ON THE USE OF OVER-APPROXIMATE ANALYSIS IN SUPPORT OF SOFTWARE DEVELOPMENT AND TESTING

A Thesis Dissertation
Presented to
The Academic Faculty

by

Richard Rutledge

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing
School of Computer Science

Georgia Institute of Technology

December 2022

ON THE USE OF OVER-APPROXIMATE ANALYSIS IN SUPPORT OF SOFTWARE DEVELOPMENT AND TESTING

Thesis committee:

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Spencer Rugaber
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Marcelo d'Amorim
Department of Computer Science
North Carolina State University

Date approved: December 2022

Although this may seem a paradox, all exact science is dominated by the idea of approximation. When a man tells you that he knows the exact truth about anything, you are safe in inferring that he is an inexact man.

Bertrand Russell

For my family

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many people. Thank you to my advisor, Dr. Alessandro Orso, for your patience, guidance, and support. I have benefited greatly from your wealth of knowledge and meticulous editing. I am extremely grateful that you took me on as a student and continued to support my work over the years.

Thank you to my committee members, Dr. Vivek Sarkar, Dr. Qirun Zhang, Dr. Spencer Rugaber, Dr. Milos Prvulovic, and Dr. Marcelo d'Amorim. Your encouraging words and thoughtful, detailed feedback have been very important to me. Special thanks are due to my friend and colleague, Ravi Mangal, for his time, advise, and moral support. Last, but not least, my warm and heartfelt thanks go to my family for their endless patience and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The effectiveness of dynamic program analyses, such as profiling and memory-leak detection, crucially depend on the quality of the test inputs. However, adequate sets of inputs are rarely available. Existing automated input generation techniques can help but tend to be either too expensive or ineffective. For example, traditional symbolic execution scales poorly to real-world programs and random input generation may never reach deep states within the program.

For scalable, effective, automated input generation that can better support dynamic analysis, I propose an approach that extends traditional symbolic execution by targeting increasingly small fragments of a program. The approach starts by generating inputs for the whole program and progressively introduces additional unconstrained state until it reaches a given program coverage objective. This approach is applicable to any client dynamic analysis requiring high coverage that is also tolerant of over-approximated program behavior– behavior that cannot occur on a complete execution.

To assess the effectiveness of my approach, I applied it to two client techniques. The first technique infers the actual path taken by a program execution by observing the CPU's electromagnetic emanations and requires inputs to generate a model that can recognize executed path segments. The client inference works by piece wise matching the observed emanation waveform to those recorded in a model. It requires the model to be complete (i.e. contain every piece) and the waveforms are sufficiently distinct that the inclusion of extra samples is unlikely to cause a misinference. After applying my approach to generate inputs covering all subsegments of the program's execution paths, I designed a source generator to automatically construct a harness and scaffolding to replay these inputs against fragments of the original program. The inference client constructs the model by recording the harness execution.

The second technique performs automated regression testing by identifying behavioral

differences between two program versions and requires inputs to perform differential testing. It explores local behavior in a neighborhood of the program changes by generating inputs to functions near (as measured by call-graph) to the modified code. The inputs are then concretely executed on both versions, periodically checking internal state for behavioral differences. The technique requires high coverage inputs for a full examination, and tolerates infeasible local state since both versions likely execute it equivalently.

I will then present a separate technique to improve the coverage obtained by symbolic execution of floating-point programs. This technique is equally applicable to both traditional symbolic execution and my progressively under-constrained symbolic execution. Its key idea is to approximate floating-point expressions with fixed-point analogs.

In concluding, I will also discuss future research directions, including additional empirical evaluations and the investigation of additional client analyses that could benefit from my approach.

CHAPTER 1
INTRODUCTION AND MOTIVATION

Software permeates our everyday living environment, from critical services (e.g. aircraft separation and safety) to helpful conveniences (e.g. smartwatch reminders). When software fails, the ramifications can range from catastrophic loss of life, to a consequential loss of corporate market share. Moreover, the cost of software defects is a significant drag on the economy. A widely cited 2002 NIST report approximated the cost of bugs in the U.S. as $59.5 billion. A more current report by the Consortium for IT software Quality (CISQ) estimated the U.S. cost of software operational failures in 2020 as $1.56 trillion and identified the primary underlying cause as unmitigated software flaws [1].

1.1 Motivation

To reduce these costs, developers must detect these defects early. Dynamic program analysis is a common approach to defect detection, in which the analysis is performed during program execution. This approach can be effective in diverse tasks such as memory-leak and concurrency errors, but good results crucially depend on the quality of the test inputs. Since dynamic analysis can only verify a property along an executed path, the test inputs must provide high coverage of the targeted program behaviors. However, high-coverage inputs are not always available. Existing automated input generation techniques can help but tend to be either too expensive or ineffective. For example, traditional symbolic execution scales poorly to real-world programs and random input generation may never reach deep states within the program. For example, consider a simple server application that accepts a complexly formatted input, performs an analysis or transformation, and outputs the result. A typical architecture for such a system would entail two major modules: a parser to validate and instantiate the input and an operational module for analysis or transformation of

the instantiated object. Randomly formatted inputs would often fail pass the parser validation to reach and exercise the operational module, especially if the parser's input included a magic identifier value, checksum, or hash value over the input. Symbolic execution can find solutions for validated input, but due to the complex, looping flow control common to input parsers, the majority of the executor's explored paths may remain within the parser. Depending upon the executor's path scheduling algorithm, the paths within the operational module may easily be starved by their more numerous parser-bound brethren.

Traditional symbolic execution [2] has advanced considerably in the past decade. By treating all program inputs as symbolic values and conjoining constraints upon these input values at conditional branch statements (the so-called path condition), symbolic execution can theoretically explore all feasible program paths. At each statement, the symbolic executor can use the constrained symbolic values to compute the state of the analytic property. However, as both a precise and a complete analysis technique, symbolic execution suffers from scalability limitations. Since the number of paths explored tend to be exponential in the number of program branch statements, the symbolic execution of real-world programs often induces path explosion [3]. Also, since each path condition accumulates terms at every conditional branch, they can become lengthy and computations entailing the path condition can become a performance bottleneck. In describing the state of symbolic execution, Cadar and Sen [3] listed constraint solving performance as the second-greatest challenge to progress. Symbolic execution's runtime is often dominated by the solver itself.

1.2 Approach

For scalable, effective automated input generation that can better support dynamic analysis, I propose an approach that extends traditional symbolic execution by targeting increasingly small fragments of a program. The approach starts by generating inputs for the whole program and progressively introduces additional unconstrained state until it reaches a given program coverage objective. This approach is applicable to any client dynamic analy-

sis requiring high coverage that is also tolerant of over-approximated program behavior–behavior that cannot occur on a complete execution from the standard program entry point (i.e. `main`). Note that my approach does not attempt to compose the fragments into a coherent set of program inputs, which would violate the Anti-Extensibility axiom [4]. Instead, the approach considers behavior in a local neighborhood of a program point. Feasible program behavior is then a subset of this neighborhood behavior.

To assess the effectiveness of my approach, I applied it to two client techniques. The first technique infers the actual path taken by a program execution by observing the CPU's electromagnetic emanations and requires inputs to construct a model for recognition of executed path segments. In this technique, the approach generates inputs to code fragments generated in a neighborhood of each path segment for training. The client inference works by piece wise matching the observed emanation waveform to those recorded in the model. It requires the model to be complete (i.e. contain every piece) and the waveforms are sufficiently distinct that the inclusion of extra samples is unlikely to cause an inference error. After applying my approach to generate inputs covering all segments of the program's execution paths, I designed a source generator to automatically construct a harness and scaffolding to replay these inputs against fragments of the original program. The inference client constructs the model by recording the harness execution.

The second technique performs automated regression testing by identifying behavioral differences between two program versions. This technique identifies related neighborhoods in both versions of the program that bound the modified code statements. It then explores local behavior in these neighborhoods by generating inputs to functions near (as measured by call-graph) to the modified code. The inputs are then concretely executed on both versions, periodically checking both external and internal state for behavioral differences.

The final work presented in this thesis is preliminary work to extend the coverage of inputs generated by symbolic execution of floating-point programs. My technique automatically substitutes fixed-point data types and operations for native floating-points types in the

module under test. By replacing the floating-point types, the executor does not require a floating-point enabled solver, nor extensions to the constraint language for type awareness. The approach has limitations as compared to native floating-point. However, if the goal of the analysis is high coverage input generation, the substitution of fixed-point operations can find symbolic rational values to explore execution paths that current state-of-the-art techniques cannot.

1.3 Contributions

Inputs generated by extending traditional symbolic execution to target progressively smaller program fragments can improve the effectiveness of dynamic program analysis techniques that can tolerate some amount of infeasibility by considerably increasing the analysis's coverage.

My research provides the following novel contributions:

- Progressive Symbolic Execution (PSE), which progressively introduces additional unconstrained symbolic state for maximum path coverage.
- Zero-Overhead Path Inference (ZOPI) model construction, which extracts program fragments, uses PSE to generate fragment inputs, and constructs suitable scaffolding for the piece-wise execution of all program paths.
- Over-approximate Differential Testing (ODIT), which given two versions of a program, systematically explores all the changed program behaviors.
- Publicly available prototype tools for the techniques.
- Empirical evaluations supporting the utility and scalability of the techniques.
- A prototype tool and initial empirical evaluation of KLEE-FXP, a technique to substitute fixed-point approximation of floating-point expressions to improve symbolic execution coverage.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 gives a brief overview of foundational techniques upon which this work builds and defines some common terminology used throughout this thesis. Chapter 3 details my technique for high-coverage input generation, Progressive Symbolic Execution (PSE). Chapter 4 details my contributions to Zero-Overhead Path Inference (ZOPI). Chapter 5 describes my technique for behavioral differencing, Over-approximate Differential Testing (ODiT) and discusses experiments and case studies supporting ODiT scalability. Chapter 6 outlines ODiT future work for ODiT and applying fixed-point approximations to symbolic execution. Chapter 7 discusses related works. Finally, Chapter 8 concludes my thesis.

CHAPTER 2

BACKGROUND

This chapter provides a background discussion covering traditional symbolic execution [2] and under-constrained symbolic execution [5], together with limitations of both. It also includes a discussion on an essential supporting technique, lazy initialization.

2.1 Traditional Symbolic Execution

Traditional symbolic execution [2] executes a program with symbolic input instead of concrete values, exploring all feasible program paths by executing the program with symbolic external state. The symbolic executor maintains a set of symbolic states, each with an associated logical conjunction, the path condition PC, accumulated during execution. The executor starts with a single state with a symbolic value for each of the program inputs and an empty PC. It then executes program statements in their normal execution order and with their original semantics, with the following two exceptions:

1. The executor computes statement values as symbolic values relative to the original symbolic input values.

2. The executor forks a single state S when executing a branch statement with condition C into two states, S_T and S_F, with associated PCs:

   - $PC_T = PC \wedge C$
   - $PC_F = PC \wedge \neg C$ respectively.

   While retaining only those states whose PCs are satisfiable.

The executor continues by selecting a state and executing its next statement. Executors work in conjunction with a constraint solver which can compute satisfying solutions

to PCs. After the executor completes the final instruction for a state, the solver can then supply program input values that will reproduce the state's path through the program. In theory, symbolic execution could provide inputs for complete path coverage. But, in practice path explosion [3] (the number of states is exponential in the number of branches), solver performance overhead, and solver supporting theories required to solve PCs (e.g. floating-point) limit path coverage [3].

2.2 Under-constrained Symbolic Execution

Starting from $main$, traditional symbolic execution may never reach a targeted function before suffering path explosion [3] or arriving at an unsolvable constraint. Under-constrained symbolic execution [5] extends symbolic execution by substituting symbolic values for internal state as well as program inputs. By substituting symbolic values for function arguments and global variables, this technique can begin execution from any function, not just program $main$. However, the consequence of this ability to reach deep within a program for a targeted function is that under-constrained symbolic execution can produce infeasible inputs for a function f; that is, no program inputs executed from $main$ can call f with the generated inputs.

2.3 Lazy Initialization

Lazy initialization [6] is an important enabling technique for under-constrained symbolic execution, as it allows the handling of unconstrained pointers or references. In languages supporting pointers, lazy initialization [6] accounts for an unconstrained pointer p by forking states for each of:

1. p = $NULL$
2. p = new allocation
3. p = pre-existing memory object(s)

Various tools have implemented lazy initialization for symbolic execution in Java (e.g., [6, 7, 8]), C/C++ (via LLVM IR [9]), and object code [10]. Because this technique can inflict a significant performance penalty, researchers have proposed optimizations for Java-based symbolic execution [11, 12, 13]. However, these optimizations would be difficult to implement without Java's memory manager and strict type safety.

2.4 Definitions

In this section, I define some terms used throughout the remainder of my thesis.

Control Flow Graph (CFG). A *CFG* for a function f is a directed graph $G = \langle N, E, en, ex \rangle$, where N is a set of nodes that represent statements in f, $E \subseteq N \times N$ is a set of edges that represent the flow of control between nodes, and $en \in N$ and $ex \in N$ are the unique entry and exit points for the CFG.

Basic Block. A *basic block* in a CFG is a contiguous sequence of nodes (i.e., instructions) with no incoming branches except for the first node in the block and no outgoing branches except for the last node in the block.

Interprocedural Control Flow Graph (ICFG). An *ICFG* is a graph built by composing a set CFGs, where the CFGs are connected according to the call relationships between the functions they represent. Specifically, if a function f_1 calls a function f_2, the two CFGs for f_1 and f_2, G_1 and G_2, are connected as follows: the node n in f_1 that represents the call site to f_2 is replaced by two nodes n_c (call node) and n_r (return node), such that all predecessors of n are connected to n_c, and n_r is connected to all successors of n. Then, n_c is connected to G_2's entry node, and G_2's exit node is connected to n_r.

Call Graph. A *call graph* is a directed graph $G =< M, E >$, where M is the set of functions in the program, and an edge $(f_a, f_b) \in E$ implies that function f_a may call function f_b.

CHAPTER 3

PROGRESSIVE SYMBOLIC EXECUTION (PSE)

Dynamic program analysis is an important class of techniques to detect software defects. But these techniques can only analyze portions of the software that are executed. So, they are crucially dependent upon input coverage to be effective.

Analyzing targeted behavior deep within the program structure requires a set of program inputs that not only reaches the targeted code, but also thoroughly covers it. Manually created test inputs tend to be ad-hoc and cover only a small-subset of program behavior Automated input generation techniques can help, but tend to be either expensive or ineffective. For example, Random Input Generation may never reach program states deep within the program due to structured formats and context-dependent inputs such as checksums or cyclic redundancy checks (CRCs).

Traditional symbolic execution often also fails to reach the target due to state explosion or solver failures. Rice's theorem [14] tells us that any given program analysis in generality cannot be both sound and complete. In practice, this means the analysis will produce either false-negatives, false-positives, or both. Any single analysis technique can be designed to favor one or the other. By unconstraning only program inputs and executing from program entry, traditional symbolic execution avoids false-positives arising from infeasible program states. Therefore, if a vanilla symbolic execution tool reports a safety property violation, such as a memory fault, then the developer can be confident that the program has a defect and the tool can produce an input to demonstrate the bug.

But by emphasizing completeness, traditional symbolic execution can deprecate soundness by allowing false-negatives in the form of portions of the program not covered by the analysis. But, what if a particular client analysis requires high coverage to be effective? If precise techniques for input generation are ineffective and the dynamic analysis is tolerant

of some over-approximation (infeasible inputs), we can just symbolically execute increasingly smaller portions of the program in a local neighborhood of the target until achieving the coverage goal. This insight leads to my technique for Progressive Symbolic Execution (PSE).

3.1 PSE: Input Generation

To reach a coverage goal by executing increasingly smaller fragments of a program, PSE progressively unconstrains symbolic program state in four phases or strategies. (The phases are progressive in that the symbolic state in each phase is a superset of the symbolic state in the prior phase.)

uInp (symbolic input state): Execution of the whole program with symbolic inputs. This is equivalent to traditional symbolic execution performed from the entry point of the program.

uExt (symbolic external state): Execution of each function with symbolic input parameters and symbolic global state. Intuitively, this strategy corresponds to executing a function as if it could be reached with every possible state and is equivalent to under-constrained symbolic execution [5].

uStub (symbolic stubs): Execution of each function with symbolic input parameters, symbolic global state, and all callees replaced by symbolic stubs (see example in Listings 3.1 and 3.2). Symbolic stubs return an unconstrained value and unconstrain global state and values passed as output parameters. Intuitively, in addition to executing a function as if it could be reached with every possible state, this strategy also assumes that callees can modify the state in every possible way.

uInt (symbolic internal state): Executions of fragments of a function with symbolic local and global state. Intuitively, this strategy corresponds to unconstraining the state reachable by a code fragment, so that the code fragment can be executed as if it could be reached with every possible state.

```
1  int P(char *p) {
2    mark(m1);
3    int x=complex(p);
4    while (x > 4) {
5      mark(m2);
6      // some code
7    }
8    // some code
9    mark(m3);
10 }
```

Listing 3.1: Original code.

```
1  int P'(char *p) {
2    mark(m1);
3    int x=symbolic_int();
4    while (x > 4) {
5      mark(m2);
6      // some code
7    }
8    // some code
9    mark(m3);
10 }
```

Listing 3.2: Modified code.

The specifics of PSE's input generation are described in Algorithm 1. The algorithm takes as input (1) a program, (2) a set of entry points for the program, and (3) three tuning timeout parameters, and produces as output a set of replay cases. Each replay case consists of a program fragment (an internal program function) and corresponding inputs. (Although in most cases programs have a single entry point, supporting multiple entry points allows for applying the same approach to libraries.) The timeout values specify maximum time budgets for the progressive unconstraining strategies: $t0$ applies to uInp, $t1$ applies to uExt, and $t2$ applies to both uStub and uInt. The output of the algorithm consists of the replay cases generated by all progressive phases, which are stored in the container initialized in line 2.

The algorithm starts by performing the uInp strategy on each of the program entry points (lines 4-5). The uExt strategy iterates over each program function as discovered in a breadth-first traversal of the program call graph and maintained in a work list (lines 7-12).

I selected this traversal order to lengthen the average replay trace, as entry from functions closer to program entry should produce longer traces. Lines 13-14 ensure that the algorithm only executes uExt if a remaining path segment is reachable from the current function in the call graph traversal. In that case, the algorithm invokes PSE on the function with the uExt strategy, adds the resulting replay cases to set *cases*, and updates the set of remaining path segments (lines 15-17).

Note that, for clarity, I treat the utility function *extractPaths* as polymorphic; that is, the function always returns a set of intraprocedural path segments contained within its single argument. The precise segmentation definition is expected to vary with the client analysis technique and embodies the desired coverage metric. PSE will complete when inputs are found to execute each of these segments or the technique times out. If the argument is a function, it simply returns the set of path segments in the function. Conversely, if the argument is a program or a set of functions, it returns the union of the path segments in each function. Finally, if the argument is a basic block, it returns the set of path segments reachable from the block and within the function containing the block.

If there are remaining path segments within the current function, the algorithm aggressively unconstrains additional program state by substituting symbolic stubs for all callees within the function (uStub strategy). This strategy allows the algorithm to skip over complex callees that may be problematic for symbolic execution. (Since each function entry and exit is marked, path segments within the callee can be covered separately from the current function context.) Each replay case produced in this phase must be considered for retention (lines 21-27). These include replay cases that result in a memory fault due to the increased amount of symbolic state. The technique retains these faulting replay cases anyway in case they end up being the only cases covering a specific path segment.

When symbolic stubs fail to expose a remaining path segment, the algorithm proceeds to the uInt phase, which unconstrains also the program state at specific points within a function (lines 28-39). This portion of the algorithm is analogous to the uStub strategy,

except that it traverses the basic blocks within a functions' CFG instead of the functions within the call graph of the program. Also in this case, some replay cases may result in a memory fault and are retained in case they cover path segment not otherwise covered.

Before advancing to the next function in the call graph traversal, the faulting replay cases are examined for coverage of this function's remaining path segments to decide which ones to keep (lines 39-43). Because faulting replay cases contain a record of the faulting basic block and the number of times that block occurs in the replay trace before faulting, PSE's replay can use these replay cases, if needed, and terminate them prior to the execution of the faulting statement.

In its final part, the algorithm returns a set of replay cases selected from all the potential replay cases generated. Longer replay traces preserve more of the program context, and thus may represent more authentic internal state during a future replay. Therefore, given a set of replay cases covering a given path segment, the algorithm favors the case with the longest trace. It does so by sorting the candidate replay cases by trace length and greedily selecting cases to achieve maximum path segment coverage (lines 44-50).

Algorithm 1 relies on function *execPSE* to perform the different phases of its progressive symbolic execution. Algorithm 2 provides the details of *execPSE*. The inputs of the *execPSE* algorithm are (1) the program to symbolically execute, (2) the program point to be used as the starting point for the symbolic execution, (3) the strategy to be used, and (4) the timeout to be enforced.

The algorithm first sets *s* to the initial symbolic state, sets the first instruction to the first instruction in the *start* basic block, and initializes the set of active states with the single element *s* (lines 2-4). It then unconstrains the formal parameters to the function containing the *start* basic block (lines 5-6).

The algorithm continues to unconstrain program state according to the specified strategy (lines 6-12). The instruction processing loop (lines 13-33) is the same used in traditional symbolic execution, except for the way it handles call instructions (lines 18-33). If

either the callee f is an external function, or the unconstraining strategy is uStub or uInt, the algorithm (1) creates a new symbolic variable for each formal output parameter of f and (2) assigns this symbolic variable to the corresponding actual argument at the call site. Additionally, the algorithm creates new symbolic values and assignments for each global variable referenced by f and for f's return value, if present.

3.2 PSE: Optimizations

To make PSE more scalable, I have incorporated several optimizations in the technique.

