_uerror("bad_range!", "ptr.err");
    /* proceed down the stack */
    return MMUOPS_S(rangechk).mo_next->mo_load_8(s);
}
```

Figure 6.4: Range checking handler for 8-bit loads.

Constant Symbolic Pointer Resolution

Concrete accesses are cheaper than symbolic. If the state’s path constraints for a symbolic address to have exactly one solution, every future access can be concrete. When \( p_1, p_2 \in s[S] \implies p_1 = p_2 \), constant symbolic pointer resolution replaces every \( s \) with \( p_1 \) and dispatches the concrete access.

Create Variable

Relaxing precise memory content can simplify symbolic access complexity with state overapproximation at the cost of producing false positives. In this case, reading from a symbolic pointer \( s \) returns a fresh symbolic variable \( v \). To reduce overhead, our implementation keeps a mapping from old symbolic pointers to their variables. Since this policy is blatantly unsound (suppose \( \forall p \in s[S].v \neq *p \) ) KLEE-MC never uses it in practice. However, similar strategies appear elsewhere [45, 59, 131], indicating that some consider it a worthwhile policy, so it is included for completeness and to demonstrate the symMMU’s flexibility.
Figure 6.5 lists an example full 8-bit load soft handler which concretizes the access pointer. An 8-bit symbolic load access enters through `mmu_load_8_uniqptr`. A concrete value \( p \in s[S] \) is retrieved with `klee_get_value(s)` (the sole solver query). Next, the handler binds \( p \) to \( s \) with the constraint \((= p s)\) through `klee_assume_eq(s, p)`. Finally, the handler safely dereferences \( p \) (if \( p \) is bad, it is detected on the concrete path) and returns the value to the target program.

### Fork on Address

Forking a state for every \( p \in s[S] \) explores all feasible accesses. Instead of calling `klee_get_value` once (§ 6.4.1), forking on address loops until all feasible addresses are exhausted. Since each feasible address consumes a solver call and a new state, this policy is costly when \( |s[S]| \) is large.

Bounding explored feasible addresses reduces overhead but sacrifices completeness. In order to shed some feasible address, the handler chooses addresses based on desirable runtime or program properties. We implemented two bounded policies in addition to complete forking: one caps the loop to limit state explosion and the other forks on minimum and maximum addresses to probe access boundaries.
Prioritized Concretization

Blindly concretizing a symbolic read access potentially overconstrains the fetched value. Prioritized concretization searches for feasible addresses \( p \in s[S] \) such that the data that \( p \) points to is symbolic. If there is such a \( p \), then \( s \) is concretized to \( p \), and the symbolic value is returned.

Fork on Objects

The default klee policy (and similar policies [30, 52]) forks on feasible memory objects, issuing object-wide accesses over symbolic arrays. When a symbolic access falls within a spatially local region (e.g., a page), the access can be efficiently folded into an expression as in Figure 6.1. Assuming this locality, forking for every feasible memory object with symbolic indexing reduces the total forked states in exchange for larger and potentially more expensive queries.

The fork-on-object policy forks a state for every memory object covered by a symbolic memory access at \( s \). For every memory object, a wide memory access symbolically indexes the object state. Wide write accesses create large expressions containing update lists for the object at the time of write. Most memory objects are page-sized so our implementation uses a faster and simpler forking algorithm than klee’s binary search. Objects are enumerated by forking on unique page pointers \((p \& 4095) \in (s \& 4095)[S]\). If a feasible object is smaller than a page, the handler iterates over objects in order until the next page boundary. Wide accesses on these objects are explicitly constructed; the handler uses klee_wide intrinsics to issue symbolically indexed accesses. Although the system could conceivably symbolically index multiple objects per forked state, a distinct policy in its own right, it is unclear how many objects at once would give the best query performance relative to total states forked.

If-Then-Else Reads

A symbolic indexed read on an object state leads to unnecessary array overhead in the solver when the feasible pointer set size \(|s[S]|\) is much smaller than the object.
If-then-else reads translate an access into an if-then-else expression when the set \( s[S] \) is small. When the number of addresses is bounded by \( n \) for \( p_1, \ldots, p_n \in s[S] \) and \( p_i = p_j \Rightarrow i = j \), the expression \( e_1 \) defined by the recursive relation

\[
e_i = (\text{if } (= s p_i) (*p_i) e_{i+1}), \quad e_n = (*p_n)
\]

is returned by the dereference and the disjunction

\[
(= s p_1) \lor (= s p_2) \lor \ldots \lor (= s p_n)
\]

joins \( S \)'s path constraints when \( |s[S]| > n \) for soundness.

### 6.4.2 Shadowing Memory

A shadow memory analysis algorithm [100, 101, 117, 136] associates metadata with memory addresses; it computes metadata from memory accesses and stores it to a mapping (the shadow memory) of addresses to features. Since the symMMU controls memory accesses, shadow memory is a straightforward extension. We use a shadow memory runtime library for two additions to the executor: memory profiling and heap-checking. The profiler increments a per-word shadow counter for every access. The heap checker tracks program memory allocation status to detect heap violations.

**Runtime Structures**

Figure 6.6 illustrates the symMMU shadow memory structure. Symbolic shadow memory is a map \( S : A \rightarrow V \) from addresses to shadow values. In the figure, a symbolic access on two feasible bytes at \( a \) returns a shadow expression \( S(a) \) representing shadow values for both feasible bytes in a single state. The hashed page number of \( a \) indexes into an array of page buckets. Each bucket holds a list of demand allocated shadow pages corresponding to pages of data memory. Retrieving \( S(a) \) uses a wide access so states only fork per shadow page. Since the shadow memory code is a library, the executor transparently handles all constraints and states.

An alternative shadow memory design [99] uses a two-level page table on 32-bit
addresses. An address indexes the table through two levels of indirection to retrieve a shadow page in constant time. In practice, the symMMU shadow memory buckets remain small so linear time overhead is negligible compared to the extra levels of indirection or auxiliary tables needed for 64-bit addresses. Likewise, memory objects are usually page-sized so wide accesses split into multiple states regardless of the data structure.

