Finding Code That Explodes Under Symbolic Evaluation

JAMES BORNHOLT, University of Washington, USA
EMINA TORLAK, University of Washington, USA

Solver-aided tools rely on *symbolic evaluation* to reduce programming tasks, such as verification and synthesis, to satisfiability queries. Many reusable symbolic evaluation engines are now available as part of solver-aided languages and frameworks, which have made it possible for a broad population of programmers to create and apply solver-aided tools to new domains. But to achieve results for real-world problems, programmers still need to write code that makes effective use of the underlying engine, and understand where their code needs careful design to elicit the best performance. This task is made difficult by the all-paths execution model of symbolic evaluators, which defies both human intuition and standard profiling techniques.

This paper presents *symbolic profiling*, a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation. To help with diagnosis, we develop a catalog of common performance anti-patterns in solver-aided code. To locate these bottlenecks, we develop SymPro, a new profiling technique for symbolic evaluation. SymPro identifies bottlenecks by analyzing two implicit resources at the core of every symbolic evaluation engine: the *symbolic heap* and *symbolic evaluation graph*. These resources form a novel performance model of symbolic evaluation that is general (encompassing all forms of symbolic evaluation), explainable (providing programmers with a conceptual framework for understanding symbolic evaluation), and actionable (enabling precise localization of bottlenecks). Performant solver-aided code carefully manages the shape of these implicit structures; SymPro makes their evolution explicit to the programmer.

To evaluate SymPro, we implement profilers for the Rosette solver-aided language and the Jalangi program analysis framework. Applying SymPro to 15 published solver-aided tools, we discover 8 previously undiagnosed performance issues. Repairing these issues improves performance by orders of magnitude, and our patches were accepted by the tools' developers. We also conduct a small user study with Rosette programmers, finding that SymPro helps them both understand what the symbolic evaluator is doing and identify performance issues they could not otherwise locate.

CCS Concepts: • **Software and its engineering → Software performance**; **Automatic programming**;

Additional Key Words and Phrases: symbolic execution, solver-aided programming, profiling

ACM Reference Format:
James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 149 (November 2018), 26 pages. https://doi.org/10.1145/3276519

1 INTRODUCTION

Solver-aided tools have automated a wide range of programming tasks, from test generation to program verification and synthesis. Such tools work by reducing programming problems to satisfiability queries that are amenable to effective SAT or SMT solving. This reduction is performed by the tool's *symbolic evaluator*, which encodes program semantics as logical constraints. Effective symbolic evaluation is thus key to effective solver-aided automation.

Authors' addresses: James Bornholt, University of Washington, USA, bornholt@cs.washington.edu; Emina Torlak, University of Washington, USA, emina@cs.washington.edu.

2475-1421/2018/11-ART149
https://doi.org/10.1145/3276519

Building and applying solver-aided automation used to be the province of experts, who would invest years of work to obtain an efficient symbolic evaluator for a new application domain. This barrier to entry is now greatly reduced by the availability of solver-aided languages (e.g., [Torlak and Bodik 2014; Uhler and Dave 2014]) and frameworks (e.g., [Bucur et al. 2014; Sen et al. 2013]), which provide reusable symbolic evaluation engines for programmers to target. These platforms have made it possible for a broader population of programmers, from high-school students to professional developers, to rapidly create solver-aided tools for many new domains (e.g., Table 1).

But scaling solver-aided programs to real problems, either as a developer or a user, remains challenging. As with classic programming, writing code that performs well (under symbolic evaluation) requires the programmer to be able to *identify* and *diagnose* performance bottlenecks—which parts of the program are costly to evaluate (symbolically) and why. For example, if a program synthesis tool is timing out on a given task, the tool's user needs to know whether the bottleneck is in the problem specification or the solution sketch [Solar-Lezama et al. 2006]. Similarly, if neither the specification nor the sketch is the bottleneck, then the tool's developer needs to know where and how to improve the interpreter that specifies the semantics of the tool's input language. Yet unlike classic runtimes, which employ an execution model that is familiar to programmers and amenable to time- and memory-based profiling, symbolic evaluators employ an unfamiliar execution model (i.e., evaluating *all* paths through a program) that defies standard profilers. As a result, programmers currently rely on hard-won intuition and ad-hoc experimentation to diagnose and optimize solver-aided code.

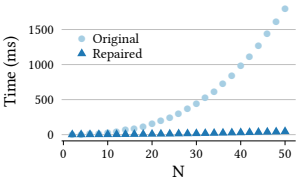
This paper presents *symbolic profiling*, a systematic new approach to identifying and diagnosing code that performs poorly under symbolic evaluation. Our contribution is three-fold. First, we develop SymPro, a new (and only) profiling technique that can identify root causes of performance bottlenecks in solver-aided code. Here, we use the term 'solver-aided code' to generically refer to any program (in any programming language) that is being evaluated symbolically to produce logical constraints. Second, to help programmers diagnose these bottlenecks, we develop a catalog of the most common programming anti-patterns for symbolic evaluation. Third, we conduct an extensive empirical evaluation of symbolic profiling, showing it to be an effective tool for finding and fixing performance problems in real applications.

Symbolic Profiling. What characterizes the behavior of programs under symbolic evaluation? The fundamental challenge for symbolic profiling is to answer this question with a performance model of symbolic evaluation that is *general*, *explainable*, and *actionable*. A general model applies to all solver-aided platforms and must therefore encompass all forms of symbolic evaluation, from symbolic execution [Clarke 1976; King 1976] to bounded model checking [Biere et al. 1999]. An explainable model provides a conceptual framework for programmers to understand what a symbolic evaluator is doing, without having to understand the details of its implementation. Finally, an actionable model enables profiling tools to precisely identify root causes of performance bottlenecks in symbolic evaluation. Developing a model of symbolic evaluation that satisfies all three of these goals is the core technical contribution of this paper.