Lazy Initialization PSE uses lazy initialization [6] to construct pointer inputs for execution at an arbitrary program point. Specifically, accessing an unconstrained pointer value causes PSE to explore potential program paths in which the pointer (a) is `null`, (b) points to a newly allocated memory object of the targeted type, or (c) points to an existing memory object of the targeted type. To prevent lazily initialized pointers to lazily initialized pointers from unrolling infinitely, PSE tracks the depth of lazy memory objects. When the depth exceeds a configurable threshold, only states for cases (a) and (c) above are considered.

Pointer Type-Casting Type-casting between pointer types is a common practice in C programs. For example, a pointer may be declared as a `char *`, accessed, and later cast to `struct foo *`. In these cases, the lazily initialized memory behind the pointer may no longer be large enough to store memory objects of the new type. To address this issue, lazily initialized symbolic objects have an immutable maximum physical size and a flexible visible size. A lazy object's initial visible size depends on the size of the allocated type. A subsequent cast to a larger type can increase the visible size, up to its physical size, whereas a cast to a smaller type does not decrease it. The visible size is used when reporting symbolic solutions and when enforcing the inbounds pointer assumption, which I discuss next.

Inbounds Pointer Assumption Since out-of-bound pointer accesses can result in non-deterministic behavior, lazy initialization ensures that unconstrained pointers either point to allocated memory or are `null`. However, there are other ways to have a potentially out-of-bounds pointer, such as through array indexing and pointer arithmetic. In PSE, an indexed operation automatically inserts a path constraint requiring the resulting pointer to be within the target allocation block. This approach reduces the path search space while only eliminating undesirable paths. Faulting or non-deterministic paths have in fact low utility when used to generate training samples.

Path Explosion Mitigation Rather than mitigating path explosion [3] using search strategies, PSE tries to eliminate undesirable states early using multiple heuristics. The inbounds pointer assumption discussed above, for instance, eliminates many abnormally terminating paths. Loops are also a significant cause of path explosion, as symbolic conditions within a loop body can create, at each iteration, a number of new paths exponential in the number of branches. To mitigate this issue, PSE periodically samples the number of active path states in each loop body. If the number of states in a single loop body grows across the sample interval by more than a configurable threshold, those paths are randomly reduced by 90%.

3.3 PSE: Summary

This chapter has described PSE, my technique for generating high-coverage inputs to drive a client dynamic analysis that is tolerant of some infeasible inputs. The next two chapters will present two such analyzes. The first enables inferring a program execution path from a distance (ZOPI) and the second is a fully automated behavioral differential testing tool (ODIT).

Algorithm 1: Input generation

```
Input  : program: program to analyze
         entry_points: program entry points
         t0: classic symbolic execution timeout
         t1: PSE native callees timeout
         t2: PSE stubbed callees timeout
Output: result : {< frag, input >}
1  begin
2    cases ← ∅
3    remaining ← extractPaths (program)
4    foreach fn ∈ entry_points do
5      cases ← cases ∪ execPSE (program, fn, uInp, t0)
6    remaining ← remaining \ coverage (cases)
7    visited ← ∅
8    fn_worklist ← entry_points
9    foreach fn ∈ fn_worklist do
10     visited ← visited ∪ {fn}
11     new_fns ← callees (fn) \ visited
12     append (fn_worklist, new_fns)
13     reachable_fns ← reaching (fn)
14     if remaining ∩ extractPaths (reachable_fns) ≠ ∅ then
15       new_cases ← execPSE (program, fn, uExt, t1)
16       cases ← cases ∪ new_cases
17       remaining ← remaining \ coverage (new_cases)
18       if remaining ∩ extractPaths (fn) ≠ ∅ then
19         faulting ← ∅
20         new_cases ← execPSE (program, fn, uStub, t2)
21         foreach case ∈ new_cases do
22           if remaining ∩ coverage (case) ≠ ∅ then
23             if faulted (case) then
24               faulting ← faulting ∪ {case}
25             else if completed (case) then
26               cases ← cases ∪ {case}
27               remaining ← remaining \ coverage (case)
28         if remaining ∩ extractPaths (fn) ≠ ∅ then
29           bb_worklist ← {bb | bb ∈ CFG(fn) sorted by BFS}
30           foreach bb ∈ bb_worklist do
31             if remaining ∩ extractPaths (bb) ≠ ∅ then
32               new_cases ← execPSE (program, bb, uInt, t2)
33               foreach case ∈ new_cases do
34                 if remaining ∩ coverage (case) ≠ ∅ then
35                   if faulted (case) then
36                     faulting ← faulting ∪ {case}
37                   else if completed (case) then
38                     cases ← cases ∪ {case}
39                     remaining ← remaining \ coverage (case)
40           foreach case ∈ faulting do
41             if remaining ∩ coverage (case) ≠ ∅ then
42               cases ← cases ∪ {case}
43               remaining ← remaining \ coverage (case)
44   result ← ∅
45   remaining ← extractPaths (program)
46   cs_worklist ← {case | case ∈ cases sorted by trace length}
47   foreach case ∈ cs_worklist do
48     if remaining ∩ coverage (case) ≠ ∅ then
49       result ← result ∪ {case}
50       remaining ← remaining \ coverage (case)
51   return result
```

Algorithm 2: execPSE(simplified)

```
Input  : program: program to symbolically execute
         start: program point to begin PSE
         strategy: uInp | uExt | uStub | uInt
         timeout: maximum time to perform PSE
Output: result : {< frag, input >}
1  begin
2      s ← initial_state[start]
3      fn ← containingFn(start)
4      active_states ← {s}
5      foreach arg ∈ formalArgs(fn) do
6          unconstrain(s, arg)
7      if strategy ∈ {uExt, uStub, uInt} then
8          foreach var ∈ globalvariables do
9              unconstrain(s, var)
10         if strategy = uInt then
11             foreach var ∈ localVars(fn) do
12                 unconstrain(s, var)
13     stop ← now() + timeout
14     while active_states ≠ ∅ ∧ now() < stop do
15         s ← selectState(active_states)
16         inst ← nextInstruction(s)
17         switch inst do
18             case Call do
19                 f ← targetFunction(inst)
20                 if strategy ∈ {uStub, uInt} ∨ f ∉ program then
21                     ¡ foreach arg ∈ actualArgs(inst) do
22                         if isOutputPointer(arg) then
23                             unconstrain(s, value)
24                             arg ← value
25                     foreach var ∈ globalvariables do
26                         if isReferenced(f, var) then
27                             unconstrain(var, value)
28                             arg ← var
29                     if returnType(f) ≠ void then
30                         unconstrain(s, value)
31                         setReturn(s, inst, value)
32                 else
33                     executeCall(s, f)
```

CHAPTER 4
ZERO OVERHEAD PATH INFERENCE

4.1 Introduction

Program tracing consists of logging selected events during program execution. Such trace logs are then used for various tasks, such as computer forensics, debugging, performance analysis, and user profiling. Typically, program tracing is implemented through instrumentation, that is, by adding probes to a program that log events as they occur.

Albeit effective, instrumentation can cause issues due to its intrusive nature. In particular, instrumentation adds runtime overheads that can be problematic in many scenarios, including real-time systems, embedded software, and deployed applications. To address these issues, while still being able to collect accurate (partial) program traces, prior work developed zero-overhead profiling (ZOP) [15], a technique that can profile a program without instrumenting it by leveraging the electromagnetic (EM) emissions generated by the processing hardware during execution. ZOP, although effective, has three main limitations. *First*, it requires extensive code coverage, and therefore a thorough set of test inputs, during its training phase to achieve good accuracy. Basically, in ZOP, each relevant path segment must be executed, so that its EM signal can be recorded and later matched. Unfortunately, in real-world programs, the test cases are frequently few, of poor quality, and often completely absent. *Second*, ZOP predicts acyclic path profiles [16], rather than complete execution traces. These path profiles count executions of unique, acyclic paths within the program. Although useful for some tasks, path profiles summarize away the exact sequence of events that would instead be logged in a complete path trace. *Third*, ZOP can fail to recover from a misprediction. ZOP attempts to match EM signals by following the control flow graph of the program being profiled. When a misprediction occurs, ZOP back-

tracks until it can find a path that better matches the signal. Although this approach can avoid mispredictions that result in infeasible paths, the predicted and actual control flows can diverge beyond recovery when the EM signals collected during training do not closely match the signals observed during profiling for more than a short time.

For this work, I and co-authors [17] propose zero-overhead path inference (ZOPI), a novel approach that extends ZOP and addresses its shortcomings. I made two principle contributions to ZOPI. *First*, to support the training phase even in the absence of an extensive set of inputs, I developed a new input-generation technique based on symbolic execution (see chapter 3): progressive symbolic execution (PSE). PSE overcomes some of the limitations of traditional symbolic execution by taking advantage of the fact that program path segments need not be observed in the context of a complete execution. More precisely, initially PSE executes the whole program with symbolic inputs as done in classic symbolic execution. If this fails to achieve sufficient coverage for ZOPI training, it proceeds to execute functions with symbolic inputs and unconstrained global state (similar to UC-KLEE [9]). PSE then continues by (1) substituting called functions with symbolic stubs and (2) increasingly unconstrained local state, until a given coverage objective is achieved. Although this approach can result in infeasible paths, this is not problematic when the execution of such paths is used in training—in most (if not all) cases, the over-sampled EM emissions will simply never match a signal produced during profiling. *Second*, I developed a technique to automatically generate the scaffolding to replay the PSE identified program fragments with their corresponding generated inputs. Co-authors [17] used these replays to record program path-segment electromagnetic emanation waveforms to populate a path inference model.

To evaluate ZOPI, I and co-authors [17] applied it to the three original ZOP benchmarks and to a new, larger benchmark. The results show that ZOPI is a promising approach. In particular, they show that ZOPI does produce accurate path inferences, with an accuracy over 90% for the cases considered. They also show that PSE is an effective tech-

nique, in that it was able to cover feasible paths missed by traditional symbolic execution, which contributed to increasing ZOPI's accuracy.

4.2 Background

In this section, I provide some necessary background information on ZOP, the previous technique for zero-overhead (acyclic paths) profiling, and on symbolic execution. I also define some terms that used in the rest of the chapter.

4.2.1 Zero-Overhead Profiling

Zero-overhead profiling (ZOP) [15] computes acyclic path profiles [16] for a program P by observing the electromagnetic (EM) emanations produced by a computing system during execution of (an unmodified version of) P. ZOP consists of two main phases: training and profiling. In the training phase, ZOP runs P against a set of inputs to collect waveforms for the EM emanations generated by the computing system running P. In the profiling phase, ZOP (1) runs P, uninstrumented and unmodified, against inputs whose executions need to be profiled, (2) records the EM emissions produced by the program, and (3) matches these emissions with those collected during training to predict which acyclic paths were exercised and how often. In an evaluation performed on several benchmarks, ZOP was able to predict acyclic-path profiling information with an accuracy greater than 94% on average.

Despite these positive results, however, ZOP has some shortcomings that limit its usefulness and general applicability. First of all, as discussed above, ZOP requires an extensive set of inputs in order to build good models in the training phase. In fact, the empirical results described above were obtained by using test suites that achieved complete branch coverage, which are rarely available in practice. In addition, acyclic path profiles provide useful information, but they summarize events into histograms and discard information about the full sequence of events. They therefore cannot be used for the many tasks for

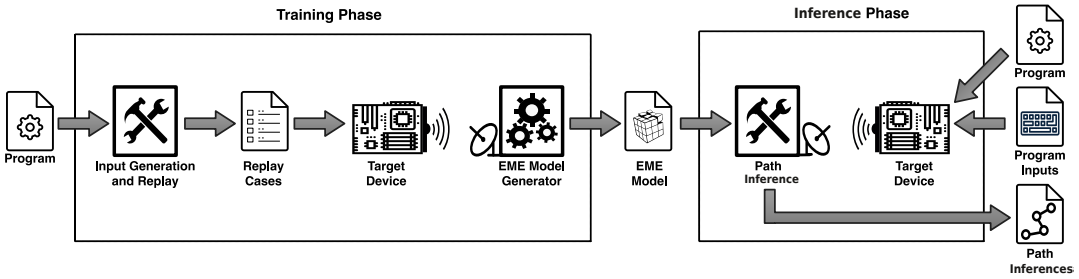


Figure 4.1: Overview of the ZOPI approach.

which information about complete traces is needed.

Finally, due to the way ZOP matches signals, the predictions it computes can suffer unrecoverable accuracy losses.

4.2.2 Symbolic Execution

Symbolic execution [2] is a technique that executes a program using symbolic instead of concrete inputs. At any point in the program's (symbolic) execution, the technique keeps track of (1) the *symbolic state*, expressed as a function of the inputs, and (2) the *path condition (PC)*, a set of constraints in conjunctive form that consists of the conditions on the inputs under which the execution reaches that point. The symbolic state and the PC are built incrementally during symbolic execution. When the technique executes a statement s that modifies the value of a memory location m, it computes the new symbolic value of m according to s's semantics and suitably updates the symbolic state. When it executes a conditional branching statement c, it forks the execution, follows both branches, and updates the PC along each branch by adding an additional conjunct that represents c's predicate.

When successful, symbolic execution can compute an input that would cause a given point in the program to be reached. To do so, the PC for that point would be fed to an SMT (Satisfiability Modulo Theories) solver, which would try to find an assignment to the free variables in PC (i.e., the inputs) that satisfies the PC.

4.3 Zero-Overhead Path Inference

Figure 4.1 shows an overview of ZOPI, the technique for zero-overhead path inference. (Please note that, to avoid clutter, some elements in the figure are repeated.) As the figure shows, ZOPI consists of two main phases: Training and Inference.

The *Training Phase* takes as input (the source code of) a *Program* P, whose complete paths are to be inferred, and generates the *EME Model*, a model of the electromagnetic (EM) emissions generated by the program. Two modules of ZOPI take part in this phase: the Input Generation and Replay module and the EME Model Generator. Given P, the goal of the *Input Generation and Replay* module is twofold. The first goal is to generate *Replay Cases*: inputs for P, or fragments thereof, that achieve a given coverage goal, typically expressed in terms of program path segments. The second goal is to replay the generated inputs against the program (or against a program fragment), so that the *EME Model Generator* can record the EM emissions generated during the replay and generate the *EME Model*, which links such emissions to the part of the program that generated them.

The *Inference Phase* takes as input the *EME Model*, *Program* P, and a set of *Program Inputs* for P, and generates a set of *Path Inferences*, one for each provided program input. The path inferences consist of complete execution traces for P and are computed by the *Path Inference* module, which (1) observes the EM emanations produced by the *Target Device* as it runs P against the provided inputs and (2) matches the observed emanations with those in the *EME Model*.

The approach described in the Training Phase is applicable to any language, architecture, or platform supporting an automated function-level input generation technique that achieves high structural coverage. (Of course, our prototype tool is more limited than that.) Conversely, the inference phase is currently constrained to very simple hardware designs. For example, the emanations from multiple cores are difficult to distinguish, modern multi-level memory caches drastically alter execution timings, and multi-stage execution

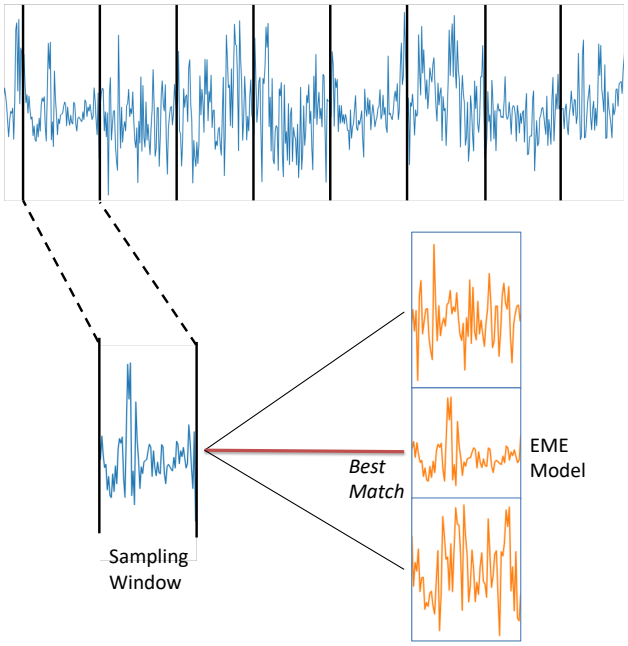


Figure 4.2: EM emission matching (Callen et al. [15]).

pipelines overlap executing instructions.

However, my contribution to this work is in all phases up to the physical recording of the *EME Model* and will be described in detail beginning in section 4.4. For context, here I will briefly describe the work of co-authors [17] to realize a complete functioning ZOPI prototype. As a microprocessor executes an instruction, electrical flows within the processor generate electromagnetic (EM) waves. Since each instruction uses different circuits within the processor, each instruction has a distinct impact on the overall EM wave generated. Although each individual instruction is brief and EM noisy, a sequence of instructions can be recognizable, and their waves constitute a side-channel emanation from the processor To capture the EM, a co-author targeted the processor with an antenna connected to a digital spectrum analyzer for recording. At the conclusion of the training phase, the EME model is populated with a superset of all feasible execution path segments linked to its execution EM recording. During inference, the captured waveform is compared to segments in the EME Model to find the best match. Note that an exact match is not required. As long as the signal-to-noise ration is within tolerable limits during both training and inference, the closest match is the most likely segment taken.

In the rest of this section, I return to describing my contribution to ZOPI in detail.

4.4 Input Generation

The training phase of ZOPI requires the recording of sample EM emissions collected from the *Target Device* (i.e., the device that runs the program whose traces to be collected). Moreover, for the training to be effective, ZOPI needs to collect a comprehensive set of samples. Ideally, the technique would need to collect one sample for every possible path in the program, which is clearly impractical. Because ZOPI's signal matching divides the EM emissions for an execution into smaller sampling windows, however, having samples of path segments whose length is comparable to that of the sampling window is typically enough. The goal of the Input Generation and Replay module is therefore to generate inputs that adequately cover a suitably identified set of segments within the program.

The first step performed by this module is a preprocessing of the source code that performs a set of semantics-preserving transformations aimed to facilitate later source code manipulation. Specifically, the preprocessing expands macros, refactors short-circuiting boolean expressions, encloses single statement blocks in braces, and rewrites each return statement that involves a complex value as an assignment to a temporary variable followed by a simple return of that variable.

Next, the module instruments the source code by inserting *markers* (i.e., special probes) at selected program points. These markers partition an execution trace into segments, referred to as *m2m paths* (marker-to-marker paths). These m2m paths are the path segments within the program functions that the inputs need to cover; that is, they are the coverage requirements for the input generation. The level of granularity of the inserted markers is critical to the effectiveness of the approach. Path segments that are too short would be easy to cover but would result in EM signals that are hard to recognize and match. Conversely, segments that are too long would generate EM signals that are easy to recognize and match but would be difficult to cover. Based on past experience [15], preliminary experimen-

tation, the way ZOPI performs signal matching, and domain knowledge, I selected the following points for inserting markers: entry nodes of functions, exit nodes of functions, loop heads, and target nodes of `goto` statements. Furthermore, because these markers can occasionally result in excessively long m2m paths, the technique splits m2m paths longer than a selected threshold by suitably inserting additional markers. I selected these marking criterion so that (1) m2m paths are intraprocedural and do not contain cycles, (2) any program trace can be represented as a sequence of contiguous m2m paths, and (3) the length of the signals generated by the m2m paths is comparable to that of the sampling window used by the Path Inference module.

Given a complete set of m2m paths, the technique tries to generate inputs that cover all such paths using an approach based on symbolic execution. In principle, traditional symbolic execution can generate inputs for all feasible paths in a program. However, its effectiveness and scalability are limited in practice by several issues, and in particular by the path explosion problem—the fact that the number of feasible paths is usually exponential in the number of code branches [3]. In fact, traditional symbolic execution has problems covering even individual statements, let alone m2m paths, that are located deep in the call graph or hidden behind complex, looping control flow. To address this problem, and be able to generate inputs that cover most m2m paths, I defined a new technique that extends classical symbolic execution, called *progressive symbolic execution (*PSE*)* (see chapter 3).

The key insight that allows PSE to generate inputs for ZOPI is that a given m2m path mp in program P need not be observed along a complete path, that is, a path that starts from P's entry and follows a complete, top-down execution. If PSE cannot generate an input that executes mp, it therefore derives a related program P' that contains an equivalent path segment for which it can find an input. To generate P', PSE operates along two dimensions: it (1) considers increasingly smaller fragments of the program and (2) replaces calls to other functions or libraries with symbolic stubs.

The example shown in Listing 4.1 illustrates this second dimension. Assume that we are

```
1  int P(char *p) {
2    mark(m1);
3    int x=complex(p);
4    while (x > 4) {
5      mark(m2);
6      // some code
7    }
8    // some code
9    mark(m3);
10 }
```

Listing 4.1: Original code.

```
1  int P'(char *p) {
2    mark(m1);
3    int x=symbolic_int();
4    while (x > 4) {
5      mark(m2);
6      // some code
7    }
8    // some code
9    mark(m3);
10 }
```

Listing 4.2: Modified code.

interested in covering m2m path $\langle 5, 6, 7, 8, 9 \rangle$, between markers $m2$ and $m3$, and that the symbolic execution of function `complex()` either results in a timeout or cannot be performed because the code of the function involves theories not supported by the underlying solver. In such a case, no input covering the path of interest would be produced. However, it would be straightforward to generate an input that covers the analogous path segment $\langle 5, 6, 7, 8, 9 \rangle$ in the derived program P' in Listing 4.2, where the integer value returned by `complex()` has been replaced with a symbolic integer.

PSE produces a set of inputs encoded as *Replay Cases*, where each replay case is an ordered pair that consists of a program fragment and inputs for that fragment. It also constructs, for each replay case, the scaffolding necessary to run the corresponding code fragment against its input. The next section discusses how P or fractions thereof execute against the generated inputs to support ZOPI's training phase.

4.5 Input Replay for EME Model Construction

The replay cases (i.e., inputs) generated by PSE for a program P using its uInp strategy can be run directly on P. This is not true, however, for the replay cases generated by PSE using its uExt, uStub, and uInt strategies, for which suitable scaffolding must be created. The reason is that these replay cases are generated by unconstraining program state, considering fragments of the program, and replacing called function with symbolic stubs.