**Access Profiling**

An access profiler can help a programmer find memory hot spots or understand cache behavior. We developed a profiler to count symbolic accesses on each address. The profiler keeps shadow counters for memory addresses on every state; every symbolic access increments a set of feasible counters.

Unlike a traditional profiler, symbolic shadow state influences the profiler results. For instance, a complete symbolic byte store on $s[S] = \{p, p+1\}$ touches two memory locations, $p$ and $p+1$. Storing to two feasible memory locations updates two shadow counters within a single state; the shadow counters become symbolic via symbolic write. The profiler is free to choose any feasible counter value. In our implementation, the profiler greedily increases memory coverage by choosing $p \in s[S]$ when $S(p) = 0$ and assuming $S(x) > 0$ on path termination (when feasible) for all addresses $x$. 

Figure 6.6: Retrieving a disjunction of shadow values for a symbolic address.
Profiling also demonstrates the importance of guarding against recursion. A symbolic access leads to a shadow counter update on multiple feasible shadow counters; the shadow counter address is symbolic. Incrementing the counter through a symbolic pointer triggers another profiling event in the software handler causing an infinite loop. To prevent hanging, a flag guards against profiler handler reentrance.

Heap Checking

Heap access violations are common errors in system programs. A program accesses freed or uninitialized heap data with little immediate consequence but eventual state corruption. Dynamic analysis algorithms [66, 100] detect these errors at the point of violation with shadow memory. The symMMU extends the memcheck algorithm [100] to binary symbolic execution with incomplete heap information.

Symbolically processing heap memory accesses exploits pointer reach knowledge lost to concrete test cases. Although symbolically derived tests are compatible with traditional heap violation detection algorithms [59], concrete testing alone leads to false negatives. Consider a read from a buffer \( b \), initialized up to some \( n \), at a symbolic offset \( i \leq n \). For a concrete test, \( i \) may be chosen such that \( b[i] \) is initialized \( (i < n) \), missing the error. Under symbolic analysis \( b[i] \) is both initialized and uninitialized so the error is modeled.

The heap checker models the heap as the program runs. A set of disjoint address ranges (heap blocks) \( [a, a+n) \in \mathcal{H} \) tracks a state’s current heap \( \mathcal{H} \). When a program allocates a heap block \( b \), the checker adds \( b \) to \( \mathcal{H} \). If a pointer \( p \in s[S] \) is in \( \mathcal{H} \), \( \exists b \in \mathcal{H}, p \in b \), then \( s \) is a feasible heap address.

An access is checked by observing the shadow value for each accessed byte address. Each byte is assigned two bits of shadow state representing its heap status (similar to purify [66] and memcheck [100]). Each byte is assigned one of the following shadow values:

- **OK.** Access is safe; may be a heap data.
- **UNINIT.** Heap address with uninitialized data.
- **FREE.** Former heap address.

Figure 6.7 illustrates the finite state machine which manages the heap status.
Initially, accesses default to OK; the tracked heap is set to empty. To avoid overhead from repeatedly calling the symMMU, long, contiguous OK ranges are inserted into the concrete address pass-through TLB. When the program acquires a pointer $p$ through an allocation function (e.g., `malloc(n)`), the heap checker intercepts the call, records the allocated block and marks the shadow memory for $[p, p+n)$ as UNINIT. When $p$ is deallocated with a call `free(p)`, the heap checker retrieves $p$’s length $n$, marks $[p, p+n)$ as FREE, and drops the $[p, p+n)$ record from $\mathcal{H}$.

The heap checker detects three heap violations:

**Dangling Accesses.** An access to freed heap memory is a dangling access. This violates the heap discipline because all heap accesses from client code should reference allocated memory; if a program accesses freed memory, it may corrupt state or retrieve an unexpected value. A handler reports a dangling access whenever an access can feasibly resolve to a FREE shadow value.

**Uninitialized Reads.** The contents of the heap are undefined on allocation. Reads from freshly allocated uninitialized memory produces undefined behavior. If a read access pointer feasibly maps to UNINIT, the heap checker produces a test case for an uninitialized read.

**Double Frees.** Each heap allocation pairs with at most one deallocation. Feasibly deallocating memory twice consecutively causes the heap checker to terminate the state and report a double free. Early termination has a performance benefit by skipping the double-free detection error path in libc.
6.4.3 Unconstrained Pointers

Unconstrained execution [50, 68, 75, 110] tests functions rather than full programs. Instead of trying to reach a specific function through a path from the beginning of a program, the symbolic executor jumps directly to the target function, lazily initializing pointer arguments with symbolic data. These unconstrained pointers initialize on first access, expand as needed, are relocatable, and require no type information. We describe handlers written for the symMMU which manage unconstrained pointers.

An unconstrained pointer is a special symbolic pointer defined by the way it is accessed on a path. Rather than having a concrete address in memory, accessing an unconstrained pointers maps to a demand allocated buffer. All pointers passed to a target function for unconstrained execution are unconstrained. To avoid false positives for memory access faults, the executor optimistically assumes symbolic pointers are valid. Initially accessing an unconstrained symbolic pointer $u$ demand allocates unconstrained memory buffers with symbolic data to back $u$. These buffers have concrete physical length, associated with the allocated backing buffer, as well as a symbolic virtual length, based on precise observed range of access, which facilitates expansion. Subsequent accesses to $u$ retrieve the same data through a consistent translation map. The translation map routes structurally distinct, but semantically nearby, unconstrained pointers to the same buffer. Finally, unconstrained pointers resolve to concrete pointers to generate a test case.

Translation Entries

To make unconstrained accesses optimistic, rather than unbounded and contingent, the symMMU handlers must translate unconstrained pointers to buffers mapped into the state’s address space. Figure 6.8 shows the organization of this translation. From the top, unconstrained pointers map into a translation table by expression hash. The translation table points to translation entries which each describe an unconstrained pointer’s demand allocated memory buffer.