ZOPI generates the needed replay scaffolding in the same language as P, and the scaffolding consists of four major parts: replay bodies, replay stubs, input data, and replay harnesses.

The *replay bodies* contain the source statements that comprise the m2m paths recorded for ZOPI training. Bodies for replay cases generated using the uExt strategy simply consist of the original function and corresponding callees. Bodies for replay cases generated using the uStub strategy also consist of the original source function, but they are linked against newly created replay stubs that return the right values and suitably set output parameters and global variables. In addition to this, bodies for replay cases generated using the uInt strategy must also be able to (1) start executing from an internal basic block bb, (2) initialize the local and visible global state at bb, and (3) exit at the right point in the execution (i.e., after visiting a termination basic block a specific number of times). To do so, PSE first creates a copy of the original function containing the fragment of interest and identifies the statement $stmt$ from which to start the execution. It then inserts a `goto` instruction at the beginning of the current body, causing the execution to jump to the $stmt$. Finally, it inserts a call to a function that suitably initializes local and global state.

The *replay stubs* correspond to symbolic stubs and provide suitable values for returns, output parameters, and global variables. Furthermore, because each called function can be invoked multiple times, ZOPI creates ordered sets of values, enabling the production of

the right values for the different invocations. When executing a replay case with symbolic stubs, ZOPI substitutes these replay stubs to the original called functions in the corresponding replay body. It is worth noting that, because the function entries and exits are marked program points, during signal matching the EM emissions from an actual recording of the called function can be considered instead of the EM emissions generated by a stub.

The *input data* consist of static-initialized data structure arrays in the original source language produced from the generated input values in the replay cases. Since the solver produces byte arrays for each symbolic entity, I treat each memory object as a Binary Large Object (BLOb) and decompose into fundamental data types and map onto the source language data types for the target machine architecture. Note that the data input set for a replay case may not contain complete values for all the required input variables. For instance, a function may read the value of an input pointer but only write to the pointed memory block. In general, since the values not contained in the input set do not affect the execution, ZOPI can safely initialize the missing data items with some default value.

Among the input data, pointer variables require special handling, as the address space during input generation is different from the address space during replay. To address this issue, ZOPI adds to the replay cases a map of the address space at the time the case was generated. This map includes the address and size of each memory object, the allocation type, and a list of the cast operator types applied to the memory object. ZOPI needs the entire address space because pointer values that are not in the generated input set cannot be assigned a default value as ZOPI does with fundamental types. Writing memory by dereferencing such a pointer would in fact likely result in an access violation if not in undefined behavior. Given this map, if the pointer value resolves into a memory object in the address space of the replay case, PSE calculates its offset and emits the replay pointer value as an offset into the statically-initialized data structure for the target object. Otherwise, it is given a default initializer based on its type. I iterate through members of a structure variable emitting a sequence of static initializers and extract each member's data from the memory

object BLOb at the correct offset. Nested structure member variables within an enclosing structure variable are recursively decomposed.

Finally, the *replay harnesses* contain one harness for each replay body and fragment, and a driver that invokes each replay fragment harness in turn. The fragment harness iterates through each input data set for the fragment, initializing global variables, declaring and initializing fragment parameters, and initializing substituted stubs.

4.5.1 EME Model Generator and Path Inference

The goal of the EME (EM Emissions) Model Generator is record the replayed path segments, relating EM emissions to the code instructions in the program that generated them. Path Inference is a fundamental part of ZOPI, as it is the component that actually produces path inferences using the EME Model generated during the training phase. Since co-authors completed these modules, this work will not be further addressed within this thesis.

4.6 Empirical Evaluation

To evaluate the effectiveness of the technique, I implemented it in a prototype tool and performed an empirical evaluation on a set of benchmarks. In the evaluation, I and co-authors [17] addressed the following research questions:

RQ1: Does ZOPI provide accurate path inference?

RQ2: How does ZOPI compare to ZOP?

RQ3: To what extent does state unconstraining help coverage?

Although RQ1 and RQ2 directly measure path inference results, they indirectly support the effectiveness of PSE and replay scaffolding generation.

4.6.1 Implementation Details

I implemented the modules of ZOPI discussed in Section 4.3. I used CIL [18] and clang [19] to preprocess source code. I used a combination of clang and NetworkX [20] for control flow graph analysis. I implemented PSE by extending the KLEE [21] symbolic execution engine. I also relied on LLVM [22] and on the STP constraint solver [23], My implementation of PSE is publicly available as a self-contained docker image [17].

4.6.2 Evaluation Setup

Table 4.1: Benchmark statistics.

Benchmark	LOC	Basic Blocks	M2M Paths
replace	495	245	229
schedule	464	175	153
print_tokens	579	178	153
mDNS	24,815	3,939	5,763

To answer the research questions I and co-authors [17] selected four benchmarks. The first three were used to evaluate ZOP in previous work [15]. The fourth benchmark, a real-world mDNS server, shows the scalability of the approach, as it is two orders of magnitude larger than the other benchmarks. Table 4.1 provides size metrics for these benchmarks. The target device was an Altera Cyclone II FPGA with a Nios IIe processor. Using an FPGA permits leveraging various debugging features and I/O pins to better understand program behavior at the individual-cycle level. Unfortunately, however, it also considerably limits the size of the potential benchmarks.

Using the timeout parameters described in Section 4.4, I defined three variants of PSE. This allowed me, together with the use of vanilla KLEE, to evaluate the effects of different input generation techniques on ZOPI's accuracy:

CS: Classic symbolic execution. Vanilla KLEE [24].

UC: Under-constrained symbolic execution. UC-KLEE proxied by performing PSE at the function level.

PG: PSE with all unconstraining strategies enabled.

FN: PSE without the uExt strategy. (Basically, this parameterization skips under-constrained symbolic execution by introducing symbolic stubs and considering sub-function fragments right away, which makes the analysis considerably faster at the cost of generating shorter paths.)

4.6.3 RQ1: Inference Accuracy

Table 4.2: Mean path inference accuracy (%).

Benchmark	**CS**	**UC**	**FN**	**PG**
replace	90.05	93.89	90.31	93.89
schedule	94.27	94.31	94.45	94.44
print_tokens	77.71	93.35	78.72	93.36
mDNS	98.94	98.97	98.95	98.95

To answer RQ1, I first generated replay cases (i.e., input sets) for the four benchmarks and for the four input generation strategies considered: CS, UC, FN, and PG. Second, I used ZOPI to generate EME Models for each benchmark and input set. Then, for replace, schedule, and print_tokens, I randomly selected 100 inputs from the tests provided in the SIR repository [25]. For mDNS, I used Avahi [26] to generate 9 inputs (i.e., mDNS queries) that target different host addresses and ask for different services, including incorrect queries. (I manually checked that the inputs exercise different aspects of the mDNS

protocol.) Finally, co-authors [17] used ZOPI to perform path inference using the generated EME Models.

Table 4.2 reports the path inference accuracy for the cases considered, computed by measuring the edit distance between actual and inferred paths. The inference error is the edit distance divided by the length of the actual path, and the inference accuracy is 1 minus the inference error.

As the table shows, for all four benchmarks the inference accuracy tends to be generally fairly high. The highest accuracy is achieved with either UC or PG in three of four cases, with the fourth case (schedule) showing a very close result for FN and PG. Interestingly, as I will show in section 4.6.5, PG achieves higher coverage but does not always result in higher inference accuracy. The reason for this is that the increased accuracy from a fully populated waveform model is offset by an increased possibility of a misinference when the sampling window straddles the entry or exit of a symbolic stub. Approaches to match sub-window EM signals could address this issue and further improve the results for PG.

Also interestingly, ZOPI achieves high inference accuracy for mDNS regardless of the input generation strategy involved. Further analysis of the results showed that this happens for different reasons, with the main ones being that (1) all the paths the program takes when receiving a valid mDNS packet are similar because traces are largely dominated by loops with a large number of iterations, and (2) all the paths the program takes when receiving an invalid mDNS packet are extremely short. This skews the results considerably and makes it so that any input generation strategy that produces even just a few valid and invalid packets result in reliable EME Models, and thus high accuracy in the inference.

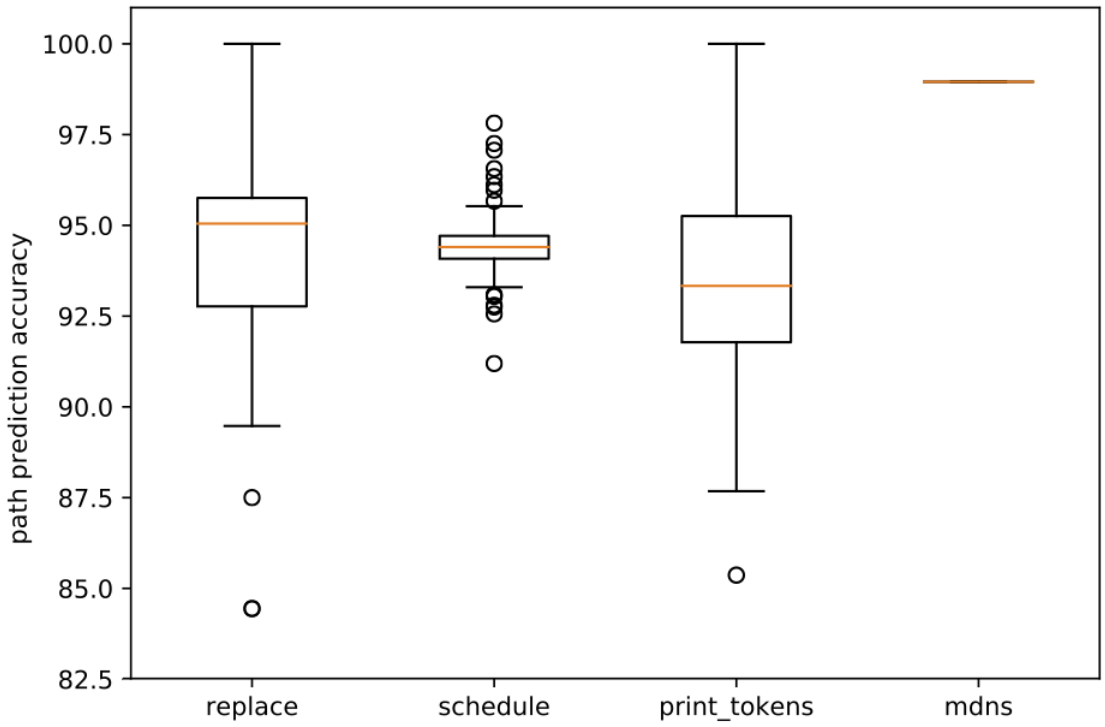


Figure 4.3: Detailed path inference accuracy for PG.

To better understand how path inference accuracy varies across inputs, consider the box-and-whisker plot in Figure 4.3, which shows detailed results for PG. (The plot for UC is fairly similar.) For each benchmark (x-axis), the figure shows the range of inference accuracy (y-axis). The boxes represent the 1st and 3rd quartiles, with an interior band at the median. The whisker ends represent the lowest and highest points within 1.5 of the interquartile range. Dots signify outliers.

Although Figure 4.3 shows consistently high path inference accuracy, it also shows that there is room for improvement. For example, accuracy above 96% for the schedule benchmark is an outlier. In general, some limitations of the inference technique can affect the results. Furthermore, I found that different m2m paths may sometime result in instruction sequences that are very similar to each other, even if they execute different parts of the program, and thus generate EM emissions that are also very similar to each other [27]. For example, two different m2m paths may have the exact same mix of store and ALU operations. For another example, switch statements are commonly compiled into jump tables,

which leads to multiple paths generating EM emissions that are difficult distinguish (a case that often happens for print_tokens and mDNS and causes a drop in accuracy).

4.6.4 RQ2: Comparison with ZOP

Table 4.3: Acyclic-path profiles prediction accuracy (%).

Benchmark	**ZOP**	**ZOPI (UC)**	**ZOPI (PG)**
replace	94.70	94.11	94.11
schedule	95.10	94.71	94.96
print_tokens	97.90	91.36	91.33

ZOP and ZOPI produce to some extent apples and oranges, as the path profiles computed by ZOP and the complete traces inferred by ZOPI are not directly comparable. To generate complete traces with ZOP would require extending the approach—and basically re-invent ZOPI. However, the path profile prediction accuracy of ZOP can be compared with ZOPI inferences by extracting acyclic-path profiles from complete traces. Table 4.3 shows these results, computed for the traces generated using UC and PG and for the three benchmarks with published ZOP results. As the table shows, ZOPI's accuracy is lower than but comparable to that of ZOP for replace and schedule (less than one percentage point), and slightly lower for print_tokens (around 6.5 percentage points).

The main reason for this slight decrease in accuracy lies in the fact that ZOP used a stateful model that selects the best match from m2m paths reachable from the currently predicted marker. This strategy reduces mispredictions, but is problematic when a misprediction does occur. Conversely, ZOPI's stateless inference suffers no additional penalty for misinferences, which makes it effective for inferring complete paths, but can result in a larger number of individual incorrect path segment inferences.

4.6.5 RQ3: Coverage

Table 4.4: Coverage comparison among CS, UC, and PG/FN.

	M2M Paths			
Benchmark	**total**	**CS**	**UC**	**PG/FN**
replace	229	168 (73.4%)	179 (78.2%)	206 (90.0%)
schedule	153	119 (77.8%)	136 (88.9%)	149 (97.4%)
print_tokens	153	115 (75.2%)	132 (86.3%)	143 (93.5%)
mDNS	5763	509 (8.8%)	2454 (43.3%)	4147 (72.0)%

Table 4.5: Missing m2m paths in the training inputs.

Benchmark	**CS**	**UC**	**PG/FN**
replace	3	2	0
schedule	7	2	0
print_tokens	9	3	0
mDNS	96	18	0

To answer RQ3, I measured the m2m path coverage achieved on all the benchmarks by the four input generation techniques considered. Table 4.4 shows the results, together with the total number of m2m paths determined by static analysis. The PG and FN replay cases produced identical coverage, so their results are shown together.

As the table shows, PG/PN achieved higher m2m path coverage than UC, which in turn achieved higher coverage than CS. This result is not surprising, as increasing symbolic state should necessarily lead to higher coverage. A more interesting question is how many additional *feasible* m2m paths were covered by PG/PN. Unfortunately, I cannot compute this information automatically, as it is an undecidable problem, and doing it by hand would be extremely time-consuming and error-prone. I can however compute a lower bound for

this information by checking how many of the m2m paths covered by the evaluation inputs in the study (i.e., in the actual executions used to evaluate inference accuracy) were missed by the generated input sets used for training; a decreasing number of uncovered m2m paths would necessarily indicate an increased number of feasible paths covered. Table 4.5 reports this information and clearly shows that PG/FN (i.e., PSE) consistently covers additional *feasible* path segments that UC does not cover, and so does UC with respect to CS. This result provides initial indication that it is worth pursuing more aggressive unconstrained state approaches when generating inputs. However, more research and experiments are needed to demonstrate that client techniques can indeed benefit from the additional coverage achieved.

4.7 Threats to Validity

In this section, I briefly discuss the main threats to the validity of the empirical evaluation and steps taken to mitigate them. The main threat to internal validity is the potential for defects in my implementation. In mitigation, I based the implementation on KLEE, a reliable and stable symbolic execution engine. I also carefully tested the implementation of PSE, which is available for public inspection [17]. Threats to external validity include the size and number of benchmarks. As discussed in Section 4.6.2, the maximum size of a potential benchmark was unfortunately constrained by the limitations of the embedded processor used in the evaluation. (Programs larger than mDNS could not be loaded onto the FPGA board.) Similarly, long signal recording, measurement, analysis, and human checks limited the number of benchmarks. Other threats to external validity are the way I selected inputs, especially for mDNS, and the possible lack of generalizability of the results to other devices.

4.8 Conclusion

This chapter presented ZOPI, a new approach that can collect complete execution traces accurately and with zero overhead by leveraging EM emanations. ZOPI can greatly benefit and support several important developer tasks, such as debugging applications, tuning performance, and profiling users. Although stateless signal matching allows ZOPI to recover from mispredictions, it also causes some additional mispredictions. In future, a mediated use of control flow information may allow for increasing accuracy while still permitting recovery after mispredictions. The evaluation results show that ZOPI is indeed effective and can produce accurate execution traces without requiring any program instrumentation.

CHAPTER 5

OVER-APPROXIMATE DIFFERENTIAL TESTING (ODIT)

5.1 Introduction

Most software continuously evolves through repeated cycles of program updates for bug fixes, performance improvements, and addition of new features. As software evolves, developers must verify that a proposed update contains all planned, intentional behaviors and does not introduce unintended behaviors, typically known as *regression errors* or simply *regressions*. In an ideal world, developers would have a formal specification of the desired program behavior and verification could be completely formal and automated. In practice, however, specifications are typically missing, and developers rely on testing to detect regression errors. Specifically, developers execute regression test suites, or parts thereof, on the modified software, hoping that regressions would result in test failures.

Unfortunately, test suites are typically and necessarily limited in terms of the behaviors they can cover [28, 29]. Moreover, even when a behavior is covered by a test suite, oracles may fail to detect behavioral differences because they often focus on and check specific parts of the program state.

Researchers have proposed approaches that try to address these issues, but the effectiveness of these techniques tends to be limited by practical factors. In particular, techniques that under-approximate behavior with random program input generation or symbolic execution (e.g. [30, 31]) have difficulty finding inputs that reach changes that are distant from the program entry. Conversely, over-approximated techniques (e.g., [32, 9]) only check externally observable (macroscopic) differences, such as program crashes or leaked memory.

To address these limitations of traditional regression testing approaches, in this chapter I introduce a new technique called ODIT—overapproximate differential (regression)

testing. Given two program versions, ODIT (1) analyzes the two versions to identify input-compatible internal functions defined in both versions, (2) generates test inputs for these functions that reach the changed code, and (3) leverages differential testing [33] to identify behavioral differences between program versions. A function f_0 in the original program and a function f_1 in the modified version are input-compatible if they have the same signature; that is, the same function name, sequence of argument types, and return type. Inputs generated for f_0 can then be applied to f_1 and vice versa.

ODIT utilizes progressive symbolic execution (PSE) (see chapter 3) to generate inputs (i.e., parameter and global values) for input-compatible functions that exercise the modified code in either version. For behavior comparison, the technique then concretely executes each input on the original version, while retaining periodic program-state snapshots. Each input, together with the state snapshot sequence resulting from its original version execution (used as an oracle) serves as a test case for the modified version. After concretely executing the test case on the modified version, the technique then compares corresponding state pairs from the original and modified program snapshot sequences to identify behavioral differences. In addition to potential regressions, this over-approximated list will also contain redundant, infeasible, and intentional differences. Therefore, ODIT clusters differences and ranks them based on their likelihood of being unintentional according to a heuristic.

To evaluate the technique, I considered 61 real-world, known regressions from the CoREBench suite [34]. I then performed two studies. First, I applied ODIT to the 43 out of the 61 CoREBench regressions for which I could reliably define a ground truth (in the form of an oracle). ODIT was able to automatically identify 25 of the 43 regressions, and for 10 of these 25 regressions, ODIT produced no false positives. For the remaining 15 regressions, ODIT also generated false positives, but it ranked the true positives in the top three positions in most cases. Next, I compared ODIT to Shadow [31], a state-of-the-art under-approximated technique, using its published results for 20 of the regressions in

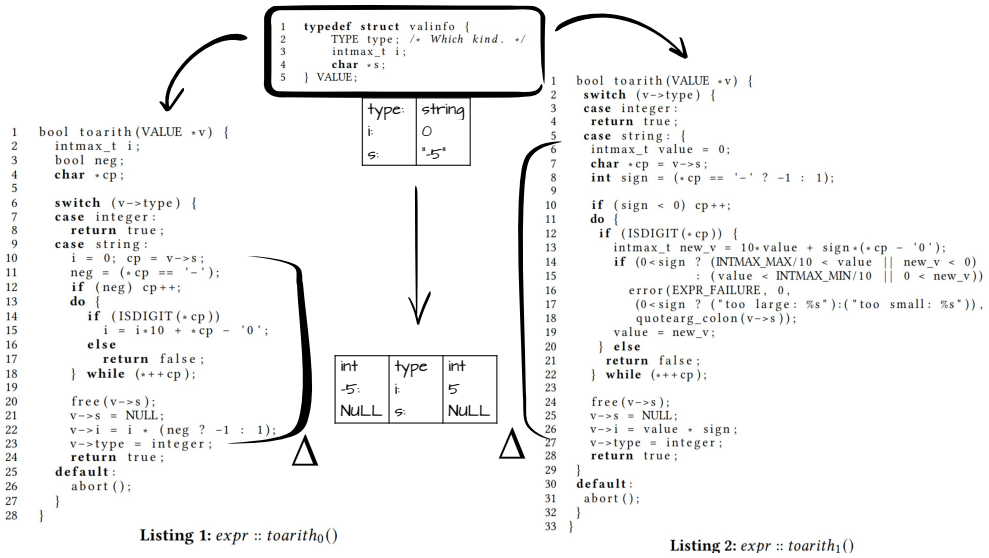


Figure 5.1: Illustrative Example

CoREBench. ODIT detected 14 of the 20 regressions considered, whereas Shadow detected at most 8 of them. Moreover, ODIT was an order of magnitude faster than Shadow.

These results, albeit still somewhat preliminary in nature, are promising, as they show that the technique can (1) automatically identify regressions, while (2) generating a low number of false positives, and (3) ranking the true positives in top positions. The results also show that ODIT (4) can outperform a state-of-the-art technique.

I envisage developers applying ODIT routinely (e.g., prior to committing changes to a repository) to verify that all behavioral changes are intentional. The evaluation provides initial evidence that the technique supports such usage.