The translation table maps unconstrained pointers to translation entries by address expression hash. A hash function $h : E \rightarrow \{0, 1\}^{64}$ from expressions to 64-bit
values maps a symbolic address $u$ to the table for solverless lookup. Since $u$ is symbolic, it must have some variable subexpression (e.g., $(\text{select } i \ a)$ reads the symbolic array $a$ at index $i$). The hash $h$ function relies on the idea that distinct variable subexpressions imply feasibly distinct pointers. For example, $u = (\text{select } 0 \ x)$ and $u' = (\text{select } 4 \ x)$ give $h(u) \neq h(u')$ because the values are feasibly unrelated. In contrast, $(\text{add } 1 \ (\text{select } 0 \ x))$ and $(\text{add } 2 \ (\text{select } 0 \ x))$ refer to one base symbolic pointer $u$ and therefore hash equally; the difference is a small constant offset. Theoretically this heuristic is unsound in that two distinct variable subexpressions may hash equally, but this is never observed in practice.

A translation entry defines a demand allocated buffer for an unconstrained pointer, maintaining both the buffer bounds and backing memory. The buffer centers around a pivot $u$, the unconstrained pointer of the first access. The entry holds the buffer's concrete pointer $p$ and its current concrete radius $r_c$. For precise bounds, the entry tracks the minimum $u_{\min}$ and maximum $u_{\max}$ access pointers. The difference between $u_{\min}$ and $u_{\max}$ defines a symbolic radius $r$. Taking all entries, the $i$th entry has a pivot $u_i$ and a symbolic radius $r_i$. Pivots are ordered by entry, $u_0 < u_1 < ... < u_k$, when resolving to concrete addresses on path termination.

**Demand Allocation**

Unconstrained demand allocation has two phases. First, an initial access to a distinct unconstrained pointer creates a translation entry and buffer around the pointer. Next,
future accesses near the pointer expand the buffer keeping values and constraints from prior accesses.

**Buffer Initialization.** An access to a new unconstrained pointer $u$ allocates space so future accesses are consistent. First, the soft handler determines $u$ is distinct and distant from previously accessed unconstrained pointers. Next, the handler creates a translation entry pivoted around $u$ and allocates a buffer centered about $u$. Finally, the access is serviced through the buffer and the target program proceeds.

If $u$ has no translation, the runtime allocates an unconstrained buffer. First, a translation entry keyed by $h(u)$ is inserted into the table. A small initial buffer (16 bytes) filled with unconstrained symbolic data is allocated to a fresh translation entry. Subsequent accesses around $u$ route to the buffer through the access handlers.

**Buffer Expansion** Suppose an access to a pointer $u'$ based on an unconstrained pointer $s$ exceeds the bounds of the initially allocated buffer. By the optimistic access policy, the buffer is extended when the distance from the buffer is reasonably short. There are two cases: the fast case $h(u') = h(u)$ and the slow case $h(u') \neq h(u)$. The fast case $h(u') = h(u)$ suffices to explain buffer extension; the $h(u') \neq h(u)$ case is handled in Section 6.4.3.

When $h(u') = h(u)$, the runtime uses the translation entry for $u$. When the distance between $u$ and $u'$ is less than the buffer radius, $|u - u'| < r_c$, there is no need to extend the buffer. Otherwise, the buffer is copied to a new buffer with a radius of $2r_c$; the buffer pointer and $r_c$ are updated in the translation entry to reflect the expansion.

**Aliasing**

Two unconstrained pointers $u_i$ and $u_j$ may map to the same buffer, $|u_i - u_j| < r_i$, but hash unequally, $h(u_i) \neq h(u_j)$. Having distinct translation entries for $u_i$ and $u_j$ potentially assigns multiple values to one address, leading to nonsense paths from an incoherent memory model. This incoherence is caused by aliasing; we outline our method for efficiently resolving aliases.

An access to an unconstrained pointer $u$ with an untranslated hash $h(u)$ undergoes alias resolution before acquiring a translation entry. First, the pointer $u$ is tested for
inclusion in the unconstrained arena \([\text{UC\_MIN}, \text{UC\_MAX}]\) with an and mask; if \(u\) is unconstrained, then \(\text{UC\_MIN} \leq u < \text{UC\_MAX}\) is satisfiable. Next, the runtime tests translation entry inclusion feasibility to resolve the entry for \(u\). Inclusion testing relies on the pivot ordering \(u_0 < ... < u_n\) and entry ordering \([u_i-r_i, u_i+r_i) < [u_j-r_j, u_j+r_j)\) when \(i < j\). Hence, \(k = \arg \min_i (u' \leq u_i + r_i)\) implies \(u' \in [u_k - r_k, u_k + r_k]\). If \(k\) exists, the runtime updates the translation table to point \(h(u)\) to the entry for \(h(u_k)\). Inclusion testing may accrue additional aliases by forking new states where \(u_i - r_i \leq u' < u_i + r_i\) for \(i > k\). If there is no feasible entry, \(h(u_k)\) is assigned its own translation entry.

**Concrete Address Assignment**

A dereferenced unconstrained pointer eventually binds to a concrete address. The runtime postpones address assignment by translating unconstrained pointers to a backing buffer on the fly. To produce a concrete test case, the runtime uses the solver to assign concrete addresses to pointers.

Premature address assignment unnecessarily constrains unconstrained pointers, masking otherwise feasible paths. Consider the \texttt{memcpy} library call which copies values in memory from a pointer \(u_1\) to a non-overlapping location starting at pointer \(u_2\). A naive \texttt{memcpy} loads a byte from \(u_1\), stores it to \(u_2\), then increments each pointer. However, this implementation has needless overhead; multi-byte accesses reduce the total iterations by at least half. Such multi-byte accesses incur additional cost for unaligned addresses. Hence a fast \texttt{memcpy} uses distinct code depending on the alignment (i.e., the lower bits) of \(u_1\) and \(u_2\). Eagerly assigning concrete addresses to \(u_1\) and \(u_2\) drops the unaligned corner cases.