5.2 Illustrative Example

Before describing the details of the technique, I introduce an illustrative example. First, I describe a scenario in which ODIT could have prevented the release of an actual regression. Then, I give a high level walk-through applying the technique to the example program modification. Here and throughout this paper, I adopt the convention that $entity_0$ (where entity is a program element such as the whole program, function, variable, etc.) refers to the original version and $entity_1$ refers to the analogous entity in the updated version.

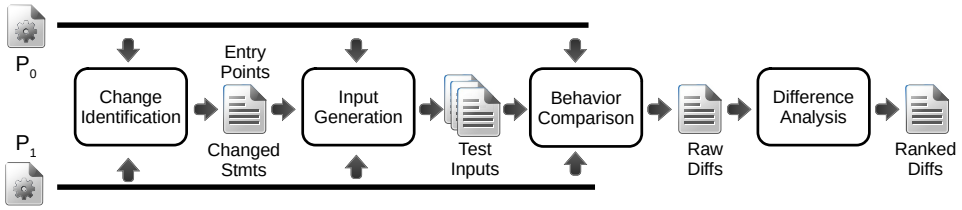


Figure 5.2: ODIT Overview

Listings 1 and 2 contain code taken directly from one of the benchmarks, CoREBench [34] regression number 22. The subject program is $expr$, a utility for expression evaluation. The first listing contains the original source code for a function $toarith_0$ to convert a value from a string to an integer representation. The second listing contains an updated version committed to the project's repository and released. Based upon subsequent bug reports, the developer likely intended $toarith_1$ to be equivalent to $toarith_0$, but the former can incorrectly convert strings representing negative numbers. Incidentally, the $toarith_1$ fault lies in duplicate sign adjustment is lines 13 and 26. The code modification introducing this regression changed a single function, $toarith$, without changing the $VALUE$ structure shown above the listings.

This scenario begins when the developer, Ridley, completes coding the update $expr_1$, but before committing it to the project source repository. Much as the actual developer probably did, Ridleyinspects the changes, runs some exploratory tests, and sees nothing amiss. For additional assurance, Ridleysubmits $expr_0$ and $expr_1$ to ODIT. A few minutes later, ODIT reports inputs to $toarith$ which result in different output values in v_0 and v_1. Using specific values provided by the technique, Ridleycan recognize and correct the problem with $toarith_1$ before committing to the project repository.

To detect internal behavioral differences, ODIT employs the four phases depicted in figure 5.2. *Change Identification* starts by recording additions, deletions, type, and implementation changes of global variables and functions. In the $expr$ example, no variables or functions were added, deleted, or changed; and only one function, $toarith$ has a modified implementation. This phase then identifies functions with equivalent signatures that can

reach modified functions in either version. It forwards a list of these functions as entry points, ordered by function-call distance from program changes, to the next phase. Since, the type of $toarith$'s parameter, $VALUE$, is unchanged, $toarith$ itself will be at the head. Note that had $VALUE$ equivalence not held (e.g. due to a field addition), then the list head would be a parent function in the program's call graph.

Input Generation then produces inputs for each entry point. Since this list only contains functions with equivalent signatures, all generated inputs for fn_0 can execute on fn_1, and conversely. To simplify this example, I only discuss the head of the entry point list, $toarith$. This phase first generates inputs for $toarith_0$ and $toarith_1$, discarding inputs that do not execute changed statements.

Behavioral Comparison identifies differences by concretely executing the generated inputs on both versions of the program, while accumulating sequences of program state snapshots at each function return. In the example above, there is a single function return during concrete execution and a single pair of program states for comparison, S_0 and S_1. For each state pair, the S_0 values for global variables, $toarith$ return value, and output parameters provide an oracle for comparison with S_1. Since the prior phase will generate multiple $VALUE$ inputs encoding negative strings, each such input will fail the field i comparison of $toarith$'s parameter.

The final phase, *Difference Analysis*, organizes the differences detected for developer consumption. Input generation may have produced many inputs triggering the same difference (e.g. "-1", "-2", etc.) In the example, this phase will report to the developer that there is a single behavioral difference effecting $toarith$'s output parameter, field i; along with a set of inputs reproducing it.

The $expr$ example is an actual regression error that ODIT could have prevented. $expr$ is also a good illustration of the advantages of over-approximate input generation. To reach the modified function, $toarith$, from program entry requires traversing complex, recursive, and looping control flow. Other top-down, dynamic symbolic execution based approaches

to differential testing may not reach the changed function.

For example, Shadow [31] did not generate any tests for this regression. In this case, the technique produced no false positives. But, under less ideal conditions, the reported differences will arise from both intentional and unintentional changes, infeasible inputs, and side effects of prior behavioral differences. The next section details the algorithm described in this section and my method for ordering and ranking the detected behavioral differences to emphasize the unintentional, but feasible ones.

5.3 Over-approximate Differential Test

Developers typically modify software to change program behavior, and the actual behavior of the update can be different from its expected behavior. These differences, which ODIT computes in terms of over-approximated differences between observable program values, are what I call unintended behavior. Figure 5.2 shows an overview of the ODIT technique. Its only inputs are an original version of a program P_0 and a modified version P_1. It does not require prior instrumentation of either version. ODIT outputs a list of behavioral differences, ranked in an order favoring unintended, feasible differences.

ODIT proceeds in four major phases: 1) *Change Identification*; 2) *Input Generation*; 3) *Behavior Comparison*; and 4) *Difference Analysis*, detailed in the following subsections.

5.3.1 Change Identification

From P_0 and P_1, *Change Identification* outputs a list of entry points for the next phase, *Input Generation*, and a list of modified code statements. Each entry point is a function defined in both P_0 and P_1 with an equivalent signature and that can reach changed code in either program's call graph; together with that function's call-graph distance to changed code. Note that there is always at least one such entry point, $main$. $main_0$ and $main_1$ will have equivalent signatures and, as the root of each program's call graph, will reach the changed code. ODIT takes advantage of other potential entry points closer to the actual

change.

Algorithm 3 (*Change Identification*) starts by identifying functions defined in both P_0 and P_1 (C, line 4), functions added (A, line 5), and functions deleted (D, line 6). Lines 7 and 8 of the algorithm then identify functions in C that are modified in P_1 (Δ), as well as the subset of Δ with a different function signature (Δ_S); that is, functions defined in both P_0 and P_1, but with non-equivalent parameters or return types. ODIT computes type equivalence of primitive types (e.g. int, float) by direct type comparison. Pointer types are equivalent if each points to equivalent types, and composite types (e.g. structures, arrays) are equivalent if all members are equivalent. Type decomposition for equivalence is essential since data types may have changed between P_0 and P_1. For example, if the example in Listings 1 and 2 contained data type $VALUE_0$ and a modified $VALUE_1$ with an additional field, then $toarith$ would not be a valid entry point. The loop in lines 10-12 iterates over every function f in C that have equivalent signatures. If there is a path in the static, context-insensitive call-graph of P_0 from f_0 to any changed or deleted function or there is a path in P_1 from f_1 to any changed or added function, then f is an entry point. Note that the constructed call-graphs are context-insensitive and only consider direct calls, i.e. the graph does not include calls that could occur through a function pointer. As a final step, the set of entry points are ordered by function-call distance to the closest changed code.

The final line 13 of the algorithm produces a list of changed statements for each common function in C. Both of *Change Identification* two outputs are inputs to the next phase.

5.3.2 Input Generation

ODIT leverages under-constrained symbolic execution (SE) [5] to generate function-level inputs. Traditional symbolic execution [2] executes a program with symbolic input instead of concrete values. Symbolic execution explores all feasible program paths by executing the program with *symbolic* external state. It maintains a set of symbolic states, each with

Algorithm 3: Change Identification

Input : P_0: original version of program P
P_1: updated version of program P
Output: E: ordered list of functions reaching D
Δ_i: list of changed program stmts

begin
  $F_0 \leftarrow \{f \in fns \mid f \in P_0\}$
  $F_1 \leftarrow \{f \in fns \mid f \in P_1\}$
  $C \leftarrow F_0 \cap F_1$
  $A \leftarrow F_1 \setminus F_0$
  $D \leftarrow F_0 \setminus F_1$
  $\Delta \leftarrow \{f \in C \mid f_0 \neq f_1\}$
  $\Delta_s \leftarrow \{f \in \Delta \mid sig(f_0) \neq sig(f_1)\}$
  $E \leftarrow \emptyset$
  foreach $f \in C \setminus \Delta_s$ **do**
    if $reaches(P_0, f, \Delta \cup D) \vee reaches(P_1, f, \Delta \cup A)$ **then**
      $E \leftarrow E \cup \{f\}$
  $\Delta_i \leftarrow \{s \in statements\ in\ P \mid s_0 \neq s_1\}$
  return $< E, \Delta_i >$

an associated logical conjunction, the path condition PC, accumulated during execution. The executor starts with a single state with a symbolic value for each of the program inputs and an empty PC. It then executes program statements in their normal execution order and with their original semantics, with the following two exceptions. 1) The executor computes statement values as symbolic values relative to the original symbolic input values. 2) The executor forks a single state S executing a branch statement with condition C into two states, S_T and S_F, with associated PCs $PC = PC \wedge C$ and $PC = PC \wedge \neg C$ respectively. The executor continues by selecting a state and executing its next statement. Executors work in conjunction with a constraint solver which can compute satisfying solutions to PCs. When the executor state executes a state's terminating instruction, the solver can then calculate program input values that will reproduce the state's path through the program. In theory, symbolic execution could provide inputs for complete path coverage. But, in practice path explosion [3] (the number of states is exponential in the number of branches), solver performance overhead, and required supporting theories required to solve PCs (e.g. floating-point) limit path coverage [3].

Starting from $main$, symbolic execution may never reach a targeted function before suffering path explosion [3] or arriving at an unsolvable constraint. Progressive symbolic execution (PSE) extends traditional symbolic execution by substituting symbolic values

for more than just program inputs. By substituting symbolic values for function arguments and global variables, under-constrained symbolic execution can begin execution from any function, not just program $main$. In languages supporting pointers, lazy initialization [6] accounts for an unconstrained pointer p by forking states for each of: 1) p = $NULL$, 2) p = a new allocation, 3) p = a pre-existing memory object(s). However, the consequence of this improved scalability is that under-constrained symbolic execution can produce *infeasible* inputs for a function f; that is, no program inputs executed from $main$ can call f with the generated inputs.

Input Generation constructs inputs for each of the entry points, within a configurable maximum function-call distance from changed code, as found by *Change Identification*. Selection of the distance value improves ODIT scalability and represents a trade-off in time vs precision. A large distance will provide more unchanged execution context before reaching the changed function, thereby potentially reducing the risk of infeasible inputs to the changed code. However, the additional entry points also increase technique overhead with diminishing benefit, as distant entry points are less likely to find inputs reaching changed code. In my experience with the benchmarks in section 5.5, a threshold function-call distance of 2 yielded good difference detection within a reasonable runtime and was selected for all of the experiments. I leave the selection and tuning of a 'best' value for a given program to future work.

For each selected entry point e, ODIT performs under-constrained symbolic execution for a configurable period of time on both e_0 and e_1—while only unconstraining global variables common to both programs—with equivalent data types and function arguments. Other global values are concretely assigned with their original initializer. Allocating more time to under-constrained symbolic execution may reach deeper into the program's call graph from e but is generally unnecessary, as the technique selects e for proximity to the program changes. For the experiments in Section 5.5, I used a 60-second timeout. For comparison with traditional symbolic execution, the KLEE [24] evaluation allocated 60

minutes for each of the coreutils. Since e was selected to have equivalent signatures, inputs generated for either e version can be executed on the other. The ODIT symbolic executor uses the statement difference detail provided by *Change Identification* to only persist inputs that execute a changed statement. This step filters out inputs that cannot demonstrate a difference. For example, consider the illustrative example in Listing 1. A generated $toarith$ parameter of type $integer$ executes the same statements in both versions, so ODIT discards the input. The technique generates inputs for e_0 and e_1 identically, with the single exception of crashing inputs. Since P_0 is the baseline behavior, a crashing e_0 input is presumed to be infeasible and discarded. However, a crashing e_1 input may be new behavior and is retained.

When the PSE symbolic executor completes a retained path from an entry point e, it obtains concrete, initial values for the path's unconstrained global variables and program arguments and persists them as a test input. When the symbolic executor's allotted timeout arrives, will likely still be in progress. Rather than discard potentially useful inputs, PSE's symbolic executor selects satisfying values for the path explored so far, and completes the execution concretely and persists test inputs as with prior completed paths.

For ODIT's PSE tool, I extended my prior implementation with two major specializations in addition to lazy initialization. Since loops often lead to path explosion [3], I set a maximum threshold on the number of states whose next program statement can be in any single loop body. When this threshold is exceeded, states within the loop body are inhibited from branching into multiple states. Instead of forking additional states at a branch statement, an inhibited state selects a single branch to follow. The executor will try to select a branch corresponding to a loop exit, if available. Otherwise, a random satisfiable branch is selected.

My other major extension to my PSE tool handles external functions, i.e. declared functions without a function definition or body. System calls and external library calls are common examples of external functions. All non-trivial programs (including the benchmarks)

generally contain at least one call to an external function. Since symbolically executing the native library code is impractical, symbolic executors often provide a simplified model for handling external calls. However, reliance on a system model can limit the accessible breadth of program behavior. For example, Palikareva *et al.* cite KLEE's incomplete environmental model as a contributing factor to several of Shadow's unsuccessful executions. To prevent false positives, their tools requires an under-approximating model that is currently manually constructed. In contrast, ODIT dynamically uses an over-approximating model by generating new, unconstrained values for the external function's output parameters and return value. These external values are treated as program inputs and concretized values are stored in the persisted test input for future execution replay.

This step completes the *Input Generation* phase. By construction, the generated tests are executable on both P_0 and P_1; and only contain inputs that execute modified code statements.

5.3.3 Behavior Comparison

This phase executes each generated test input on both P_0 and P_1, logging all raw behavioral differences found. Algorithm 4 defines the high-level algorithm for the *Behavioral Comparison* phase.

To execute a test input, (lines 4 and 6) ODIT utilizes its symbolic executor as a concrete executor. In this mode, the executor prepares an initial program state as in symbolic mode. But instead of creating symbolic state, it retrieves values for global variables and program arguments from the test input. If a value for a global variable is not available in the test input (added/deleted or changed data type), then its default initializer is used. Values for function arguments will always be present in the test input since they were generated only for functions with equivalent signatures. When the executor encounters a call to an external function during execution of a test input, it returns the call specific output parameters and return values specified within the test input.

Executing test inputs within the concrete executor permits a comprehensive and detailed view into program state at fine granularity and without instrumentation. The executor first runs the test input on P_0. During replay, the executor stores a sequence of P_0's memory address space snapshots; appending a new snapshot upon return from any functions common to both P_0 and P_1. If P_0 does not terminate the test input in a controlled fashion (e.g. entry function return, $exit$, $abort$), then ODIT discards the test input. Since P_0 defines baseline behavior, crashing inputs are presumed to result from infeasible test inputs (line 5). Upon execution completion (lines 4 and 6) $execute$ returns a result code, a sequence of state snapshots, and the execution's final state.

Because external function calls are also a source of behavioral differences, the executor records each external call executed. Recall that during *Input Generation*, the symbolic executor creates new, unconstrained values for the external function output parameters. During behavior comparison, the external function's input values can reflect a behavioral difference. For example, assume program P contains a function f that changes a file's access mode through the $chmod$ system call. If inputs exist that can cause f_1 to call $chmod$ with difference access control bits than f_0, then this could indicate a behavioral difference.

If P_0 executes the test inputs without crashing or timeout, the executor replays the test inputs on P_1 (line 6). If P_1 terminates differently than P_0 (e.g., P_0 aborts and P_1 exits), the executor logs the behavioral difference. Otherwise, the executor compares program states at selected pairs of snapshots, called checkpoints, from the two executions (lines 10-13).

The technique computes checkpoints by aligning pairs of snapshots by function return in both program versions. For example, suppose we have a P_0 and P_1 with functions $\{$ f, g, h $\}$ in common. Executing test input I on P_0 produces the snapshot sequence [f_0^0, g_0^0, h_0^0] and on P_1 produces the snapshot sequence [f_1^0, g_1^0, h_1^0], where a_n^m signifies the snapshot taken on the m-th return from version n (0,1) of function a. In this example, pairing snapshots for comparison is straightforward: $\{$ f_0^0, f_1^0 $\}$, $\{$ g_0^0, g_1^0 $\}$, and $\{$ h_0^0, h_1^0 $\}$. Aligning the memory snapshots into comparable pairs is more challenging when executing

the test input I on P_0 and P_1 result in different call sequences, S_0 and S_1. To pair snapshots in this general case, ODIT computes the longest common subsequence (LCS) between S_0 and S_1. That is, the longest sequence of snapshots (not necessarily consecutive) occurring in the same order as in S_0 and S_1.

ODIT conducts state comparisons (lines 13-14) as a series of type-aware value comparisons. $v_0 = v_1$ is a defined relation only if $type(v_0)$ is equivalent to $type(v_1)$. Therefore, if the type of a program element v (e.g., a function return type) has changed, the technique cannot compare v_0 and v_1 and will not log a detected difference. The technique defines value equality by type as follows: **primitive type** $\Rightarrow$ bit-wise comparison; **structure type** $\Rightarrow$ iterative comparison of member types; **array type** $\Rightarrow$ iterative comparison of consecutive type; **pointer type** $\Rightarrow$ recursive comparison of referenced type.

Pointers are equal if they point to equal memory objects. Dereferencing occurs as a depth-first-search of the memory graph to avoid cycles. I construct state comparison as a sequence of value comparisons: 1) the returning callee's return value; 2) the callee's output arguments; 3) global variables; 4) external call input parameters; 5) stdout and stderr.

For each detected difference, *Behavioral Comparison* logs three fields: 1) an annotated identifier, 2) the form of difference found, and 3) current call distance through unchanged functions. The identifier specifies the program element containing the detected difference. Possible program elements include global variables, function return values, function output parameters, and output streams. The technique adds annotations to identifier names to signify structure members, array element access, and pointer dereferencing. For primitive data types, the only possible form of difference is a bit-wise value difference, but other types can differ in other ways. For example, a pointer type may differ in value of dereferents (a standard value difference), one pointer may be NULL, or one pointer may lack a referent (i.e. a dangling pointer). The final field is used by the *Difference Analysis* 5.3.4 phase as a distance metric for ranking clustered differences.

At this point, ODIT has logged a list of raw differences discovered during *Behavior*

Algorithm 4: Behavior Comparison

```
   Input : P_0: original version of program P
           P_1: updated version of program P
           T: set of test inputs
   Output: DiffLog: log of raw differences found
1  begin
2      DiffLog ⇐ ∅
3      foreach t ∈ T do
4          r_0 ← execute(P_0, t)
5          if r_0 ∈ { return, exit, abort} then
6              r_1 ← execute(P_1, t)
7              if r_0 ≠ r_1 then
8                  DiffLog ⇐ termination diff
9              else
10                 S_0 ⇐ snapshots(r_0)
11                 S_1 ⇐ snapshots(r_1)
12                 foreach chkpt ∈ aligned(S_0, S_1) do
13                     DiffLog ⇐ cmp(S_0[chkpt], S_1[chkpt])
14                 DiffLog ⇐ cmp(final(r_0), final(r_1))
```

Comparison. The difference log will likely contain many entries immaterial to regression detection. It also may contain feasible, unintended behavioral differences threatening to become regression errors. The objective of the next phase is cluster, order, and rank the raw behavioral differences to emphasize the feasible, unintended changes.

5.3.4 Difference Analysis

The raw difference log produced by *Behavior Comparison* will contain redundant differences, infeasible differences, and intentional differences along with the unintentional regression errors. This phase clusters, orders, and ranks individual difference reports into an actionable list for developer consideration.

For each test input, the log contains a list of differences found in checkpoint state comparisons and the final state comparison. Each state comparison contains a list of value differences found. Each entry in this list records the differing program element's identifier, a descriptor for the form of difference, and the distance through unchanged functions calls prior to difference detection. The intuition behind this heuristic is that intentional changes tend to have immediate effect, whereas unintended changes are more likely farther from the site of the code update. For example, changing a function to use a more efficient sorting algorithm may have a localized detectable state difference. But at a distance, through

unchanged portions of the program, the state will be equivalent: a sorted list. However, if the updated sorting function contains a bug, execution distant from the change will still detect the list difference. The heuristic also favors feasible behavior. A common form of infeasible behavior arises from infeasible function inputs, such as an invalid pointer. If both the original and modified function react to the infeasible input in the same way (e.g., memory fault), then comparison finds no differences. And if they behave differently, the difference is likely to be immediately observable.

This phase first iterates over each test input with a failing comparison. For input, it collects the set of differing program elements along with the maximum distance at which the element difference was found during this input execution. The technique updates a co-occurrence matrix (an identifer$\otimes$identifer sparse array) accordingly. If program elements x_1 and y_1 both fail comparison to x_0 and y_0, respectively, in the same test, then co-occurrence[x, y] is incremented. When completed, co-occurrence allows ODIT to identify dependent differences. Conceptually, if every test input that detected a difference in x also detected a difference in y, then x is likely dependent on y. For example, consider a program with a function f returning an integer, and two functions a and b that both call f and store the returned value into variables ga and gb, respectively. Suppose f is modified and introduces a regression. Then *Input Generation* will produce test inputs for a, b, and f and *Behavior Comparison* will detect differences in f's return value, ga, and gb. The co-occurrence matrix allows ODIT to infer that the differences in ga and gb are dependent on f's return value. More formally, the co-occurrence relation provides a partial order over the set of element identifiers with detected differences.

Each element identifier in the raw comparison log is assigned a distance metric that is the mean of the distances in the test inputs for which it is logged. The co-occurrence dependency forms a graph whose roots correspond to independent differences detected. ODIT ranks the roots of the resulting lattice by decreasing distance for presentation to developers.

5.4 Limitations

ODIT targets *incremental* changes in a program. Having multiple unrelated changes with intertwined control flow can increase the number of reported root behavioral differences, affect difference ranking, and distort the co-occurrence relation. Three of the regressions in the empirical study originated from the same commit in which the developer, in an abundance of optimism, changed 65% of the program. Nevertheless, ODIT performed well on two of these three regressions.

Another limitation of ODIT concerns changes to a program element's type. If a function argument changes type between versions, the technique is unable to generate a test starting from that point. In this case, the technique looks for alternative entry points preceding the changed function in program's call-graph. If a functions' return type changes, the technique can generate a test input, but cannot compare the functions' return value. Similarly, the technique cannot compare a global variable with changed data type.

5.5 Empirical Evaluation

To evaluate ODIT, I implemented a prototype tool and performed an empirical evaluation on a set of benchmarks. In the evaluation, I addressed the following research questions:

RQ1: Can ODIT detect and effectively rank regressions?

RQ2: How do ODIT's over-approximating results compare to a similar tool's under-approximating results?

RQ3: How does ODIT perform on refactored, real-world code?

5.5.1 Implementation Details

I implemented ODIT's *Change Identification* phase as a clang [19] analysis pass and an LLVM [35] analysis pass. The clang analysis pass records declarations of external function constant parameters to limit unconstrained output parameters during *Input Generation*,

since this information is no longer available to the LLVM bitcode. The LLVM pass examines two bit-code modules to determine functions and global variables added, removed, or changed and constructs a mapping of comparable types between the two module's namespaces. To identify modified program statements, the LLVM pass iterates over each function fn. For each basic block in fn, ODIT computes a hash from the blocks sequence of instructions, named values, and constants. Hashing the basic block hashes in a depth-first-search decent of the fn's control flow graph computes a hash for fn. For the implementation of *Change Identification*, $hash(f_0) \neq hash(f_1)$, flags f as modified. Statement level changes are identified by matching block hashes between f_0 and f_1.

I forked the KLEE symbolic execution engine [24, 21] to implement ODIT's under-constrained symbolic execution for *Input Generation* and concrete execution for Behavior Comparison. During *Input Generation*, I used the STP constraint solver version 2.1.2. Finally, I developed the *Difference Analysis* phase in python3. My implementation of ODIT is publicly available together with the experiment data and infrastructure (https://sites.google.com/view/odit-ase2021/).

5.5.2 Evaluation Setup

To evaluate ODIT's ability to detect regressions, I considered potential benchmarks based on the following criteria: (1) an active development cycle with frequent repository commits; (2) availability of associated artifacts (e.g., error-introducing and error-fixing commits). Given these criteria, I selected a prior established set of regression defects CoREBench [34]. CoREBench contains a curated collection of real regression errors from four widely-used open-source projects, which provides realistically complex benchmarks for evaluating ODIT. CoREBench validated each of the regressions in their benchmarks, providing commit identifiers introducing the regression and bug reports describing the defect. For this preliminary study, I evaluated the 70 CoREBench regressions. I omitted regressions requiring 32-bit compilation, unsupported multibyte locales (e.g., ja_JP.sjis), and regressions reporting dif-

ferent effects but resulting from the same code modification. After discarding these, 61 CoREBench regressions remained. Table 5.1 lists the CoREBench programs, entailed regression identifiers, and average source code size.

Table 5.1: Regression benchmarks.

program	regressions	LOC
rm	1	1044
cut	3, 6, 12, 17, 21	519
tail	4, 5, 16	1039
seq	7, 8, 9, 18, 19, 20	254
cp	10	2498
ls	13, 14	3106
du	15	624
expr	22	583
find	23 - 37	8,738
grep	38 - 52	6,153
make	53 - 70	23,805

To answer **RQ1** and **RQ2**, I must be able to determine whether a behavior difference captured by ODIT relates to a specific regression fault. Although CoREBench [34] includes failing test cases gleaned from each regression's original bug report, their inputs are not suitable for my use. Each of these tests detects the regression's manifestation to external output. From the example in section 5.2, the provided regression test contains program arguments that cause $expr_1$ to print an incorrect result. However, in the under-constrained exploration of $expr$ behavior, a test input for $toarith$ will never execute to a print statement. To correctly evaluate ODIT, I need to determine when a value computed by $toarith$ would later manifest as the regression's error.

To this end, I use bug oracles as an approximated specification of the regression. A

bug oracle refers to an oracle that detects a specific incorrect behavior [36]. Thus, for each regression defect in the benchmark, I designed a bug oracle to detect the faulty behavior described in the bug report and addressed in the error-fixing commit. The bug oracles consist of a pair of regression identifier and a declarative expression (i.e. without side effects). When the expression evaluates to false, the specific identified regression will manifest. Then, execution blithely continues unaffected, since the expression is without side effects. If the expression never evaluated to false over the entire execution, then the bug did not occur. As an example of a bug oracle, consider listing 5.1. The oracle is the `o_assert` in line 31. The `o_assert` condition exactly matches bug expression, that is the bug will occur if and only if the oracle condition is **false**. This particular bug will not be expressed until much later, when *expr* prints the resulting miscalculated value. But, since ODIT only differentially executes in a neighborhood of the changed code (i.e. *toarith*), evaluation of ODIT to detect the regression requires an oracle close to the modified code. It is worth noting that *the bug oracles are only required as ground truth in the evaluation* but are not required by ODIT.

Introducing bug oracles into P_1's code would add control flow that would bias the input generation phase to produce inputs exercising the oracles themselves. Therefore, I implemented the bug oracles subject to conditional compilation to build both the original P_1 and the oracle instrumented P_1^O. After running *Input Generation* on P_0 and P_1, I performed *Behavior Comparison* on P_0 and P_1^O to provide an approximation of ground truth correlating with the specific benchmark's regression defect. The regression identifier distinguishes regressions when a single code update introduces multiple regressions (e.g. regressions 9, 18, and 20). During the *Behavior Comparison* phase, the executor maintains a list of signaled bug oracles, strictly for evaluation purposes. Therefore, the evaluation can not only determine whether ODIT found a true regression, but whether it found the specific regression addressed in the benchmark's bug report.

Throughout this section, I will refer to results that signal the corresponding bug oracle

as a *true positive*. If the oracle is not signaled, then the result is a *false positive*. Note that in this terminology, a difference originating from an intentional change, a different program defect, and an infeasible test input are all labeled as false positives. I do not know the developer's intent, have perfect program comprehension, or have a decidable algorithm to determine feasibility. But I do know if the bug oracle's condition was satisfied.

I performed the experiments on a server with dual Intel Xeon® E5-2650 processors running at 2.20 GHz with 48 cores, 128 GB DRAM, and running Ubuntu 20.04. I set a limit for ODIT of 60 minutes for each pair of code versions analyzed, and this limit was never reached in the experiments.

5.5.3 RQ1: Regression Detection and Ranking

Table 5.2: ODIT Experimental Results

bench	regression detection							cmp	
mark	inputs	diffs	+o	PPV	-o	FDR	rank	odit	shdw
01-rm	671	0	0	-	0	-	N/A	✗	✗
03-cut	30641	5	0	0.0%	5	100.0%	N/A	✗	✗
04-tail	11407	1	1	100.0%	0	0.0%	1	✓	✗
05-tail	8311	1	0	0.0%	1	100.0%	N/A	✗	✓[1]
06-cut	3198	5	2	40.0%	3	60.0%	1	✓	✓
07-seq	13427	2	1	50.0%	1	50.0%	1	✓	✗
08-seq	14088	3	1	33.3%	2	66.7%	2	✓	✗
09-seq	15248	2	2	100.0%	0	0.0%	1	✗	✗
10-cp	4239	0	0	-	0	-	N/A	✗	✓
12-cut	28606	7	3	42.9%	4	57.1%	3	✓	✓[2]
13-ls	13062	3	2	66.7%	1	33.3%	1	✓	✓
14-ls	10186	10	10	100.0%	0	0.0%	1	✓	✗
15-du	1402	10	8	80.0%	2	20.0%	1	✓	✗
16-tail	8296	1	0	0.0%	1	100.0%	N/A	✗	✓[1]
17-cut	28573	7	3	42.9%	4	57.1%	3	✓	✓[2]
18-seq	15248	2	2	100.0%	0	0.0%	1	✓	✗
19-seq	8533	3	1	33.3%	2	66.7%	3	✓	✗
20-seq	15250	2	2	100.0%	0	0.0%	1	✓	✗
21-cut	18841	11	11	100.0%	0	0.0%	1	✓	✓
22-expr	2644	1	1	100.0%	0	0.0%	1	✓	✗

Table 5.3: ODIT Experimental Results

bench	regression detection							cmp
mark	inputs	diffs	+o	PPV	-o	FDR	rank	odit
23-find	2552	67	1	1.5%	66	98.5%	67	✓
24-find	22994	3	0	0.0%	3	100.0%	N/A	✗
26-find	180975	65	10	15.4%	55	84.6%	1	✓
27-find	7771	1	0	0.0%	1	100.0%	N/A	✗
28-find	89420	4	0	0.0%	4	100.0%	N/A	✗
30-find	35945	7	7	100.0%	0	0.0%	1	✓
31-find	180873	64	10	15.6%	54	84.4%	1	✓
32-find	52459	9	2	22.2%	7	77.8%	1	✓
33-find	52268	9	2	22.2%	7	77.8%	1	✓
34-find	34835	7	0	0.0%	7	100.0%	N/A	✗
36-find	68074	4	0	0.0%	4	100.0%	N/A	✗
37-find	89012	2	2	100.0%	0	0.0%	1	✓
38-grep	4583	8	5	62.5%	3	37.5%	2	✓
41-grep	27704	51	0	0.0%	51	100.0%	N/A	✗
42-grep	2965	15	13	86.7%	2	13.3%	1	✓
44-grep	586	0	0	-	0	-	N/A	✗
45-grep	3142	1	0	0.0%	1	100.0%	N/A	✗
46-grep	9069	5	3	60.0%	2	40.0%	2	✓
47-grep	25758	22	0	0.0%	22	100.0%	N/A	✗
48-grep	25918	16	0	0.0%	16	100.0%	N/A	✗
49-grep	58	0	0	-	0	-	N/A	✗
51-grep	168	13	0	0.0%	13	100.0%	N/A	✗
52-grep	2012	3	3	100.0%	0	0.0%	1	✓

Table 5.2 and Table 5.3 present the quantitative results of my experiments. The leftmost column identifies the CoREBench [34] benchmark together with its CoREBench numerical identifier. I refer to the regression detection columns to answer **RQ1**. The first column of this group (inputs) reports the number of test inputs produced by *Input Generation*. The

subsequent columns detail the number of independent differences found (diffs), the number of true positive differences (+o), the positive predictive value (precision), the number of false positive differences (-o), the false discovery rate, and the highest ranked true positive. When a single update introduces multiple regressions, these values reflect differences coincident with any of the updates bug oracles.

In 19 of 43 benchmarks, the top ranked difference was a true positive, and 25 (58%) ranked near the top (i.e. at least third). ODIT detected behavioral differences in 39 of the benchmarks, with 29 reporting false positives. Although false positives are to be expected due to intentional behavioral changes and generated infeasible inputs, the technique produced no false positives on 14 regressions.

Inspection of the benchmarks in which ODIT either low-ranked or missed the regression difference revealed four general conditions limiting the technique's difference analysis, behavior comparison, and input generation:

First, the introduction of multiple, unrelated changes in the same code update impedes *Difference Analysis* and leads to a disproportionate number of false positives and low ranking true positives. For example, regressions 12 and 17 were both introduced in the same commit to cut. The two unrelated changes increase the number of differences detected, and the intermingled control flow prevent co-occurrence from detecting and ordering dependent differences. For future work, data flow analysis may help untangle co-occurrence.

Second, program updates to use different external APIs prevent *Behavior Comparison* from taking address space snapshots and performing state comparisons local to the changed code. For example, regression 5 (and 16) was introduced when $find$ transitioned file update detection from periodic polling to event notification ($inotify$). Differing external function calls unconstrain in-comparable variables. And differing API use methodologies (poll vs. event) significantly alter the function call sequence forcing relatively coarse-grained comparisons.

Third, program logic requiring the use of unsupported constraint solver theories limits

the ability of *Input Generation* to produce inputs with broad coverage. For example, seq (regressions 7, 8, 9, 18, 19, 20) uses many *float* data types. Since the solver lacks a theory of floating-point arithmetic, the executor must concretize symbolic floats on first operation, prohibiting future symbolic branching. This limits program paths considered and inputs generated.

Forth, program structure can severely restrict the available entry points for *Input Generation*. For example, both rm and cp (1 and 10) populate a configuration data structure in $main$ incorporating a detailed task description from command arguments, and then refer to this structure as a parameter in a chain of auxiliary functions. A program update that modifies this configuration structure renders all of these auxiliary functions nonequivalent types, and prevents input generation from anywhere but $main$. Similarly, $grep$ first constructs a deterministic finite automaton (DFA) and then supplies input to the DFA. Changes to the DFA structure limit $grep$ input generation to a close proximity to $main$.

Since ODIT reported a highly-ranked true positive in the majority of the benchmarks, the technique can detect and rank real-world regression errors.

5.5.4 RQ2: Comparative Results

In this subsection, I compare the results from ODIT's over-approximate approach to those of a related under-approximating tool, Shadow [31]. Both techniques use symbolic execution to explore differential program behavior and both tools are forks of KLEE [24]. But Shadow does not report infeasible behavioral differences. Shadow symbolically executes both programs in parallel, with the original version shadowing the updated execution. When the versions' paths diverge, Shadow combines path condition elements from both executions to construct inputs comprehensively covering the modified code. Fortunately for technique comparison, Palikareva *et al.* also conducted experiments on CoREBench [34] regressions.

The last column group in table 5.2 indicates the specific CoREBench regression errors

reported by each tool. A checkmark in the ODIT column indicates that the experiment detected the specific regression described by CoREBench. For example, regressions 12 and 17 are different defects, introduced in the same code update. My use of bug oracles during technique evaluation allow an association of a reported difference to a specific regression. This assures that ODIT not only found a regression, but both regressions.

Palikareva *et al.* conducted experiments on the core utilities in CoREBench (regressions 1 - 22). The authors combined results for regressions { 5, 16 } and {12, 17}, simply reporting bug detected. Since both of these code updates introduced multiple regressions, which specific regression was detected cannot be distinguished. Depending upon how the bug reports were combined, Shadow detected between 6 and 8 of the 20 core regressions. ODIT detected 14 of the same 20 regressions. Over-approximating behavior allowed ODIT to detect more errors than Shadow, but at the cost of potentially reporting infeasible differences.

Another advantage of /pse/ is reduced overhead. Shadow's median runtime of 9,175 seconds is much higher than ODIT's median of 335 seconds. Also, Shadow presumes a pre-existing test input that reaches the updated code and requires code instrumentation as annotations identifying the changed code. ODIT requires neither.

5.5.5 RQ3: Scability to Real-World Software

As discussed above, these experimental results were preliminary. Though the results were promising and suggest the viability of the ODIT approach, generalizing these results requires more diverse benchmarks. My derivation of ODIT's executor from KLEE [24] places some constraints on potential experiment subjects. The program's source language must be entirely in 'C', with minimal inline assembly. Enhancement to KLEE supporting other LLVM compiler frontends such as C++ and Rust is still a work in progress and has not been back-ported into ODIT.

I envision ODIT as a tool that developers routinely run before committing code to a repository, or even (automatically) every time the code is saved. However, the tool would

not be useful if it generated too many false alarms. In 5, I demonstrated that ODIT can (1) generate a low number of false positives and (2) rank the true positives above the false positives in most cases. In support of this approach, I present an experiment and an extended case study discussion of three real-world programs that are currently widely deployed.

In this subsection, I describe work supporting the scalability and utility of Over-approximate Differential Testing (ODIT) to real-world software projects. For this evaluation, I enhanced the prototype tool used in the experiments of chapter 5. In addition to the usual latent bug fixes, I made several enhancements to lower the overhead and improve the performance of the prototype tool.

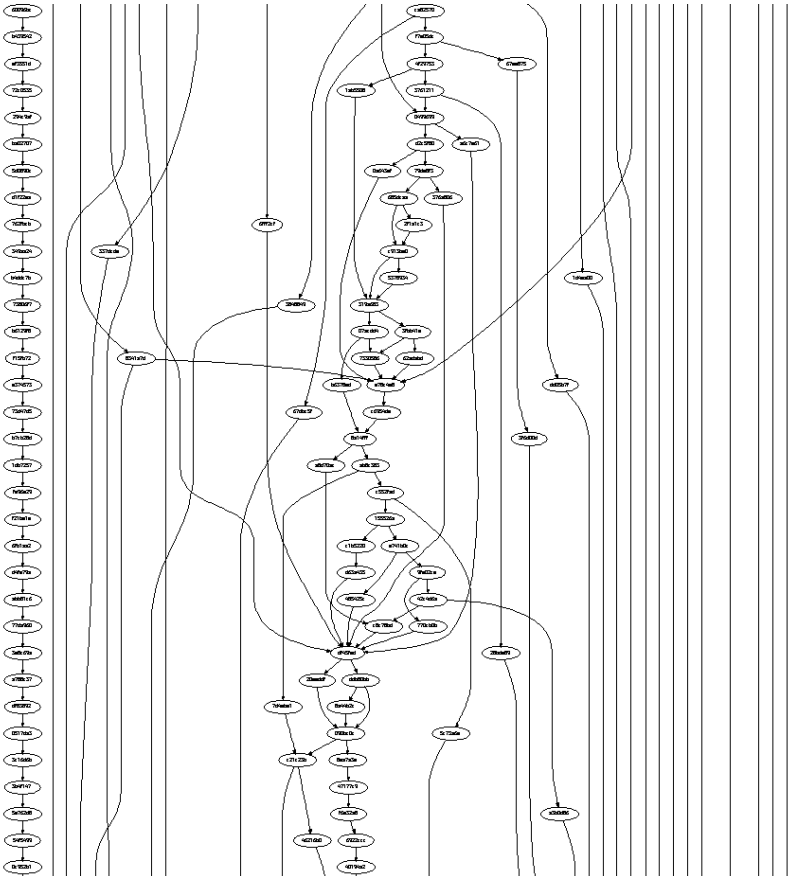
First, I extended the use of loop inhibition first described in chapter 5. To be self-contained, I will briefly describe the motivation and approach to this novel modification to standard Klee [21] state explosion mitigation. Klee starts with a configurable maximum threshold for the total amount of memory devoted to containing states for the program under test. If this threshold is exceeded, Klee randomly selects and terminates states until the surviving states are below the threshold. The issue with this scheme, is that exceeding the memory threshold generally occurs during a path explosion [3] such as from a symbolic condition within a loop body. Terminating a few percentages of these states still leaves the majority of them also within the loop. Often, then leading to another threshold violation since, with high probability, the next state selected will still be in the exploding loop body. Instead, the under-constrained [5] executor in ODIT tracks the states currently executing within each loop body. When the number of states executing in any single loop body exceed a configurable threshold, those states (and those only) are *fork-inhibited*. When a fork-inhibited state reaches a branch instruction: 1) if a loop-exit is a satisfiable branch, then it is selected, disregarding any other possible branch, 2) otherwise, if more than one branch is satisfiable, then one is selected at random. This approach has the advantage that states outside the loop body are more likely to be chosen and those within the body are biased to take the loop exit. To improve the performance of the ODIT prototype tool, I

extended fork-inhibition to a generalized condition that applies to all states, not just those in a loop body. When ODIT's symbolic executor reaches the timeout for these larger, real-world programs, often many thousands of states are still executing. Adding a short period of global fork-inhibition allows many of them to complete.

Since many potential benchmarks are multithreaded programs, PSE must model the use of the pthread library for threading and synchronization. Recall from chapter 5 that ODIT dynamically generates new, unconstrained values for all output parameters for each external library function. Since many of the functions in the pthread library take non-constant points for arguments, by default, ODIT treats these as function outputs and upon return from a call, supplies unconstrained values. When the mutex is embedded in the protected object (as is normally the case), whenever the program locks the mutex, ODIT's default behavior is to supply an unconstrained value for the entire data structure. In response, I implemented a system model for this library. Since multithreaded code is out of scope and unsupported by ODIT, the model for this class of library functions always succeeds, never waits, and constructs concrete return values for select output parameters, such as thread handles.

Next, I applied the enhanced tool to consider **RQ3**. To evaluate the extent to which ODIT would generate false alarms for code without regressions, I considered potential benchmarks with: (1) an active development cycle with frequent repository commits; and (2) a sequence of commits that consist of refactorings or small incremental changes. For a case study on ODIT scalability, I selected three programs satisfying these requirements: `redis` [37], `memcached` [38] and `lighttpd` [39]. Redis is an in-memory data structure store used as a database, cache, message broker, and streaming engine. Memcached is a distributed memory object cache generally used to optimize web service delivery. Multiple web servers can access and update the same memory object pool, reducing the load on backend data storage such as SQL servers. Lighttpd is a flexible web server optimized for performance, low overhead, and portability. As an aside, lighttpd was started while the original author was finishing their thesis on an unrelated topic; the existing `apache`

Figure 5.3: Example commit graph from Redis



servers were a bottleneck.

From the release history of these three programs, I selected sets of sequential release pairs for consideration. Each pair consisted of a program major release and its subsequent minor release. I selected relatively modern major releases to avoid build failures due to the evolution of `gnu autotools`. Because selecting commit difference pairs in a real-world program is not at straightforward as might be expected, I wrote a tool to extract them. The commit graph for complex software is often complex as well; without the simple branching of a development and subsequent merge back into a main branch. For example, consider the example from Redis in figure 5.3. In some cases, again e.g. Redis, the prior release need not even be an ancestor of a later release. The tool takes `begin` and `end` references and finds relevant pairs by:

1. Constructs a directed acyclic graph (DAG) from the commits,

2. Computes the sub-graph of ancestors of `end`,

3. Adjusts `begin` to the closest ancestor (maybe `begin` itself),

4. Enumerates all paths from `end` to `begin` in the commit graph,

5. Includes all edges in these paths whose leading node has a single parent (i.e. not a merge node) and whose diff contains a modified program source file,

For each of the three subject programs, I selected commit pairs between a major release and its next tagged release. Each subject program occupies a column in table 5.4. The left most sub-column contains the commit pair in the form `prev..post` and listed from latest to earliest pairs. For simplicity, henceforth I will refer to the pair by its `post` commit short identifier. For each commit, I manually inspected the log message and the modified source files. Though uncommon, I did find instances in which commit messages indicated comment changes only, whereas the commit difference contained additional behavioral changes. Through this inspection, I manually classified the extent of change in each commit as one of the following categories:

comment: change involving only comments.

refactor: behavior-preserving change to program logic.

behavior: behavior-modifying change to program logic.

data: change involving only data.

platform: change was conditionally dependent upon architecture or upon a non-default build option.