The pivots \(u_0, ..., u_n\) are assigned concrete addresses on path termination. The handlers control assignment by ordering pivots at time of first access and bounding addresses within the unconstrained arena \([\text{UC\_MIN}, \text{UC\_MAX}]\). For the base case \(u_0\), the constraints \(\text{UC\_MIN} \leq u_0 - r_0\) and \(u_0 + r_0 < \text{UC\_MAX}\) hold. In general, inserting the \(k\)th pivot \(u_k\) takes the constraints \(u_{k-1} + r_{k-1} < u_k - r_k\) and \(u_k + r_k < \text{UC\_MAX}\). A cleanup handler minimizes \(r_k\) to keep buffer sizes small.
6.5 Evaluation

This section evaluates the symMMU against the criteria set in Section 6.1. Simplicity of the symMMU is quantified in terms of the implementation’s total lines of code. Dispatch microbenchmarks and a comparison of discovered access faults on Linux programs demonstrate the symMMU’s performance is competitive with a built-in hard-coded MMU. Policy microbenchmarks, heap checking on programs, and unconstrained execution on libraries show the symMMU flexibly supports a useful variety of access and analysis policies.

6.5.1 Implementation Complexity

Figure 6.3 shows the amount of code for each MMU to quantify relative implementation complexity. The line counts were calculated with SLOCcount [132] on the relevant source files. The table is split between executor code (C++ compiled to binary) and runtime code (C compiled to LLVM bitcode). Of note, the baseline KLEE MMU (373 lines) needs slightly more code than the symMMU (340 lines) implementation, suggesting the symMMU is simpler to implement.

6.5.2 Dispatch Mechanism

The design of the symMMU dispatch mechanism strongly influences its performance. We use a microbenchmark to compare the built-in symMMU with instrumentation to show explicit support for a symMMU is competitive with the baseline KLEE MMU and
superior to instruction rewriting. Furthermore, we find the concrete TLB improves symMMU performance on concrete workloads with low overhead.

**Access Instrumentation**

We compared concrete access overhead among the baseline klee MMU, the interpreter symMMU, and an instrumented symMMU to measure the concrete access performance overhead from using a symMMU. The instrumented symMMU replaces all target program loads and stores with runtime function calls which partition accesses as concrete or symbolic and forward to the appropriate handler. Each MMU was benchmarked with `strcpy` on 4096 bytes, repeated 10,000 times in a program. The baseline MMU and built-in symMMU issued the same number of LLVM instructions, since it has a built-in concrete access path, and used equal time. The instrumented symMMU issued $2.3 \times$ as many instructions and took $1.8 \times$ longer to complete; calling a handler on every program memory access is costly. Instrumentation is relatively slower so we only use the built-in symMMU.

**TLB**

To measure the cost of concrete symMMU handling, we compared the performance among the fast path, naïve handling, and TLB assistance using the benchmark from Section 6.5.2. The concrete symMMU policy passes every concrete address to the built-in fast path and the TLB is set to 128 entries. Without TLB support, the benchmark used $1.9 \times$ as many instructions and $1.5 \times$ as much time as the baseline MMU without concrete symMMU support. With TLB support, the benchmark dispatched $1.2 \times$ as many instructions and used $1.1 \times$ as much time as the baseline MMU.

**6.5.3 Policy Microbenchmarks**

A set of access intensive microbenchmarks measure the difference among symMMU policies and the baseline klee MMU. Each microbenchmark is compiled from C and symbolically executes one of 23 standard libc string functions for five minutes. Policy performance is ranked by generated tests over the baseline klee MMU (i.e., test case
speedup); paths naturally vary across policies on account of different code in the runtime handlers.

The benchmarks frequently access symbolic string buffers, stressing the MMU. A symbolic buffer $b$ spans two pages in length (8192 bytes) to highlight forking on the fork by object policy. Two symbolic integers model offsets $\Delta_1$ and $\Delta_2$ into $b$, another symbolic integer models string length (e.g., for strncpy). Two pointers $p_1$ and $p_2$, with $p_k = b + (\Delta_k \% (k \times \text{PAGE\_SIZE}))$, give two in-bound feasibly overlapping pointers which are passed into the string function.

Figure 6.9 shows the test results. A majority of benchmarks generate more tests than baseline (88 more, 48 fewer). Conservatively, most symMMU policies are no worse at completing paths than the baseline klee policy. One exception is ITE (§ 6.4.1) which submits many solver queries for feasible addresses before constructing an ite expression. On the other hand, pointer concretization, which should produce few tests, sometimes outperforms baseline. To explain, the baseline forks all feasible states at once for a symbolic access, issuing many queries. By contrast, the symMMU forks in the runtime so expensive accesses are preempted so that other states may interrupt the memory operation and run. The inconsistent performance among policies and functions demonstrates the importance of policy variety since no one simple policy is best.

6.5.4 Memory Faults in Linux Programs

To judge the effectiveness of the symMMU at scale, we ran 12508 64-bit Linux programs taken from Fedora 20 for five minutes each under the symbolic executor to find memory access violation test cases. These test inputs cause access faults in unmodified third party programs.

Table 6.4 shows the number of programs and test cases found by the symMMU to have memory errors using the klee policy (§ 6.4.1) binned by stack trace. To be sure the results were true positives, the table results were confirmed at the binary translation level through a test replay program based on the LLVM JIT. For comparison, the standard klee MMU flagged 2252 tests (1751 confirmed by JIT replay) whereas
the symMMU flagged 2667 tests (2589 confirmed). Two reasons explain the klee MMU’s extra false positives: first, the klee MMU is poorly tested (a few hundred LLVM bitcode programs), and second, the klee MMU is intricate and difficult to get right. Similarly, the symMMU pass issued 2.1× solver queries over the klee MMU pass; symMMU queries tend to solve faster.