The right hand sub-column of table 5.4 identifies each commit's category.

Table 5.4: Case Study Subjects.

Redis		memcached		lighttpd	
5.0.1 ↘ 5.0.0	**category**	**1.6.1 ↘ 1.6.0**	**category**	**1.4.41 ↘ 1.4.40**	**category**
c801283..c595050	data	2168ac8..aac7d69	behavior	375022a..5863d05	data
4c4f50e..c801283	data	9a4aa77..2168ac8	platform	ebf3af8..375022a	comment
a7b46e0..4c4f50e	refactor	c557f1c..71d4fa8	behavior	acd5e45..ebf3af8	platform
80e129d..a7b46e0	data	b5ec478..c557f1c	data	558bfc4..acd5e45	behavior
88805cb..80e129d	behavior	9198a33..b5ec478	platform	f7410da..558bfc4	behavior
6b40273..88805cb	data	c838b56..9198a33	platform	ad6d418..f7410da	data
1c637de..6b40273	behavior	ab2ae12..6ca41fc	refactor	c8e647a..ad6d418	behavior
90b52fd..1c637de	data	936325b..ab2ae12	refactor	a62bff9..c8e647a	behavior
89cbb5d..90b52fd	refactor	**1.5.1 ↘ 1.5.0**		a69a803..a62bff9	behavior
175515c..89cbb5d	platform	bac46b3..d866123	behavior	a95aaa9..a69a803	behavior
3997dd6..175515c	refactor	6107b92..bac46b3	behavior	bce293e..a95aaa9	platform
bd80291..3997dd6	platform	67287ca..6107b92	behavior	565dec2..bce293e	behavior
4369cbc..bd80291	comment	d1f34d6..b2ea920	behavior	38139fa..565dec2	behavior
1ed821e..4369cbc	data	304b77e..d1f34d6	platform	9af58a9..38139fa	behavior
b49bcd0..1ed821e	behavior	78c260a..71b2385	platform	ed34089..9af58a9	behavior
09d1849..b49bcd0	refactor	3e8f5e2..78c260a	behavior	b43fc00..ed34089	behavior
bdf6306..09d1849	comment	**1.4.11 ↘ 1.4.10**		cd33554..b43fc00	behavior
50222af..bdf6306	behavior	595572c..016a87c	refactor	78c79ea..cb468d3	behavior
643ee6e..50222af	refactor	a16ce58..595572c	behavior	1ebc83f..78c79ea	behavior
8b609c9..643ee6e	behavior	96c07ae..a16ce58	behavior	d506f4a..779c133	behavior
2710260..8b609c9	behavior	f4983b2..3b96138	refactor	4d92046..d506f4a	data
a677923..2710260	behavior	324975c..f4983b2	behavior	a3ec906..4d92046	behavior
427e440..a677923	behavior	193a653..324975c	behavior	72abc87..a3ec906	refactor
28f9ca4..427e440	refactor	b3630e1..193a653	behavior	acad2c9..9c49dc9	refactor
4bf9efe..28f9ca4	refactor	8c1c18e..b3630e1	behavior	393dfd8..acad2c9	behavior
4fbd7a3..4bf9efe	behavior	99fc043..8c1c18e	behavior	adf9159..393dfd8	data
2480db5..4fbd7a3	comment	10698ba..99fc043	behavior	052a049..00cc4d7	behavior
e5e4d2e..2480db5	behavior	40b7b4b..10698ba	behavior	e9c9f42..2cdc017	behavior
713800d..e5e4d2e	comment	f58de2a..40b7b4b	behavior	8f8fa60..e9c9f42	behavior
e79ee26..713800d	behavior	7066273..f58de2a	behavior		
505cc70..e79ee26	behavior	ee486ab..7066273	refactor		
3c36561..505cc70	comment	7bc93a6..ee486ab	data		
3761582..3c36561	behavior				
edc47a3..3761582	refactor				
9872af6..edc47a3	refactor				
3f399c3..9872af6	comment				
eaaff62..3f399c3	behavior				
43ebb7e..eaaff62	comment				
de8fdaa..43ebb7e	behavior				
dc8f111..de8fdaa	refactor				

Refactored Code

Since **RQ3** focuses on refactoring, the table 5.5 details each individual refactoring commit. The leftmost column lists the subject program and the refactoring commit identifier. The second column shows the number of files changed and the number of source lines added/deleted. The third column contains shows the number of modified functions and global variables found during ODIT Change Identification. The final column shows the number of inputs produced by ODIT Input Generation and the total number of differences found by Behavior Comparison and Difference Analysis. Rather than burying the lede, the major significance is that ODIT found no differences in the refactored code, i.e. it produced no false-positives.

Table 5.5: ODIT refactoring commit results.

subject	Δfiles	Δlines	Δfn	ΔV	inputs	diffs
redis::4c4f50e	1	3	1	0	51	0
redis::90b52fd	1	3	0	0	0	0
redis::175515c	1	6	0	0	0	0
redis::b49bcd0	1	2	0	0	0	0
redis::50222af	1	1	15	0	2	0
redis::427e440	1	1	0	0	0	0
redis::28f9ca4	3	21	0	0	0	0
redis::3761582	1	23	1	0	0	0
redis::edc47a3	1	61	12	0	9443	0
redis::de8fdaa	1	12	1	0	847	0
memcached::6ca41fc	2	4	1	0	0	0
memcached::ab2ae12	1	2	1	0	0	0
memcached::016a87c	1	6	0	0	0	0
memcached::3b96138	4	56	13	0	38422	0
memcached::40b7b4b	1	56	1	0	14916	0
memcached::f58de2a	1	18	1	0	918	0
memcached::7066273	1	8	1	0	909	0
lighttpd::a3ec906	1	5	0	0	0	0
lighttpd::9c49dc9	1	14	0	0	0	0

In the rest of this section, I discuss each of these refactoring commits, which are the focus of **RQ3**, and leave discussion of the other categories to section 5.5.5. ODIT found that seven of the refactorings contained no differences in the program's functions or global variables. Recall from chapter 5 that the Difference Analysis phase calculates hashes over the LLVM intermediate representation (IR) of each basic block and over the control flow

graph (CFG) of each function. Therefore, if no functions or global variables have been added or removed, no global variable changed type, and no function evaluates to a different hash, then there are no IR statements to target for Input Generation and ODIT completes while reporting no differences found, skipping Input Generation, Behavior Comparison, and Difference Analysis.

From these 8 refactorings:

redis::90b52fd removed an unnecessary local variable, `keys_in_multiple_nodes`.

redis::175515c simplified an if statement by incorporating a prefix increment operator.

redis::b49bcd0 changed a local variable's type from `time_t` to `mstime_t`, but both types resolved to the same underlying LLVM IR data type.

redis::427e440 inserted an additional header file. Due to C coding practices which often employ the preprocessor to redefine types and functions, the equivalence of the resulting object code is non-trivially determined.

redis::28f9ca4 replaces the read of a global variable with an architecture independent function. Under a linux build, this new function simply returns the value of prior the global variable. Since the compiler optimized this new function call away, the resulting IR is identical.

memcached::016a87c removed an unused parameter from the `spawn_and_wait` function.

lighttpd::a3ec906 added a build configuration dependent include file

lighttpd::9c49dc9 refactored an if..else chain throwing a compiler warning, in dead code subsequently discarded by the compiler.

For six of the refactoring, ODIT identified differences and generated inputs, in some cases in large numbers, without reporting any false positives:

redis::4c4f50e reversed the operands of a logical AND in an if condition. Although this modification may seem obviously equivalent, one of the logical operands is function call which could have side effects. If so, then the updated program could exhibit a behavior difference due to boolean shortcutting.

redis::50222af moved the initialization of a global variable, without changing its value.

redis::edc47a3 hoisted a fragment of common functionality into a new function with modified semantics and modified the donor site and other impacted sites for the new function. The significance of this update is reflected in the number of modified source lines (61), modified function bodies (12), and the number of tests generated (6,681).

redis::de8fdaa simplified the control flow of `msetGenericCommand`. The update eliminates a used, but algorithmically not required, variable.

memcached::3b96138 replaced memcached's object reference counting mechanism used by the server for automatic memory object garbage collection. This was also an invasive update, generating 51,758 test inputs.

memcached::40b7b4b was a complex refactoring intended to simplify the convoluted control flow of `do_item_alloc`.

memcached::f58de2a was a continued refactoring intended to further simplify the convoluted control flow of `do_item_alloc`. The equivalence of the updated code is not obvious, and could easily have introduced a fault.

memcached::7066273 relocated a mutex unlock to an earlier point prior to function exit, freeing other threads earlier. Although the change is small, it is located at the end of a function with significant flow control.

For commits redis::3761582 and memcached::6ca41fc and ODIT was unable to generate inputs that reached an updated program statement before timeout. Memcached::ab2ae12

is an interesting case that can likely be labeled as a true-positive unintended behavioral change. The git log message for this commit is ‘‘Add stdio.h,stddef.h to storage.c” Manual inspection confirms that the only change to program source files was the insertion of these two include files; which should have no impact on the `Linux` build as the inclusion was not required on that platform. Based upon the log comments and the inspection, I originally categorized this commit as a refactor. However, ODIT reported two behavioral differences. Upon manual inspection of the generated LLVM IR, I found that there had been a change in memcached's `autotools` configuration generator that modified the format of its version string. Although ODIT does not target data differences during change identification, the change in string length was detected as a parameter to the construction of the response to an API server version query. The prototype tool reported the difference found in the data packet returned by `dispatch_bin_command`. Since the behavioral change was not mentioned in the commit message, I infer that the developer was probably either unaware or unconcerned of build system side effects.

As a trial to determine whether ODIT could generate difference reaching inputs for redis::3761582 and memcached::6ca41fc if allocated more time, I re-ran input generation for these two commits with a 10-minute timeout per entry point, instead of 1-minute as used as a baseline. But, even with an order-of-magnitude more time, ODIT found no difference reaching inputs.

These results, together with **RQ1**, provide initial evident that ODIT can be effective in detecting regressions without overwhelming the user with a large number of false positives. In the next section, I discuss the other commit categories found in the benchmarks.

Case Study Discussion

This section presents a case study discussion of the remaining benchmark commit categories: *comment*, *data*, *platform*, and *behavior*. Discussion for the first three of these will be brief. Commits in the *comment* category contained only source code changes to program

comments. Even though they are behavior preserving, they cannot be considered a refactor since there is no restructuring of the existing source code. For all three benchmarks, the ODIT Change Identification phase correctly detected that these commits contained no modified functions or global variables and did not proceed to Input Generation. Note that the ODIT Change Identification alone would have been useful for a couple commits whose git log message indicated they were comment only, but additionally contained some minor behavioral code changes as well. During my manual inspection, I categorized these as *behavior*.

Commits in the *data* category contained only source code changes to data values, with no modified program functions. ODIT does not support this category of change since Input Generation targets changed IR instructions. And a data-only update will not modify any program statements. I manually inspected the eleven commits in table 5.4 in the *data* category. In all of these instances, the data change did not modify program behavior. Rather, these changes modified version string, help messages, command option names, or revised program message phraseology. However, program behavior can certainly be encoded in data. For example, Redis encodes its server commands within a lookup table mapping command name to a function for dispatch. Additional fields in the lookup table describe other steps to be taken before and/or after the indirect function call. So data determines the behavior around the dispatched command. I plan to address this limitation in future work by starting with a data flow analysis from modified data to locate affected program functions.

Commits in the *platform* category contained behavioral changes in source code that was conditionally excluded from building in my experiment setup. For example, memcached::b2ea920 pertained to ARM builds, and did not produce a difference in the compiled LLVM IR. Similarly, lighttpd::ebf3af8 modified an assertion, which were disabled in the default build, and lighttpd::a95aaa9 is conditioned on a build option that defaults to disable. All commits in this category produced no differences in generated functions.

Commits in the *behavior* category will consume the remainder of this section. Recall from chapter 5 that ODIT input generation can be constrained by the subject program's architecture. I provide more detail on this limitation here for clarity and background to the ensuing discussion. For reference, see a high-level representation of the architectural style that limited input generation for `rm`, `cp`, and `grep` in listing 5.2 In this style, the program is segmented into two major components: a parser and an applicator. The parser reads a textual specification of the operations/transformations to be performed and returns a validated binary configuration representing the operation to be applied. The applicator then applies the operation to the input, passing the operation down to a series of helper functions. For `rm` and `cp`, the structure specifies the file operations to be performed. For `grep`, the structure contains a Deterministic Finite Automaton (DFA) constructed from the regular expression argument. As a result, any modification to supported file operations that change this structure or a modification to the DFA representation prevents ODIT from comparing the memory objects returned by `parse`. Of even more consequence, Input Generation can only start at `main` since operation_0 and operation_1 are of differing types. Therefore, for this style of architecture and these types of modifications, ODIT can only perform the equivalent of classical symbolic execution.

One way to greatly mitigate this limitation would be to build a mapping between the fields of operation_0 and operation_1. Unfortunately, a structure in LLVM IR is a sequence of anonymous data types, so the mapping is not explicit. Such a field mapping could be built by heuristics from field utilization in the original and modified program, or could also be constructed from IR debug metadata. Requiring a debug build would likely be acceptable to an ODIT user, since the technique requires source code anyway.

Two other Redis commits just prior to the 5.0.0 release, 9714bba26 and a3fb28edc, pose another challenge for ODIT. In my prior work [40], ODIT was unable to generate any test inputs for either. Manual inspection revealed another architectural complication. Both of these commits added a new client command to the Redis server, while making

no other change. Since Redis dispatches client commands by matching a request string to an offset into a table of function pointers, ODIT's Change Identification only found some new functions and ignored one data element addition. And none of the new functions where reachable through the updated program's static call graph from a function present in the original program. Hence, there is no valid starting point for Input Generation.

The commits within the behavior category fall into one of three categories: 1) inputs generated, and behavioral differences found; 2) inputs generated, but behavioral differences not found; and 3) no inputs generated. Next, I discuss each of these categories in turn.

For each of the commits in which ODITreported behavioral differences, the difference could be a) a true-positive, unintended behavioral difference; b) a false-positive intentional behavior change; or c) a false-positive, infeasible difference. Note that a true-positive does not necessarily indicate a regression, but rather a significant potential for one. The developer may not have been aware of all the side effects of the modification, but upon examination could determine that the unexpected effects are benign. For example, a function update that leaves a different value in a global variable that is not accessed again during the remaining program execution is still a true-positive, if the developer did not realize the effect at the time of coding. In fact, a precise ground truth to distinguish between the a) and b) possibilities would require insight into the developer's contemporary program comprehension. Although user studies could be used to estimate the proportion of intentional vs unintentional differences, the ratio likely varies greatly with the subject program. Moreover, that information alone would not be useful to determine the intentionality of a particular difference. I considered, but ultimately rejected, multiple proxies for developer intentionality. For example, a difference that was later identified with a program bug can be presumed to have been unintentional. But this proxy omits all unintentional but benign differences as well as bug fixes that were not logged to an issue tracking system. As a practical matter, even when a bug is logged and tracked, the benchmark's developers recorded what they changed to correct the bug, not the commit introducing the bug. Finding the

introducing commit for a specific defect is certainly possible, as benchmark suites such as CoREBench [34] have done. However, these suites only considered a limited number of identified bugs. To serve as a proxy for unintended behavior, I would have to consider all issues identified in the bug tracking system for potential origin in one of the benchmark commits. This approach is clearly impractical. I also considered using modified test cases as a proxy for intentional behavior changes. Presumably, an intentional program behavioral change would be accompanied by a corresponding update to the program's regression test suite to validate the change. However, none of the evaluated repository commits contained both source code changes and test code changes. The intentional behavior changes were either untested, had no impact on existing test cases, or test cases were modified as a part of a different commit.

The remaining possibility for a reported behavioral difference is infeasibility. As discussed in chapter 5, an infeasible difference is one that is only observed while executing infeasible inputs; that is inputs producing a system state that cannot be realized by any top-down execution from program entry (i.e. main). For example, an update that removes the null-pointer check on an input from a utility function that is only called with a valid pointer may produce an infeasible difference. The neighborhood of the modified function may be insufficient to establish the impossibility of a null pointer value. A program analyst could simply increase the size of the neighborhood, but at the cost of additional analysis overhead and risk that input generation may not reach the modified code. Additional symbolic execution from program entry to reach the desired neighborhood is also unlikely; if traditional symbolic execution could find inputs reaching the boundary, then it could also find inputs reaching the changed code. In general, the path explosion problem [3] is resistant to divide-and-conquer approaches. Perhaps other analysis techniques could detect the infeasibility and eliminate some false-positives. For example, a static pointer analysis may be able to determine that the hypothetical function argument above can never be null.

Table 5.6: ODIT behavior commit results, no reported differences.

subject	inputs	diffs	comment
redis::1ed821e	149	0	Fix XCLAIM missing entry bug.
redis::a677923	1028	0	asyncCloseClientOnOutputBufferLimitReached(): don't free fake clients.
redis::3f399c3	8	0	migrate: fix mismatch of RESTORE reply when some keys have expired.
memcached::595572c	384	0	fix 'age' stat for stats items
memcached::324975c	204	0	fix braindead linked list fail
memcached::193a653	37035	0	close some idiotic race conditions

Table 5.6 lists the subject commits, number of tests generated, and commit comment for those commits with a behavioral difference and for which input generation reached modified code, but for whom behavior comparison did not find any differences.

Manual inspection of the source code differences and inputs generated revealed the following observations:

redis::1ed821e introduced a new local variable to a single modified function, `xclaimCommand`. Since the new variable resulted in a modified function preamble, ODIT recognized all generated inputs as reaching modified code. However, none of these inputs reached the function's modified behavior, near the end of the function.

redis::a677923 prevents the freeing of fake client connections. ODIT generated inputs reaching modified function statements, but detected no changed program behavior as the function does not return a value and does not modify any global program variables.

redis::3f399c3 is similar to redis::1ed821e.

memcached::595572c corrects an error in which memcached reported `age` statistics as a date instead of an interval. This commit is an excellent example of a behavioral difference missed by ODIT due to consideration of inputs in a close neighborhood of the program change. The modified function is a leaf in a stack of functions for

reporting statistics, each of which accepts and passes down a function pointer that physically writes the formatted statistic. With this architecture, no return values or global variables are written close to the function change; just lazily initialized internal values. Farther execution context would be required to observe the value differences written to global state.

memcached::193a653 expanded the scope of a mutex synchronization lock to avoid a data race condition. Since ODIT models these functions (`mutex_lock` and `mutex_unlock`) as always successful and without side effects, the relocated lock has no observable effect.

Table 5.7: ODIT behavior commit results, no generated inputs.

subject	inputs	diffs	comment
redis::643ee6e	0	0	When replica kills a pending RDB save during SYNC, log it.
redis::8b609c9	0	0	Move child termination to readSyncBulkPayload
redis::2710260	0	0	Prevent RDB autosave from overwriting full resync results
redis::4bf9efe	0	0	MULTI: OOM err if cannot free enough memory in MULTI/EXEC context
redis::2480db5	0	0	Plugs a potential underflow
redis::713800d	0	0	if we read a expired key, misses++
redis::e79ee26	0	0	Fix XRANGE COUNT option for value of 0.
redis::3c36561	0	0	Overhead is the allocated size of the AOF buffer, not its length
redis::43ebb7e	0	0	several typos fixed, optimize MSETNX to avoid unnecessary loop
memcached::b2ea920	0	0	fix null pointer ref in logger for bin update cmd
lighttpd::acd5e45	0	0	[security] disable stat_cache if !follow-symlink (fixes #2724)
lighttpd::ed34089	0	0	do not set REDIRECT_URI in mod_magnet, mod_rewrite (#2738)
lighttpd::b43fc00	0	0	[mod_status] show keep-alive status w/ text output (fixes #2740)
lighttpd::cb468d3	0	0	[core] stay in CON_STATE_CLOSE until done with req
lighttpd::779c133	0	0	[security] do not emit HTTP_PROXY to CGI env
lighttpd::00cc4d7	0	0	[mod_auth] fix Digest auth to be better than Basic (fixes #1844)
lighttpd::2cdc017	0	0	[config] inherit server.use-ipv6 and server.set-v6only (fixes #678)

Table 5.7 lists the commits for which ODIT generated no inputs in the first column and the corresponding commit comment in the last column. In these cases, PSE was unable to symbolically execute from an entry point to reach a changed program instruction. Often, the reason may simply be the same factors limiting the scalability of classical symbolic execution as well, state explosion and solver limitations. For example in lighttpd::cb468d3, the update only modified 10 lines of source code, but this lines occur near the bottom of a 1,000 line `main`. This essentially requires the traversal of a complete program execution to reach the first modified program statement.

Additionally, manual inspection of these code changes also revealed other factors lim-

iting PSE's ability to generate inputs. For example, lighttpd::acd5e45 introduced a change to a global configuration structure. Since the data type of the configuration changed, ODIT was unable to unconstrain the global variable containing the system configuration. As a result, ODIT could only explore behavior with defaulted configuration values. As reported by Shadow [31] authors, another limiting factor is that some commits require a precise environmental model. In redis::4bf9efe, memory allocation failures are required to trigger the modified behavior. Although ODIT's environmental model could be augmented to provide allocation failures to cover this scenario, the additional overhead of modeling the potential failure of every allocation would likely exceed the limited benefit.