### 6.5.5 Profiler

Figure 6.10 illustrates the symbolic profiler’s cost for a simple benchmark. The benchmark program tests the profiler coverage policy by looping over an array a with n
elements, writing 1 to $a[s[i]]$ where $s$ is an array of symbolics; $s$’s assignment controls which elements of $a$ are updated. The shadow memory’s granularity is configured to have a shadow word for each element in $a$. Complete coverage of $a$ implies $s$ contains all values between 0 and $n – 1$; the profiler imposes these constraints on $s$. For overhead, total solver queries grows linearly with respect to $n$. Time for the benchmark exponentially rises with $n$, peaking at 338 seconds for 1024 elements; at 2048 elements, solver performance completely deteriorates and the benchmark times out after 30 minutes.

### 6.5.6 Heap Violations

The symbolic executor produces a concrete test case for every heap violation it finds. Since a test could be a false positive (e.g., from bugs in the tool), it is important to automatically detect true positives. Unlike access faults, heap violations in general do not trigger specific hardware events. SAGE [59], for instance, has special support to force crashes on heap violations. To confirm errors without the luxury of faults, the symMMU heap checker replay system feeds test cases with system call logs into an unmodified third-party heap analysis tool, valgrind [100] memcheck.

![Figure 6.10: Profiler benchmark performance.](image-url)
 CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

<table>
<thead>
<tr>
<th>Type</th>
<th>Tests</th>
<th>Code Sites</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Free</td>
<td>56</td>
<td>43</td>
<td>33</td>
</tr>
<tr>
<td>Dangling Access</td>
<td>405</td>
<td>147</td>
<td>46</td>
</tr>
<tr>
<td>Uninitialized Read</td>
<td>1195</td>
<td>94</td>
<td>230</td>
</tr>
<tr>
<td>Total</td>
<td>1656</td>
<td>240</td>
<td>267</td>
</tr>
</tbody>
</table>

Table 6.5: symMMU-derived Heap Violations

The system uses a combination of `ptrace` and iterative system call realignment to replay test cases under valgrind. The test replay process uses `ptrace` to control a valgrind process running the target program. The replay process intercepts valgrind’s system calls; a call is dispatched natively or intercepted and replayed, depending on whether the program or valgrind invoked the system call. In essence, two system models must be reconciled: test cases use precise system call logs whereas valgrind rewrites and inserts system calls. On interception, if the requested system call matches the head of the system call log, the valgrind process absorbs the log’s side effects. Otherwise, the call forwards to the operating system. Since there is no clear point where valgrind calls end and target program calls begin, the system call log initially synchronizes with valgrind by ignoring the first $n$ system calls. The value $n$ is found by successively incrementing $n$ to maximize the number of replayed system calls.

We checked 1919 Linux programs from the host machine’s (x86-64 Gentoo) `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` directories, symbolically executed for five minutes. Binaries were taken from the host machine because valgrind is sensitive to its host configuration. Table 6.5 lists valgrind-confirmed heap violations by number of tests, code sites (to control for faulty libraries), and programs (to control for noisy programs). In total 14247 violations were flagged in 761 programs; valgrind confirmed 11.6% of these tests. Valgrind and the executor disagree for several reasons: neither are bug-free, system calls fail to align, and differing memory layouts interfere with system call replay. Regardless, cross-checking with a third-party tool strengthens evidence of legitimate bugs.

Figure 6.11 illustrates the complexity of errors found by the heap checker. This pared-down example from `cpio-2.11` spans three source files; it reads a buffer length (`c_namesize`) from input (`short_hdr`), allocates a buffer `c_name`, passes it around,
then reads the data. If c_namesize is 0, prefix_len relies on uninitialized data, leading to undefined behavior.

### 6.5.7 Unconstrained Pointers

Explicitly modeling data structures for testing function arguments is tedious. Demand allocation on unconstrained pointers derives argument structure automatically. We evaluate symMMU unconstrained pointers on bare functions by symbolically generating test inputs for functions in several compiled libc implementations. These tests directly translate to C sources which serve as native test fixtures. Replaying the tests across libraries reveals implementation differences and fundamental bugs.

#### Generating libc Inputs

Test inputs were derived by symbolically executing C standard library (libc) libraries with unconstrained pointers. We tested functions from four up-to-date libc implementations: newlib-2.1.0, musl-1.1.0, uclibc-0.9.33.2, and glibc-2.19. Functions were symbolically executed by marking the register file symbolic and jumping to the function; root unconstrained pointers are demand allocated on dereference of a symbolic register. Each function was allotted a maximum of five minutes of symbolic execution computation time and 200 test cases. Since we intend to find differences between supposedly equivalent implementations, only functions shared by at least two libraries were evaluated. In total 667 functions shared among at least two libraries exhibited unconstrained demand allocations.
C Test Cases

To run unconstrained test cases natively on the host operating system, test cases are first converted into C code. The unconstrained buffer information is translated to a C source file then compiled into a binary program. This binary program maps the unconstrained buffers to their expected locations to reproduce the unconstrained environment for replaying the test. The binary dynamically loads the library and runs the target function with the test input.

The C test fixture programs operate as follows:
1 Declare initial data from unconstrained buffers.
2 Allocate memory buffers at given addresses.
3 Copy initialization data into buffers.
4 Load target function pointer from shared library.
5 Call target function with symbolic register arguments.
6 Print buffer contents and function return value.

Every C test case assigns unconstrained buffer data to physical locations in the test process. Every non-alias translation entry has an unconstrained descriptor structure which stored in an array of entries. The descriptor includes the buffer contents and length, the start of the buffer, the base of the memory segment to allocate and its length in pages.

Following the target function call, test results come from two parts of the process state: the return value and the input buffers. If a function returns a value that can be dereferenced, the value is translated to a fixed constant to conservatively avoid mismatches from differing addresses across libraries. When a function mutates its arguments, the values from the input buffers soundly reflect the updates.