Finally, static type casting also limits the ability of PSE to reach changed code. A common C programming language idiom uses `void` pointers as a form of data hiding. Listing 5.3 contains the beginning of the modified function from lighttpd::b43fc00 employing this idiom. The parameter `p_d` is declared to be of type `void*` in line 3 and compiles to an `i8*` (i.e. a byte pointer) in LLVM IR. During input generation, PSE lazily initializes pointer values as an array of base types to accommodate `C` arrays, which are generally interchangeable with pointers. With the selected PSE defaults used in these experiments, `p_d` would point to an unconstrained symbolic array of eight bytes. Line 4 then assigns the address of this array to a pointer `p` to type `plugin_data` and Line 6 dereferences a field of `p` beyond the allocated size. Since the dereference is invalid, it blocks execution of the succeeding modified code. To give a sense of the scale of type casting, lighttpd::b43fc00 contains 629 bitcasts either from or to an `i8*`.

With the presentation of these results, I conclude this case study into applications of the prototype ODIT tool to real-world software.

5.6 Threats to Validity

External validity. The benchmarks consisted of 43 regressions found in 10 versions of a program and 30 refactored versions of a second program. Although the results of this

preliminary study are encouraging, generalizability requires further experiments with more, and more diverse, subjects.

Construct validity. The main risk has to do with the soundness and precision of my bug oracles, which dynamically determine if a specific regression has occurred during an execution. They are challenging to write since they must be declarative and located close to the defect location. To mitigate this risk, I conducted extensive testing of the bug oracles.

Internal validity. The bug oracles also constitute a threat to internal validity, because the additional control flow could influence input generation around the regression, favoring defect detection. To mitigate this risk, I constructed the oracles for conditional compilation (see Section5.5.2).

5.7 ODIT in Practice

I envision ODIT used in a practitioner environment as a pre-commit verification or a post-commit code review. Either way, successful integration into developer workflow presumes some standardization of version control utilization. In this section, I outline a set of version control best-practices that are conducive to continuous differential testing in general and ODIT in particular. These recommendations are not exclusive to differential testing, but also reflect commonly accepted best-practice with `git` (e.g. see [41])

Atomic Commits: Each commit should contain a single unit of work that cannot be decomposed into a sequence of smaller units. This practice allows the developer to examine each intended behavioral change in isolation to more easily recognize unintended effects.

Refactoring Commits: Related to atomic commits, developers should avoid combining code refactoring along with either feature addition or bug correction in a single commit. By definition, refactoring should preserve behavior and should not detect any behavioral differences. Including intentional behavioral differences in the differential

analysis could obscure an unintended refactoring side effect from a developer.

Mixed-Behavior Commits: Also related to atomic commits, developers should avoid combining bug-fixing code with new feature code in a single commit. While evaluating ODIT results, developers must reason about the concrete effects of their behavioral changes to distinguish the unintentional behaviors. Simultaneously reasoning about corrected behavior and newly defined behavior unnecessarily increases developer cognitive load.

Frequent Commits: As a result of the above, commits should be frequent. Developers can more readily recognize ODIT reported unintended behavior over small, incremental change.

Per-Feature Branches: Development of each new feature should occur in a dedicated feature branch rather than committed directly to a main/release branch. Intermingling the behavioral differences between unrelated code modifications reduces the ability of ODIT's difference analysis phase to recognize dependent differences and rank to emphasize the unintended differences.

Frequent Branches: Conversely, a sequence of related software commits should be committed to a development branch, then merged into main/release branches. So constituted, each branch identifies a sequence of atomic code modifications whose composition realizes the branch's feature.

Exclusive Branches: If a new feature is sufficiently complex, multiple developers may be tasked with its implementation. In this case, developers may have difficulty recognizing unintentional behavior due to modifications made by other developers. Mitigations for this effect include *i.* decomposing the feature into single-developer sized sub-features or *ii.* creating developer branches to be merged into the feature branch. In either approach, each branch contains the work of a single developer.

No Breaking-Builds: Although commits that introduce build failures are generally bad already, breaking-build commits will prevent any ODIT analysis since the tool requires both the original and updated program.

The above recommendations for version control best-practices are not particular to ODIT. For example, most of these practices are included in `GitFlow` [42].

5.8 Applying ODIT Approach to Other Development Environments

In this section, I discuss the potential, future application of my techniques, Progressive Symbolic Execution (PSE) and Over-approximate Differential Testing (ODIT), to other programming languages and paradigms. At least conceptually, classic symbolic execution as described by King [2] can be applied to any program. However, significant engineering effort may be required to realize a practical, scalable tool for any given combination of programming language and operating context (e.g. the operating system or other intermediate framework). The tool must not only support the language itself, but also be able to either symbolically execute its runtime libraries or include a model of the runtime behavior. Similarly, PSE could be applied to any program, beginning at any arbitrary program statement and assuming a completely symbolic program state. But in practice, PSE tool leverages many language design features to limit the unconstrained state that need be explored. For example, a function must contain a stack frame upon entry and may declare a returned value. And the set of defined global variables partitions the total program state into discrete elements. In the following subsections, I consider language features such as these and their impact on PSE and ODIT scalability.

5.8.1 Statically, Singleton Typed Languages

In a statically typed language, every program term has either an explicit or inferred type at compile time which determines how an object should be stored in memory and the operations that are permitted upon it. By singleton, I mean that each term can be mapped to

a single type. An example of languages violating this property would include those supporting object inheritance, since a reference to an object of base type could also refer to a derived type. PSE uses the static type system to set the initial symbolic state of global variables and function parameters as well as to lazily initialize unconstrained pointer variables. Without static types for each variable, PSE would have to assume a maximum size for each variable and allocate storage for the worst case scenario. Similarly, lazy initialization could still occur on dereference of a variable, but PSE would have to again assume a worst-case allocation for the indirect storage. Since the type of each program element is a single discrete type, the PSE executor knows the exact size of each element and its allowed operational semantics.

Since ODIT uses PSE for input generation, the above naturally applies to it as well. But, in addition, ODIT uses both of these language characteristics during state comparison. A data element's type determines the semantics of equivalence checking and by having only a single type, only a single equivalence check for each variable is required. Constructing a PSE or ODIT tool for any language with these characteristics would be conceptually straightforward; though potentially requiring significant engineering effort. Support for a language with an LLVM frontend could even use the publicly available PSE prototype tool to symbolically execute the generated program IR bitcode. However, the language runtime must be either executed by the executor or modeled. Heavy-weight runtimes such as the garbage collector in `go` and `rust` would likely require modeling to avoid over-reporting of infeasible behaviors.

Although dependence upon these language characteristics improves the scalability of both techniques, language features that enable a program to violate either can result in PSE's inability to generate inputs or ODIT's reporting of false-positive and false-negative behavioral differences. As discussed in the prior subsection, type casting can impair or completely prevent input generation. For example, casting a lazily initialized `void *` to a `struct *` will fail if the symbolic storage for the dereferent is too small to contain

the struct. And even if the struct happens to fit in the allocated space, a cast back to `void *` will defeat state comparison since ODIT will no longer know the semantics of type equivalence. In this case, ODIT will fall back to a sequential byte comparison which could result in either a false-positive or a false-negative.

5.8.2 Statically, Multiple Typed Languages

Contrasting to a singleton typed, with a multiple typed language, a program element can refer to more than one data type. An object-oriented language is a common example, due to class inheritance. If class `D` is derived from a base class `B`, then an object of class `D` can be used as a referent expecting a type `B`. Along a top-down execution from program entry, each program path will dynamically resolve to a single type ((B) or (D)). But, PSE may begin symbolic execution in an over-approximate state from an interior point in the program. If symbolic execution accesses an unconstrained global variable or begins at a function with an input parameter reference of type `B`, then lazy initialization [6] must explore the behavior of not only objects of type `B`, but also must include all derived types. Since a derived type can substitute different object method behaviors (such as through `c++` v-tables), an exploration of all behavioral differences must also include all potential object types. This multiplicity can occur in three different situations during PSE: 1) lazy initialization to a new object; 2) lazy initialization to an existing object, and 3) an output object reference from an unconstraining stub.

Moreover, object-oriented languages are not the only ones with the multiple type characteristic. Although neither `go` nor `rust` strictly support object inheritance, `go` interfaces and `rust` traits allow for a reference to multiple types. As with inheritance, each static type implementing the interface must be considered for a complete exploration of behavioral differences. The amount of overhead incurred by this extra lazy initialization will vary with implementation details. Recall that symbolic execution overhead is generally exponential in the number of branches. As a single input parameter to the neighborhood,

multi-type initialization overhead would be a constant factor. But, if incurred from an unconstraining stub, it is potentially exponential in the number of stub invocations.

5.8.3 Dynamically Typed Languages

In the above discussion of PSE application to additional program languages, *type* played a central role. Predictably, dynamically typed languages are particularly challenging to the scalability of PSE. Given concrete program inputs, an execution from program entry along a path will only access single, specific types. However, from an interior program point such as a function entry, PSE has no language hints as to a program element's type for lazy initialization. Given a function $foo(a)$, how should a be initialized? For existing element initialization, PSE would have to consider execution with a set of each of all existing memory objects. Initializing a as a new object is even worse. At a minimum, this would have to include every type of object observed so far. However, this is an under-approximation as dynamically typed languages generally also dynamically declare them. Therefore, all types available to the program may not have been observed thus far. Continuing in this vein, languages such as `python` dynamically construct and modify types (i.e. classes). Consequently, the behavior of a python object's foo method could change between consecutive invocations. As with the multi-typed languages discussed in the previous section, this type-uncertainty is repeated at every invocation of an unconstraining stub. In summary, applying PSE to dynamically typed languages is conceptually possible. Though without significantly simplifying assumptions to replace the type implicit constraints available to statically typed languages, symbolic execution may suffer a path explosion before even getting properly started.

5.9 Conclusion

This chapter presented ODIT, a novel technique for detecting potential regressions during software evolution. ODIT (1) leverages under-constrained symbolic execution to generate

function-level tests that traverse changed code, (2) executes the generated tests concretely to detect behavioral differences between old and new code, and (3) performs clustering and ranking of the detected differences to report them to developers in a useful presentation. The evaluation of ODIT, performed on real programs and real regression errors, provides initial evidence that the technique can help developers automatically identify regressions without overwhelming them with false positives, and improves on the state of the art in automated program behavioral differencing. This chapter also introduced experiments and case studies supporting the scalability of ODIT to real-world, current, and actively-maintained programs. I applied ODIT to sequential commits to `redis`, `memcached`, and `lighttpd` and discussed the results in detail. I also considered the required effort, both conceptual and engineering, to apply PSE and ODIT to other software development languages.

Listing 5.1: Example bug oracle

```
static bool toarith(VALUE *v) {
  switch (v->type) {
  case integer:
    return true;
  case string: {
    intmax_t value = 0;
    char *cp = v->s;
    int sign = (*cp == '-' ? -1 : 1);

    if (sign < 0)
      cp++;

    do {
      if (ISDIGIT(*cp)) {
        intmax_t new_v = 10 * value + sign * (*cp - '0');
        if (0 < sign ? (INTMAX_MAX / 10 < value || new_v < 0)
                     : (value < INTMAX_MIN / 10 || 0 < new_v))
          error(EXPR_FAILURE, 0,
                (0 < sign ? _("integer is too large: %s")
                          : _("integer is too small: %s")),
                quotearg_colon(v->s));
        value = new_v;
      } else
        return false;
    } while (*++cp);

    free(v->s);
    v->s = NULL;
    v->i = value * sign;
    v->type = integer;
    o_assert(22, sign * v->i >= 0);
    return true;
  }
  default:
    abort();
  }
}
```

Listing 5.2: Example problematic code style

```
1  struct operation {
2    ...
3  };
4
5  int main(args, input) {
6    operation *op = parse(args);
7    apply(op, input);
8  }
9
10 void apply(op, input) {
11   apply_method1(op, input);
12   apply_method1(op, input);
13   ...
14   apply_methodN(op, input);
15 }
```

Listing 5.3: Example type cast from lighttpd::b43fc00

```
1  handler_t mod_status_xxx(server *srv,
2                           connection *con,
3                           void *p_d) {
4    plugin_data *p = p_d;
5    ...
6    avg = p->abs_requests;
7    ...
8    <modified code>
9    ...
10 }
```

CHAPTER 6
FUTURE WORK

6.1 ODiT

In future, post-completion work, I would like to examine developer-centered aspects of ODiT. In the preceding section, I presumed that developers will likely recognize their own intended program behaviors. An important component of ODiT utility is the frequency with which that presumption holds. Perfect bug understanding does not exist [43]. Analogously, perfect correct-behavior understanding may not exist either. In a related question, what information about a behavioral difference does a developer need to distinguish intentional from unintentional behavior: effected program state, call trace, instruction trace, or more. I also envisage qualitative studies on the most effective approaches to depict behavioral differences to developers as well as an evaluation of developer perception of utility.

As discussed in section 5.4, the inclusion of multiple, unrelated changes to the same code can obscure behavioral differences. For example, suppose we have two unrelated updates to a program in which the first update intentionally modifies the value of a global variable, and the second update unintentionally modifies the same variable. If both updates are submitted as a single commit, then the expected behavioral difference could mask the unexpected one. Best practice for use of ODiT calls for *atomic* commits, that is, each commit should contain one update that cannot be decomposed into a set of smaller updates. Another area of future work that could mitigate the need for atomicity, *commit decomposition* reduces a combined-update commit into a sequence of loosely-coupled, coherent updates. ODiT would then run sequentially on each individual decomposed update. Commit decomposition would not only aid ODiT, but also any other differential testing technique as well.

6.2 Fixed-Point Approximation for Floating-Point Symbolic Execution

Developers use a variety of software analysis techniques to find and repair software defects. One such technique, symbolic execution (SymEx) [2, 24, 44, 45, 46] is popular due to its systematic approach to exploring execution paths and feasible program state. A tool implementing a SymEx technique collects constraints over program input variables along a program path. At each branching program statement, the SymEx tool depends upon a constraint solver to determine the feasible branches, that is branches with a satisfiable constraint. Therefore, critically, limitations in supported solver theories limit the execution space explored by the SymEx tool. Since early constraint solvers lacked a theory of floating-point arithmetic, most SymEx tools either do not support programs containing floating-point operations, or concretize symbolic floating-point values upon contact.

With the advent of floating-point enabled solvers such as Z3 [47], two independent efforts aimed to extend KLEE with floating-point support [48, 49]. The authors of both works collaborated on a case study comparing their experiences and approaches [50]. They jointly noted that both tools missed bugs or achieved low coverage due to the intractability of the generated constraints, and concluded that breakthroughs were needed in solver floating-point support before use in scalable SymEx. My approach in this preliminary work explores the improvement of SymEx coverage of floating-point code through approximation with fixed-point equivalents.

6.2.1 Approach

My technique automatically substitutes fixed-point data types and operations for native floating-points types in the module under test. By replacing the floating-point types, the executor does not require a floating-point enabled solver, nor extensions to the constraint language for type awareness. I note that fixed-point substitution is not the *breakthrough* referenced in [50]. It has many limitations as compared to native floating-point: 1) lower

precision and range, 2) may lack complete semantics for common floating-point values, such as *nan* and *inf*, 3) little/no support for floating-point exceptions. However, if the goal of the analysis is high coverage input generation, the substitution of fixed-point operations can find symbolic rational values to explore execution paths that concretization cannot.

As a reminder/primer on fixed-point, it is a number representation storing a fixed number of digits of the fractional part. Numbers are stored with two fixed bit fields, often represented in Qi.f form, with i integer bits and f fractional bits. For example, Q20.12 is a 32-bit number with 30 integer bits and 12 fractional ones. Of course, Q32.0 is just a standard uint32_t. Therefore, a number in Qi.f format with f fractional digits will be an integer multiple of 2^{-n}. For efficiently, fixed-point libraries are often hard-coded for a single Q form. Due to this representation, fixed-point addition, subtraction, and relational operators are closely related to their integer equivalents.

6.2.2 Prototype Implementation

For a preliminary assessment of this approach, I forked from the master branch of KLEE [21], adding an experimental feature, `-fixed-point`. Herein, I will refer to my prototype as KLEE-FxP. When this feature is active, the prototype transforms the module under test by mutating values of `double` type to Q32.32. A future version of KLEE-FxP will mutate `float` to Q16.16 and handle casts between the two types. Mutated values include global variables, local variables, function parameters, and function return types. Structure fields and array types are also re-written to substitute fixed-point types for floating-point. Floating-point constants are converted to their equivalent fixed-point value. Basic blocks including floating-point operations (e.g. add, sub, mul, div, etc.) and relations (are re-written to use analogous calls into the fixed-point library instead. Finally, calls to the floating-point library are redirected to an equivalent fixed-point function.

The example program in figure 6.1 declares a symbolic floating-point double (lines 6 and 7), performs some calculations (line 8), and then uses an `if` ladder that selects for

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <klee/klee.h>
4
5 int main(int argc, char *argv[]) {
6   double a;
7   klee_make_symbolic(&a, sizeof(a), "a");
8   double b = (2 * a) + 1;
9   if (b > 1.0) {
10    printf("gt\n");
11  } else if (b < 1.0) {
12    printf("lt\n");
13  } else {
14    printf("eq\n");
15  }
16  return 0;
17 }
```

Figure 6.1: Problematic SymEx Code

values greater than, less than, or equal to 1 (lines 9-15). When standard KLEE accesses the value of `a` while evaluating the expression for assignment to `b` in line 8, it concretizes its value. Consequently, the resulting calculation of `b` will also be concrete and standard KLEE will only find a single program path and generate a single test case.

In contrast to standard KLEE, when KLEE-FxP accesses `a` in line 8, it loads a standard int64 (i.e. Q32.32), and `b` is assigned the return value of `fix_add(fix_mul(2, a), 1)`. This simplification allows KLEE-FxP to find all three program paths and generate 3 test cases.

6.2.3 Modeling Floating-Point Semantics

Although arithmetically fixed-point addition, subtraction, and binary relations (e.g. less than and greater than) are equivalent to their integer analogs, some extra checks and validations are required for full floating-point semantics. For example, operands must be checked for non-number values such as `nan` and `inf` to implement the necessary behavior as per section 6 of IEEE standard 754 [51]. As an approximation of floating-point expression

evaluation, KLEE-FXP does not require a complete model of non-number semantics. Incorporating all of the non-number floating-point semantics would yield the greatest potential for coverage, but at a cost of additional constraint complexity. In short, the additional control flow to detect and handle the non-numbers results in either more generated execution states for the executor to consider or a more convoluted path condition over which the solver must reason. The former reduces the probability that all paths will complete, and the latter increases the probability of solver failure. For the preliminary experiments discussed in the next section, I reserved a single `NaN` and a single ∞ value instead of IEEE-754's `sNaN`, `qNan`, $-\infty$ and $+\infty$. The model does not support any type of floating-point exceptions. In the KLEE-FXP model:

- `isnan(x)` and `isinf(x)` return true if and only if x is the designated value
- binary operations with a `NaN` operand return `NaN`
- unary operations with a `NaN` operand return `NaN`
- for $x < 0$, $sqrt(x)$ returns `NaN`

The best model of non-number semantics to achieve maximum coverage is an open topic and the subject of future work. Perhaps a static analysis of the subject program could guide selection from a set of available models. For example, the subject in figure 6.1 requires no model of non-numbers for complete coverage.

Another aspect of floating-point semantics that must be modeled is wrapping overflow. The sum of two floating-point numbers cannot be a negative floating-point number. Therefore, integer overflow, in either direction, must be detected and prevented. The final element of modeled floating-point semantic expectation is atomicity, that is there is a single path through a floating-point operator. Therefore, the operator's fixed-point analog merges all states forked during the execution of the operator. A single state enters the operator, and after merging, a single state leaves.