libc Differences

Replaying the unconstrained buffers through the libraries revealed many implementation differences. The system detects subtle bugs in uncommon, but standard, cases which rely on pointer arguments. Complete bulk results suggest many potentially serious (although often benign) mismatches.
unsigned long a[16] = {0};
for (i = 0; i < 4; i++) {
    a[i] = strtol(s, &z, 0);
    if (z==s || (*z && *z != '.') || !isdigit(*s))
        return -1;
    if (!*z) break;
    s=z+1;
}

switch (i) {
    case 0: a[1] = a[0] & 0xffffffff; a[0] >>= 24;
}

for (i = 0; i < 4; i++) {
    if (a[i] > 255) return -1;
    ((char*)&d)[i] = a[i];
}

Figure 6.12: Simplified IP address parser from musl.

Figure 6.12 shows an example broken edge case detected with symMMU unconstrained pointers. The figure lists a simplified internet host address parser adapted from the musl library which converts an IPv4 numbers-and-dots notation string (s) to a network byte order integer address (d). During symbolic execution, the unconstrained buffer fills in the contents for s with symbolic values. The code works for four numeric parts (e.g., 127.0.0.1) but misinterprets other valid addresses. For example, the class C address “1.1” converts to 0x01000001 instead of the expected address 0x0101.

Table 6.6 summarizes the mismatches with glibc using unconstrained pointers. We were careful to exclude functions which rely on volatile system state, use structures with undefined width (e.g., stdio file functions), return no value, or always crashed. The percentage of mismatching functions is considerable given our conservative analysis. One interesting class of differences reflects arcane specialized configuration details. For instance, glibc’s timezone support causes newlib and musl to drift several hours when computing mktime (uClibc crashes, lacking /etc/TZ).
CHAPTER 6. SYMBOLICALLY EXECUTED MEMORY MANAGEMENT UNIT

<table>
<thead>
<tr>
<th>Library</th>
<th>Mismatched Functions</th>
<th>% Total Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>musl</td>
<td>86</td>
<td>25%</td>
</tr>
<tr>
<td>newlib</td>
<td>45</td>
<td>30%</td>
</tr>
<tr>
<td>uclibc</td>
<td>32</td>
<td>8%</td>
</tr>
<tr>
<td>Total</td>
<td>117</td>
<td>24%</td>
</tr>
</tbody>
</table>

Table 6.6: Mismatches against glibc involving unconstrained pointers

Furthermore, inconsistent locale handling across libraries contributes to mismatched wide character data.

6.6 Conclusion

This chapter introduced symbolic execution with symbolically executed memory accesses through a symMMU. By separating memory access policy from the dispatch mechanism, we have implemented a variety of access policies and memory analysis algorithms with minimal effort. Microbenchmarks demonstrate the importance of access policy for symbolic executor performance. Large scale comparative results indicate the symMMU finds more access faults with fewer false positives than a traditional built-in access dispatcher. Overall, this work suggests that forwarding memory accesses to a symbolically executed runtime is a beneficial design choice in a symbolic executor.
Chapter 7

Conclusion

7.1 Summary of Contributions

Chapter 2 introduced the design and philosophy behind the KLEE-MC binary symbolic executor. The chapter focused on the modifications and extensions necessary to adequately support machine code in a mature symbolic execution system. We demonstrate that KLEE-MC can analyze huge program sets, finding thousands of unique program faults. The additional system complexity and flood of data hints at the increased importance of robustness and correctness when analyzing machine code, a recurring theme throughout this dissertation.

Chapter 3 described the cross-checking capabilities which ensure high-integrity test case generation. The cross-checking in KLEE-MC validates the symbolic interpretation of LLVM code, the translation of machine code, solver stack, and expression optimization, finding executor bugs that otherwise would go unnoticed due to the overwhelming amount of test data. Since KLEE-MC analyzes machine code, cross-checking works down to the hardware level, both confirming when the symbolic executor models computation on physical hardware and detecting when it does not.

Chapter 4 extended the symbolic executor to support symbolic floating-point operations. The floating-point operations were self-testing by virtue of being runtime libraries, generating their own test cases. When tested against hardware, similar to the prior chapter, the tests revealed divergence among the floating-point code,
floating-point solvers, and physical floating-point hardware.

Chapter 5 presented the system’s reduction rule based expression optimizer. The system discovered expression optimizations from its own programs by searching a database of observed expressions, building reduction rules, and finally establishing equivalence with the solver. These rules generalize to match larger classes of expressions and verify their correctness using the constraint solver. Applying the rules results in a 17% reduction of total queries and a 10% speedup on average.

Chapter 6 continued extending the system by replacing built-in symbolic memory handlers with symbolically executed libraries. The library version of the built-in symbolic pointer policy is both more reliable and better at finding access faults. To demonstrate the mechanism’s flexibility, new heap analysis and lazily allocated buffer passes were also implemented as libraries, finding over a thousand heap violations and over a hundred libc implementation mismatches. The results from these passes were automatically confirmed outside the symbolic executor with third party tools.

Overall, this dissertation shows a binary symbolic executor should be built in such a way that it can test itself. Since machine code ultimately has a physical ground truth in hardware, designing a binary symbolic execution system without test case validation facilities makes debugging a needless struggle. By establishing a foundation of precision through fast detection of executor errors, should they arise, it becomes possible to develop new features, while keeping the old ones, to analyze programs with little concern over hidden false positives polluting results.

7.2 Future Work

Despite its strong self-checking properties, KLEE-MC still leaves open a great deal of research problems and engineering challenges. Generally speaking and as outlined in the introduction, the primary objective of KLEE-MC as a binary program analysis platform is to totally generate complete test suites for all interesting behavior in arbitrary machine code programs. There are several broad categories of work which can push the system toward this goal.
7.2.1 Programs and Modeling

KLEE-MC targets several architectures running Linux and 32-bit Windows; this is far from all programs. Acquiring a bulk set of programs, even with snapshotting, can be involved. Furthermore, even when programs run on the executors, many still need incredibly precise system modeling to exercise all paths.

As shown in Section 2.5.1, simply running a program on a system should not be taken for granted. Once a platform has a snapshotter, there tends to be little difficulty saving process images. On the other hand, merely running a program on its intended platform, natively constructing the process image, is less straightforward. If a program relies on an obscure library that is not installed, it will fail to launch; all dependencies must be resolved for the program to run. In general, no complex operating system perfectly automatically resolves library dependencies; the best solution may use virtual machines with all system repository packages installed. On Windows, programs and their dependencies are often distributed using software packers which unpack the true executable from a compressed data section following a few mouse clicks, therefore demanding an extra stage of extraction that requires some engineering to automate. Software packages that attempt to decode packed executables are unreliable in practice; a better method could programmatically fuzz interfaces and track file creation to retrieve any binaries written out to the file system.

Once a program is loaded, the system model should approximate the platform so that it behaves in all manners the program could possibly encounter. As discussed in Section 3.3.2, a poor model weakens the executor’s ability to produce good tests. Although continuing to develop system model code could lead to an excellent model, this approach appears rather Sisyphean: on top of carefully symbolically encoding all operating system behavior, new system interfaces would also need to be closely followed and supported (e.g., driver-defined ioctl). Ideally, the system model would be automatically derived from the target platform. This could be done by having a mechanism which reuses system code or having a method to learn an approximate model from system call traces.
CHAPTER 7. CONCLUSION

7.2.2 Scaling

There are two modes of scaling that are important for a program analysis system. First, there is breadth: the system should routinely process as many programs as possible, ideally sharing results. Second, there is depth: the system should be able to analyze programs of any size with a degree of thoroughness. Both takes on scaling deserve attention.

A system that can process nearly any binary program yet does not is a waste of potential. A modern trend in computing is the consolidation of services to huge, centrally maintained systems; one of these services in the future could be program checking. This centralization is a stark contrast to the developer tool model where each user runs the analysis tool on their own development machine; given the difficulty of installing symbolic execution tools (a task occasionally relegated to virtual machines \( [24, 34] \)), the myriad of options and failure modes that rely on specialist knowledge, the substantial and unbounded processing times, the inconvenience of merely reporting false paths (let alone fixing them), and that binary code is less closely guarded than source, the author firmly believes focusing on purely local tooling is remarkably short-sighted for pursuing future trends. The advantages to a consolidated model include greater opportunities for data sharing, thereby improving performance in general, and better visibility of system defects. Ultimately, improving breadth of data sets implies developing interesting tactics for reusing execution data (Chapter 5 is one example) as well as better automated infrastructure for managing bulk program data.

The resources needed to represent large program states pose a significant challenge. Arguably, larger programs could be handled by running the analysis on larger machines, but this quickly becomes unrealistic. The executor may eventually process larger programs by only considering small pieces of the program at once, to limit path length and compute time, or only holding a few states in memory at a time, to limit memory use. Unconstrained execution (§ 6.4.3) could be one solution for processing smaller components of larger software. A piecewise approach could then be stitched back together to analyze the program as a whole.
CHAPTER 7. CONCLUSION

7.2.3 Coverage

All symbolic executors suffer from poor coverage on large programs, partially because of the path explosion problem. If an executor fails to cover a significant portion of a program’s code, its tests will typically be superficial and of little value. Improving code coverage means addressing the path explosion problem.

If a symbolic executor begins at a program’s entry point, it often becomes lost in useless states due to following error paths; defensive programming means most inputs are cause for errors. On the other hand, running a program directly on its native platform, rather than modeling the platform on the executor, can avoid these error paths, reaching states a symbolic executor may never find. Hence, starting symbolic execution from states that are likely on an actual system (i.e., seed states) but unlikely under symbolic execution due to path explosion could improve coverage. Pulling snapshot sequences (§ 3.3.1) from a general desktop system with low-overhead could possibly be a source of useful seed states.

Good coverage can be cast as a search or scheduling problem. In essence, the executor has a set of states from which it must choose one to explore. An optimal state schedule for coverage selects states in an order that would cover the maximum amount of code while dispatching the fewest instructions. Although a general scheduler that always selects an optimal schedule is impossible, there are certainly heuristics which improve coverage in practice. Choosing the right heuristics is arguably an open problem, although as it stands, this has been an excuse to leave the executor’s scheduler as one of the least developed components of the system.

Finally, difficult to reach code can be covered by removal of context. One approach could use unconstrained execution with lazily allocated buffers to jump to code without a path; the challenge is to demonstrate that the tests are relevant with respect to the rest of the program. Another approach is to abstract [79] away functions or loops that cause path explosion, making side-effects symbolic; the challenge here consists of knowing the appropriate abstraction to apply (i.e, where to place side-effects) as well as soundly refining the abstraction to produce a legitimate program path.
7.2.4 Types of Analysis

The meaning of “interesting” behavior is intentionally vague. What precisely constitutes behavior of interest depends on the program semantics, platform semantics, and threat model. Essentially, behavior worth mentioning corresponds to the types of analysis applied to the program.

The primary focus of KLEE-MC has been on bugs that cause program faults leading to machine exceptions. These bugs are easy to confirm since the hardware detects faults on replay. However, most bugs do not raise exceptions; heap violations (§ 6.4.2) being one example explored in this work. Such bugs include privilege escalation, TOCTOU attacks, and information leaks. Additional functionality to finds these bugs, or any richer program properties for that matter, could be implemented through runtime libraries, but confirming any results will require integration with (or developing) traditional dynamic analysis tools.

Although KLEE-MC goes through great lengths to reduce system non-determinism when running programs, there are program behaviors only exposed through non-deterministic mechanisms. The two most obvious mechanisms are threading and signals. Naïve methods for handling non-determinism lead to exponential state explosion, making them impractical without further optimization (FIE [45], for instance, prunes equivalent states for small programs) or forcing determinism (Cloud9 [37], for instance, uses cooperative threading). Careful application of program knowledge, such as analyzing working sets to discover isolation boundaries, might prove more worthwhile toward finding bugs in non-deterministic code.