```
1  fix_t fix_div(fix_t a, fix_t b) {
2
3      fix_t result = klee_fix_t();
4      klee_assume(a = fix_mul(b, result));
5      return result;
6  }
7
8  fix_t fix_sqrt(fix_t a) {
9
10     fix_t result = klee_fix_t();
11     klee_assume(a = fix_mul(result, result));
12     return result;
13 }
```

Figure 6.2: Problematic SymEx Code

This approach yields satisfactory results for addition, subtraction, and multiplication. However, the fixed point algorithms for division and square root involve multiple loops with embedded conditions. Before merging, simple division generated 126 states; an unwieldy number to merge. After 60 seconds, the square root algorithm had not terminated and had generated over 1,200 incomplete states. Fortunately, the inverse operation for both of these, multiplication, is much simpler. In both cases, the result can be expressed by returning a new symbolic variable constrained by the operation's inputs. Figure 6.2 contains pseudocode for both operations. Note that this technique essentially outsources the accumulation of the path condition to the solver, but results in a simpler, but equivalent path condition.

6.2.4 Preliminary Experimental Results

I implemented the floating point transformation as an LLVM [35] re-writing pass during module preparation in KLEE [21]. The fixed-point library with the floating-point semantic model is written in C and compiled to a bitcode library linked with the target program after transformation. To evaluate the coverage achieved by KLEE-FXP, I selected a subset of the benchmarks used to evaluate both versions of KLEE-float [50]; specifically, the synthetic

Table 6.1: KLEE-FXP prelinminary experimental results.

benchmark	ICov	BCov	ICov	BCov
count_klee	33.87	22.22	91.85	85.00
interval_klee_no_bug	94.10	56.25	93.85	58.33
matrix_inverse_klee_double_4	89.20	75.00	97.06	89.47
memcpy_and_use_as_bitvector_klee	93.51	70.83	92.12	78.57
non_terminating_klee_no_bug	83.72	50.00	90.83	60.00
prefix_sum_klee_bug_double	95.65	75.00	94.81	85.00
rounding_sqrt_klee	21.90	12.50	72.73	65.00
sorted_search_klee_bug_float	81.11	54.55	95.29	91.67
sqrt_klee	74.44	40.91	93.97	67.65
sum_is_commutative_klee_double	97.96	75.00	94.19	75.00
sum_is_not_associative_klee_bug	97.96	75.00	94.19	75.00
vanishing_klee_bug	72.60	40.00	91.83	71.43
Total (12)	78.00	53.94	91.86	75.18

benchmarks contributed by the Imperial team. Of these 14 benchmarks, I eliminated one that depended upon the representation format of IEEE-754 and one for which I found no symbolic input; leaving 12 benchmarks for a preliminary experiment. Table 6.1 lists the benchmarks in the left-most column. The middle two columns contain the instruction and branch coverage resulting from running standard KLEE with a timeout of 60 minutes per benchmark. The final two columns contain the instruction and branch coverage of KLEE-FXP with the same parameters. These results must be considered preliminary as they both reflect coverage as reported by KLEE. The coverage statistics are generally indicative of performance, but some variation is expected due to module transformation injecting both additional instructions and branches. These results are very encouraging, as KLEE-FXP achieved higher branch coverage than traditional KLEE in 10 of 12 benchmarks.

6.3 Summary

As I have emphasized in this chapter, these results are preliminary. Additional work remains to complete the evaluation of KLEE-FXP's approach. The evaluation should include:

- All of the evaluation benchmarks used in [50].

- Either or both implementations of KLEE-float.
- Normalized coverage statistics to only include program instructions and branches.

Future work in this area includes an evaluation of fixed-point approximation to detect potential floating-point errors. Due to the limited range, the approach may have difficulty detecting potential overflow or underflow. I would also like to explore applications of the technique to program analysis and differential testing of scientific computing software.

CHAPTER 7

RELATED WORK

7.1 Program Tracing

Prior approaches to program tracing required instrumentation of the monitored system or runtime support. Software instrumentation is unfortunately expensive (e..g, they can incur 31% overhead for (acyclic) path profiling alone [16]). Recent processor designs, such as Intel Processor Trace, can reduce this overhead to as little as 5% [52], but they require a sophisticated processor, still entail non-zero overhead, and consume considerable storage capacity and throughput.

7.2 Symbolic Execution

The key concepts of classical symbolic execution, as provided in [2], have been implemented for input generation in many prior tools [44, 54, 24, 6, 45, 55, 46, 56, 57, 53]. My input generation work builds on this rich body of research on symbolic execution and automated test generation.

My PSE technique is closely related to UC-KLEE [32], which performs under-constrained symbolic execution from an arbitrary function call. This degree of under-constraining has been shown to be effective for patch validation and defect detection [9]. However, unlike PSE, UC-KLEE misses some path segments needed for complete ZOPI training.

Chopped symbolic execution [58] shares with my work the goal of reaching code buried deep in the call graph, but takes an orthogonal approach; it employs program slicing to exclude uninteresting portions of the code from symbolic execution, while maintaining soundness. For ZOPI training, the only uninteresting code is dead code, so this approach would not be applicable to this context.

Since symbolic execution is susceptible to path explosion [3], numerous approaches have been defined to address this issue. Guided path search heuristics select paths for exploration either randomly [59, 24] or based on the predicted likelihood of reaching a given coverage target [59, 24, 44]. Other approaches reduce the number of paths to search by removing equivalent paths [60], removing paths that cannot reach new code [61], or merging state on selected paths [62]. PSE could benefit from (suitably adapted versions of) these techniques.

Lazy initialization is an important feature of generalized symbolic execution [6], as it allows the handling of unconstrained pointers or references. Various tools have implemented lazy initialization for symbolic execution in Java (e.g., [6, 7, 8]), C/C++ (via LLVM IR [9]), and object code [10]. Because this technique can inflict a significant performance penalty, researchers have proposed optimizations for Java-based symbolic execution [11, 12, 13]. However, these optimizations would be difficult to implement without Java's memory manager and strict type safety. UC-KLEE [32] uses a version of lazy initialization that models unconstrained pointers as either NULL or pointing to a new memory object. I augmented this model with existing memory locations that were either allocated with or typecast to the type of entity being lazily initialized.

7.3 Fuzzing

Fuzzing in another approach to explore program behavior through program execution with inputs randomly sampled from the program's input space. In principle, symbolic execution are starkly different solutions to the same problem. Symbolic Execution is a static analysis, white-box technique to systematically consider the entire program input space. Fuzzing is a dynamic analysis, black-box technique to randomly consider valid program inputs. However, the intractability of thorough program behavior exploration has lead to a degree of convergence in state-of-the art techniques from both approaches. For example, symbolic execution frequently generates more symbolic states that can be explored in a fixed

period of time. In practice, executors randomly select a next state for execution (though the selection may be biased to satisfy a coverage goal). Similarly, state-of-the-art fuzzing techniques have leveraged some view into program internals (gray-box) to improve their resulting coverage.

The first recognized fuzzing utility, from Miller *et al.* [63], was a pure black-box random string generator. The inputs generated were used for crash and non-termination detection on a collection of standard Unix utilities. The tool only generated ASCII strings used as program inputs, whereas PSE can generate complex structured inputs to functions internal to the program. Fuzzers such as Randoop [64] and AFL [65] instrument the program under test to capture line and branch coverage metrics as a feedback mechanism to bias input generation toward new code. As a white-box technique, PSE does not need to instrument the program, as it is never directly executed. One challenge both symbolic execution and gray-box fuzzing have in common is scheduling. An executor must select the next state to explore. How the next state is selected (or scheduled) significantly influences the coverage of the resulting analysis. Similarly, fuzzers select and mutate prior inputs. AFLfast [66] significantly improved input generation coverage over AFL [65] through an innovative scheduling algorithm. Unlike PSE which uses the program itself to construct structured input, random input fuzzers may have difficulty generating inputs that pass input validation. For example, purely random inputs are unlikely to reach past a GIF image parser into later image manipulation code. To mitigate this limitation, T-Fuzz [67] transforms the subject program to remove the bottleneck code and libfuzzer [68] allows manual input format specification to guide input generation. Several techniques combine heavier weight analyses with lighter weight ones to improve coverage. For example, Angora [69] uses a more costly taint-analysis step to inform a subsequent low-instrumentation generation step. Another successful approach is to combine symbolic execution and fuzzing into a hybrid by alternating between them, such as Sage [55], Driller [70], and QSYM [71]. Currently, ODIT only uses PSE for input generation and could benefit from the incorporation of a

similar hybrid technique.

7.4 Regression and Differential Testing

The efficacy of regression testing is often constrained by two factors: the (1) difficulty of providing accurate oracles to expose regressions [72, 29], and (2) limitations of existing test suites to reveal unintended behavior changes [73]. Prior work has proposed many techniques addressing these difficulties.

PREAMBL applies automatically generated invariants to regression testing [74], using predicates as oracles. Pastore *et al.* applies model checking to regression testing [75]. Both techniques generate oracles on the fly and can detect regression faults that do not surface to the program's external behaviors. Felsing *et al.* present automatic regression verification for programs employing complex arithmetic on integer variables [76].

Another limitation of regression testing comes from the expense of running and maintaining test suites. Researchers have proposed numerous automated methods for selection, prioritization, minimization, and augmentation of regression tests [77, 78, 79, 80, 81, 64, 82, 83, 84]. Flaky tests also contribute to the cost of regression testing, with a growing body of work on mitigation [85, 86, 87].

Regression test suites may insufficiently test the changed code [88, 84]. Researchers have therefore proposed techniques for generating tests targeting the modified code. DiffGen focuses on automated regression unit-test generation based on source code semantic differencing [89]. eXpress selectively generates tests that are more likely to detect behavioral differences across code versions [90]. EvoSuiteR uses a search-based approach to reach and propagate changes between versions [91]. Similarly to ODIT, BERT uses java input generation and program behavior comparison [92].

Much like regression testing, symbolic execution has received considerable attention since its introduction [2] and has flourished in areas such as program verification and automated input generation [45, 55, 46, 56, 57, 53, 93]. My work builds upon all of these prior

works.

Several recent works are of particular relevance. Engler and Dunbar [5] introduced the concept of under-constrained symbolic execution (SE). Ramos and Engler [32] introduced and evaluated a general purpose, scalable implementation of under-constrained SE, UC-KLEE. Further work [9] applied UC-KLEE to verify that a software patch did not introduce crashes. UC-KLEE and PSE share a common genealogy, as offsprings of KLEE [24, 21]. Ye, Zhao, and Sarkar [94] took an approach, partial symbolic execution, similar to PSE to detect MPI usage anomalies. PSE, UC-KLEE, and partial symbolic execution can all start at any function. However, PSE goes further than either of these other tools by additionally unconstraining local variable state and substituting unconstraining stubs. SPD [95] also considered over-approximate symbolic execution by abstracting away path condition clauses. In a related approach, Godefroid [10] developed MicroExecution to execute native assembly language instructions. DiSE [96] steers dynamic symbolic execution toward program differences identified through static analysis. Related to my behavior difference clustering, the advent of field failure data collection and automated test generation techniques has produced a wealth of crash and test failure reports requiring automated approaches to triage related faults. ReBucker [97] and RETracer [98] cluster crash reports based upon stack trace information. Pham *at al.* cluster failing tests based upon a symbolic analysis of the failing test's accumulated path condition [99].

CHAPTER 8
CONCLUSION

The objective of my work described in this thesis is to improve the results obtained from dynamic analysis by increasing code coverage through over-approximated input generation techniques. In pursuit of this goal, I have developed Progressive Symbolic Execution (PSE) and Over-approximate Differential Testing (ODIT), implemented tools to realize these techniques, and applied these tools in evaluation of these techniques. All tools and datasets utilized during my empirical experiments are publicly available for reuse and evaluation.

PSE is an input generation technique that produces a set of replay cases that cover a superset of a program's feasible path segments, where a path segment is a sub-graph of a program function's control flow graph (CFG). A segment is feasible if and only if a program input exists that reaches and executes the segment. A replay case is an ordered pair of program fragment and inputs that, when applied to the fragment, executes (or replays) the embedded path segments. I implemented a scaffolding generator that realized the replay cases as a set of compilable program sources. The resulting programs were then executed on the target device for recording and incorporation into a model used by Zero-Overhead Path Inference (ZOPI). The ZOPI evaluation both directly and indirectly support the effectiveness of the technique.

ODIT is a differential testing technique that identifies local behavioral differences between two versions of a program. It uses under-constrained symbolic execution to generate function inputs in a local neighborhood (by threshold distance in program's call graph), of changed code. The technique generates function inputs constructed to be executable on both versions. It then concretely executes each input on the programs, while accumulating a sequence of program-state snapshot pairs (one from each version). Each state pair is then

compared to identify potential behavioral differences. Due to the under-constrained input generation, some of these collected differences may be infeasible; thus, they are an over-approximation of actual program behavioral differences. Also, the feasible differences will contain both intentional and unintentional changes. ODIT summarizes, orders, and ranks these differences for display to the developer to emphasize the feasible, unintentional differences.

To evaluate ODIT, I constructed prototype tools that are publicly available for download and re-use. For example, the ODIT tool is fully automated. It calculates program differences at the LLVM bitcode level, selects functions in a local neighborhood of these changes, generates and replays inputs, and reports program differences without developer input or program instrumentation. The results of the experiments presented herein show the effectiveness of my techniques and reinforce optimism for the results of my remaining work.

In future work, I would like to examine some human-factors related aspects of ODIT. In the published work, I presumed that developers will likely recognize their own intended program behaviors. An important component of ODIT utility is the frequency with which that presumption holds. Perfect bug understanding does not exist [43]. Analogously, perfect correct-behavior understanding may not exist either. In a related question, what information about a behavioral difference does a developer need to distinguish intentional from unintentional behavior: effected program state, call trace, instruction trace, or more. I also envisage qualitative studies on the most effective approaches to depict behavioral differences to developers as well as an evaluation of developer perception of utility.

In addition to the human-centric studies, I would also like to establish a proxy metric for ground-truth, unintentional behavior beyond the bug-inducing behavior changes. Though a bug can reasonably be inferred to be unintended, the contrapositive is unlikely to be true. There will likely be much unintended software change side effects inducing benign behavior. Perhaps developer intentionality could be inferred from commit associated test

suite modification. Additionally, a proxy for ground-truth, erroneous behavior is required to conduct extensive ODITstudies on real-world software beyond curated benchmarks. After all, inspection of an interval of commits is insufficient to establish whether a code update introduced an error. The error could have been detected long after the interval end, or the error may be latent and yet to be detected. Unfortunately, the reliance on manual inspection limits available analyses of ODITempirical results.

Finally, KLEE-FXP has potential application in both traditional symbolic execution and floating-point intensive areas such as scientific computing. I expect to use the prototype tool in the analysis of `kokkos` [100] software to detect such issues as invalid api usage, faulty host/device memory management, and potential data-race conditions in software written for high-performance computing clusters.

REFERENCES

[1] H. Krasner, "The Cost of Poor Software Quality in the US: A 2020 Report," p. 46, 2020.

[2] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[3] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[4] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1128–1138, Dec. 1986.

[5] D. Engler and D. Dunbar, "Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07, New York, NY, USA: ACM, 2007, pp. 1–4, ISBN: 978-1-59593-734-6.

[6] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, Apr. 2003, pp. 553–568.

[7] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF–SE: A Symbolic Execution Extension to Java PathFinder," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Mar. 2007, pp. 134–138.

[8] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04, New York, NY, USA: ACM, 2004, pp. 97–107, ISBN: 978-1-58113-820-7.

[9] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64, ISBN: 978-1-931971-23-2.

[10] P. Godefroid, "Micro Execution," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: ACM, 2014, pp. 539–549, ISBN: 978-1-4503-2756-5.

[11] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias, "BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 639–660, Jul. 2015.

[12] P. Braione, G. Denaro, and M. Pezzè, "Enhancing Symbolic Execution with Built-in Term Rewriting and Constrained Lazy Initialization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, New York, NY, USA: ACM, 2013, pp. 411–421, ISBN: 978-1-4503-2237-9.

[13] X. Deng, J. Lee, and Robby, "Efficient and formal generalized symbolic execution," *Automated Software Engineering*, vol. 19, no. 3, pp. 233–301, Sep. 2012.

[14] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[15] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead Profiling via EM Emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 401–412, ISBN: 978-1-4503-4390-9.

[16] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996. MICRO-29*, Dec. 1996, pp. 46–57.

[17] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic, *Artifact: Zero-Overhead Path Prediction with Progressive Symbolic Execution*, Feb. 2019.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Apr. 2002, pp. 213–228.

[19] *Clang C Language Family Frontend for LLVM*, https://clang.llvm.org/.

[20] *NetworkX — NetworkX documentation*, https://networkx.org/.

[21] *KLEE Symbolic Virtual Machine*, KLEE, Sep. 2022.

[22] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–, ISBN: 978-0-7695-2102-2.

[23] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 519–531, ISBN: 978-3-540-73368-3.

[24] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[25] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[26] *Avahi - mDNS/DNS-SD*, https://avahi.org/.

[27] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, Washington, DC, USA: IEEE Computer Society, 2014, pp. 242–254, ISBN: 978-1-4799-6998-2.

[28] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 93–104, ISBN: 978-1-4503-2645-2.

[29] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, Nov. 1982.

[30] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 85–99, ISBN: 978-1-4503-4261-2.

[31] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a Doubt: Testing for Divergences between Software Versions," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1181–1192.

[32] D. A. Ramos and D. R. Engler, "Practical, Low-Effort Equivalence Verification of Real Code," in *Computer Aided Verification*, Springer Berlin Heidelberg, Jul. 2011, pp. 669–685.

[33] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[34] M. Böhme and A. Roychoudhury, "CoREBench: Studying Complexity of Regression Errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: ACM, 2014, pp. 105–115, ISBN: 978-1-4503-2645-2.

[35] *The LLVM Compiler Infrastructure Project*, http://llvm.org/.

[36] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys*, vol. 51, no. 1, 17:1–17:24, Jan. 2018.

[37] *Redis Homepage*, https://redis.io/.

[38] *Memcached - a distributed memory object caching system*, https://www.memcached.org/.

[39] *Lighttpd - fly light*, https://www.lighttpd.net/.

[40] R. Rutledge and A. Orso, "Automating differential testing with overapproximate symbolic execution," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2022, pp. 256–266.

[41] *What are Git version control best practices?* https://about.gitlab.com/topics/version-control/version-control-best-practices/.

[42] *Introducing GitFlow*, https://datasift.github.io/gitflow/IntroducingGitFlow.html.

[43] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 199–209, ISBN: 978-1-4503-0562-4.

[44] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 10:1–10:38, Dec. 2008.

[45] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, New York, NY, USA: ACM, 2005, pp. 213–223, ISBN: 1-59593-056-6.

[46] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly*

with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. ESEC/FSE-13, New York, NY, USA: ACM, 2005, pp. 263–272, ISBN: 1-59593-014-0.

[47] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[48] *KLEE-float*, https://github.com/COMSYS/klee-float, May 2021.

[49] *KLEE-float*, https://github.com/srg-imperial/klee-float, Apr. 2022.

[50] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zahl, and K. Wehrle, "Floating-point symbolic execution: A case study in N-version programming," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 601–612.

[51] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019.

[52] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015.

[53] G. Li, I. Ghosh, and S. P. Rajan, "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Jul. 2011, pp. 609–615.

[54] C. Cadar and D. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," in *Model Checking Software*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Aug. 2005, pp. 2–23.

[55] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, 20:20–20:27, Jan. 2012.

[56] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Apr. 2008, pp. 134–153.

[57] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara, "FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 106–117, ISBN: 978-1-4503-4390-9.

[58] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped Symbolic Execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: ACM, 2018, pp. 350–360, ISBN: 978-1-4503-5638-1.

[59] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446, ISBN: 978-1-4244-2187-9.

[60] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Mar. 2008, pp. 351–366.

[61] S. Bugrara and D. Engler, "Redundant State Detection for Dynamic Symbolic Execution," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, Berkeley, CA, USA: USENIX Association, 2013, pp. 199–212.

[62] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, New York, NY, USA: ACM, 2012, pp. 193–204, ISBN: 978-1-4503-1205-9.

[63] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[64] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, Montreal, Quebec, Canada: Association for Computing Machinery, Oct. 2007, pp. 815–816, ISBN: 978-1-59593-865-7.

[65] *American fuzzy lop*, https://lcamtuf.coredump.cx/afl/.

[66] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1032–1043, ISBN: 978-1-4503-4139-4.

[67] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 697–710.

[68] *libFuzzer – a library for coverage-guided fuzz testing. — LLVM 16.0.0git documentation*, https://www.llvm.org/docs/LibFuzzer.html.

[69] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 711–725.

[70] N. Stephens *et al.*, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2016, ISBN: 978-1-891562-41-9.

[71] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761, ISBN: 978-1-939133-04-5.

[72] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," *University of Sheffield, Tech. Rep. CS-13-01*, 2013.

[73] J. Voas, "PIE: A dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[74] P. Sagdeo, N. Ewalt, D. Pal, and S. Vasudevan, "Using automatically generated invariants for regression testing and bug localization," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13, Silicon Valley, CA, USA: IEEE Press, Nov. 2013, pp. 634–639, ISBN: 978-1-4799-0215-6.

[75] F. Pastore *et al.*, "Verification-aided regression testing," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 37–48, ISBN: 978-1-4503-2645-2.

[76] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, New York, NY, USA: Association for Computing Machinery, Sep. 2014, pp. 349–360, ISBN: 978-1-4503-3013-8.

[77] L. Zhang, "Hybrid Regression Test Selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 199–209.

[78] M. J. Harrold *et al.*, "Regression test selection for Java software," *ACM SIGPLAN Notices*, vol. 36, no. 11, pp. 312–326, Oct. 2001.

[79] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[80] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "TimeAware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06, New York, NY, USA: Association for Computing Machinery, Jul. 2006, pp. 1–12, ISBN: 978-1-59593-263-1.

[81] H.-Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 419–429.

[82] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A Hybrid Directed Test Suite Augmentation Technique," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Nov. 2011, pp. 150–159.

[83] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel, "Continuous test suite augmentation in software product lines," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC '13, New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 52–61, ISBN: 978-1-4503-1968-3.

[84] Z. Xu, Y. Kim, M. Kim, M. B. Cohen, and G. Rothermel, "Directed test suite augmentation: An empirical investigation," *Software Testing, Verification and Reliability*, vol. 25, no. 2, pp. 77–114, 2015.

[85] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: Association for Computing Machinery, May 2018, pp. 433–444, ISBN: 978-1-4503-5638-1.

[86] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Apr. 2019, pp. 312–322.

[87] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 545–555, ISBN: 978-1-4503-5572-8.

[88] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-Suite Augmentation for Evolving Software," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 218–227.

[89] K. Taneja and T. Xie, "DiffGen: Automated Regression Unit-Test Generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 407–410.

[90] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided path exploration for efficient regression test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 1–11, ISBN: 978-1-4503-0562-4.

[91] S. Shamshiri, "Automated unit test generation for evolving software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 1038–1041, ISBN: 978-1-4503-3675-8.

[92] W. Jin, A. Orso, and T. Xie, "Automated Behavioral Regression Testing," in *2010 Third International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2010, pp. 137–146.

[93] F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, Sep. 2017, pp. 8–9.

[94] F. Ye, J. Zhao, and V. Sarkar, "Detecting MPI Usage Anomalies via Partial Program Symbolic Execution," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 794–806.

[95] R. Santelices and M. J. Harrold, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10, New York, NY, USA: Association for Computing Machinery, Jul. 2010, pp. 195–206, ISBN: 978-1-60558-823-0.

[96] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 1, 3:1–3:42, Oct. 2014.

[97] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, Jun. 2012, pp. 1084–1093, ISBN: 978-1-4673-1067-3.

[98] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "RETracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, New York, NY, USA: Association for Computing Machinery, May 2016, pp. 820–831, ISBN: 978-1-4503-3900-1.

[99] V.-T. Pham, S. Khurana, S. Roy, and A. Roychoudhury, "Bucketing Failing Tests via Symbolic Analysis," in *Fundamental Approaches to Software Engineering*, M. Huisman and J. Rubin, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2017, pp. 43–59, ISBN: 978-3-662-54494-5.

[100] *Kokkos*, https://github.com/kokkos.