Still, this assumes some prior knowledge of what behavior is of interest. Aside from classes of bugs which may be ignored as an oversight, there are certainly classes of bugs without names. For instance, it may make sense to mine for idiomatic use of system features. Alternatively, searching for so-called “divergent” behavior could reveal new bugs. However, at the binary level, care must be taken to avoid false alarms since the meaning may be unclear, making analyzing potential bugs by hand a costly endeavor.
7.3 Final Thoughts

Binary symbolic execution is both the worst program analysis method and the best. It’s the worst because it’s expensive to run and hard to get right. One the other hand, it’s the best way to automatically produce sound test cases on arbitrary software. If finding software bugs en masse is the future of program checking, the human factor must reduced to primarily a supervisory role. Binary symbolic execution can lead the way toward such aggressively automated program checking by mechanically generating high-quality reproducible test cases for commodity software.
Appendix A

Glossary

This glossary defines important terms and acronyms that appear in this dissertation. Some of these terms may be overloaded in the literature but for this dissertation only have one meaning.

**Basic Block** – A code sequence with a single entry point. All control flow decisions exit the sequence.

**Binary Program** – A computer program, usually compiled from a higher-level language, in machine code format.

**Binary Symbolic Executor** – A symbolic executor that can symbolically execute binary programs.

**Bitcode** – Code compiled into the machine format for the LLVM intermediate representation.

**Branch** – A conditional jump in program code guarded by a predicate.

**Concrete Data** – Data that takes the form of a numeric constant.

**Concrete Replay** – Reproducing a path, free of symbolic data, with a concrete test case, usually with the JIT.

**Concrete Test** – A variable assignment and system call log used by the JIT for the concrete replay of a path.

**Concretize** – The process of converting a symbolic value to a concrete value with respect to a constraint set.

**Constraint** – A predicate that is an element of a state’s constraint set.
Constraint Set – A satisfiable conjunction of constraints associated with a state.
Contingent – The property of a predicate where both it and its negation have a satisfying assignment assuming the current state’s constraints.
Coverage – A metric for measuring unique instructions executed by a test.
Cross-check – Comparing two similar components for equivalence.
Dynamic Binary Translator – A system to translate basic blocks of machine code to an intermediate representation on demand
Expression – A well-formed formula over a set of operations and atoms. For symbolic execution, the set of bitwise and arithmetic operators, arrays, and numeric constants.
Expression Builder – A component of a symbolic executor responsible for constructing expressions.
Fault – Program behavior that causes hardware to raise an exception, often leading to program termination.
Feasible – The property of a predicate which has a satisfying assignment assuming the current state’s constraints.
Fork – The action of splitting a state into two states where one assumes a predicate $p$ and the other assumes the negation $\neg p$. Often brought about by a contingent branch.
Guest – The target program running inside the symbolic executor.
Instrumentation – Code that is inserted inside other code for analysis purposes.
Intermediate Representation – A language intended to express code semantics in a way that is easy to process by compiler-like tools.
Intrinsic – A built-in function in the symbolic executor accessible by runtime libraries. Similar to a system call.
IR – See intermediate representation
JIT – A Just-in-Time interpreter that compiles and runs LLVM code as executable machine code.
klee – An LLVM symbolic executor.
klee-mc – A machine code extension of KLEE.
Machine Code – The instruction format directly processed by a CPU.
Memory Object – A memory region in a state which represents a single copy-on-write unit.
MMU – Memory management unit

**Overconstrained** – A state or model that underapproximates possible program configurations.

**Path Constraint** – A constraint on a state incurred by following a contingent conditional branch.

**Predicate** – A boolean expression.

**Query** – A satisfiable set of constraints and a formula predicate bundled together for a solver which questions whether their conjunction is satisfiable.

**Reduction Rule** – A term rewriting template that matches and rewrites a class of expressions to logically equivalent smaller expressions.

**Register Log** – A trace of a program’s register states prior to every basic block dispatched by the symbolic executor.

**Replay** – The process of reproducing the path described by a test case.

**Runtime Library** – A symbolically executed collection of bitcode, loaded as part of every state, which supplies support functions for the executor.

**Satisfiable** – The property of a predicate which will evaluate to true under some variable assignment.

**SMTLIB** – A human-readable language for expressing constraint satisfaction problems for bit-vectors over arrays.

**Snapshot** – A point-in-time copy of memory, registers, and other resources for a running program

**Soft Floating-point** – An integer code implementation of floating-point operations. Typically used for emulating floating-point instructions on integer-only processors.

**Solver** – Software that implements a decision procedure to resolve constraint satisfaction problems.

**Solver Call** – A request made by the symbolic executor into a complete decision procedure to solve a query.

**State** – The materialization of partially explored path in the symbolic executor. A machine configuration with symbolic data.

**Symbolic Data** – Data that includes a variable term. (cf. concrete data)

**Symbolic Executor** – A complete system for symbolically executing code, including
a solver, expression builder, code translation, and test case generation.

**Symbolic Interpreter** – The component of the symbolic executor responsible for dispatching instructions on symbolic data.

**System Call** – A binary program request for an operating system service.

**System Call Log** – A log of side effects from the system model when servicing system calls for a path.

**System Model** – A runtime library that emulates an operating system or program runtime environment.

**Test Case** – Data which can be used to reproduce a program path.

**Unconstrained Pointer** – A pointer which maps to a lazily allocated buffer.

**Underconstrained** – An overapproximation of possible program configurations; potentially infeasible

**Valid** – The property of a predicate which has no satisfying assignment for its negation assuming the current state’s constraints.

**Validate** – To demonstrate correctness of code for a path.

**Variable Assignment** – A set of arrays of concrete values which concretizes an expression or set of expressions.

**Verify** – To prove correctness of code for all paths and values.
Bibliography


[104] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A Fistful of Red-pills: How to Automatically Generate Procedures to