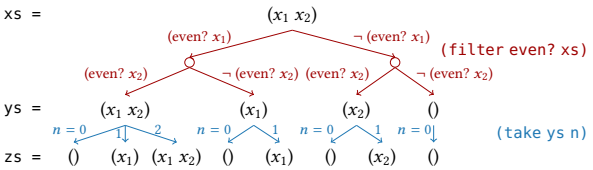
To illustrate the symbolic profiling challenge, consider applying a standard time-based profiler to the toy program in Figure 1a. The program checks that the sum of any $n \leq N$ even integers is also even. Under symbolic evaluation for $N = 20$, the `take` call (line 5) takes an order of magnitude more time than `filter` (line 4), so a time-based profiler identifies `take` as the location to optimize. The source of the problem, however, is the call to `filter`, which generates $O(2^N)$ paths when applied to a symbolic list of length N (Figure 1c). The `take` procedure, in contrast, generates $O(N)$ paths. A time-based profiler incorrectly blames `take` because it is evaluated 2^N times, once for each path generated by `filter`. The correct fix is to avoid calling `filter` (Figure 1d). This repair location is missed not only by time-based profiling but also by models that rely on common concepts from

```
1 (define (sum-of-even-integers-is-even N)
2   (define-symbolic* xs integer? [N]) ; xs = list of N symbolic integers
3   (define-symbolic* n integer?)      ; n = single symbolic integer
4   (define ys (filter even? xs))      ; ys = even integers from xs
5   (define zs (take ys n))            ; zs = first n elements of ys
6   (assert (even? (apply + zs))))     ; Check that the sum of zs is even
```

(a) A program that checks that the sum of any $n \leq N$ even integers is even.



(b) The original program (a) performs poorly as N grows.



(c) The original program (a) creates $O(N2^N)$ paths (here, $N = 2$).

```
1 (define (sum-of-even-integers-is-even N)
2   (define-symbolic* xs integer? [N])
3   (define-symbolic* n integer?)
4   (define zs (take xs n))
5   (when (andmap even? zs)
6     (assert (even? (apply + zs)))))
```

(d) Repairing (a) to obtain asymptotically better performance.

Fig. 1. A toy solver-aided program that performs poorly under symbolic evaluation.

the symbolic evaluation literature, such as path condition size or feasibility. For instance, a simple model that counts the total number of paths generated by a call will also blame `take`. It is, of course, possible to design more sophisticated time- and path-based models that can handle our toy example, and we examine such designs in Section 4.1, but they fall short on real code.

Symbolic profiling employs a new performance model of symbolic evaluation that is based on the following key insight: effective symbolic evaluation involves maximizing (opportunities for) concrete evaluation while minimizing path explosion. Classic concrete execution is thus a special, ideal case of symbolic evaluation—all operations are evaluated on concrete values along a single path of execution. Since this ideal cannot be achieved in the presence of symbolic values, symbolic evaluators choose which goal to prioritize at a given point by basing their evaluation strategy on either symbolic execution (SE) or bounded model checking (BMC). As illustrated in Figure 4, SE maximizes concrete evaluation but suffers from path explosion, while BMC avoids path explosion but affords few opportunities for concrete evaluation. Performant solver-aided code elicits a practical balance between SE- and BMC-style evaluation in the underlying engine. The challenge for a symbolic profiler is therefore to help programmers identify the parts of their code that deviate most from concrete evaluation by generating excessive symbolic state or paths.

SymPro. We address this challenge with SymPro, a new profiling technique that tracks two abstract resources, the *symbolic heap* and the *symbolic evaluation graph*, which form our performance model. The symbolic heap consists of all symbolic values (constants, terms, etc.) created by the program, while the symbolic evaluation graph reflects the engine's evaluation strategy (which paths were explored individually, which were merged, etc.). In concrete execution, the symbolic heap is empty, and the evaluation graph consists of a single path. In symbolic evaluation, these resources evolve depending on the evaluation strategy. For example, the evaluation graph is a tree for SE engines, a DAG for BMC engines, and a mix of sub-trees and sub-DAGs for hybrid engines. The symbolic heap and graph are implicit in every forward symbolic evaluation engine, making our model general. They also capture the full spectrum of symbolic evaluation behaviors in an implementation-independent way, making our model explainable and actionable. SymPro tracks the evolution of the symbolic heap and graph, identifying where new symbolic values are created, which values are frequently accessed, which values are eventually used in queries sent to a satisfiability solver, and how evaluation paths are merged at control-flow joins. It ranks procedure

calls by these metrics to present the most expensive calls to the programmer. Given our motivating example from Figure 1a, SymPro correctly identifies the call to `filter` as the bottleneck.

Anti-Patterns. To help the programmer diagnose the identified bottlenecks, we present a catalog of the most common performance anti-patterns in solver-aided code. These include *algorithmic*, *representational*, and *concreteness* problems. For example, the program in Figure 1a suffers from *irregular representation*. It constructs a symbolic representation of $n \leq N$ even integers that describes $O(N2^N)$ concrete lists. The repaired program in Figure 1d, in contrast, constructs a symbolic representation of $n \leq N$ integers that describes $O(N)$ concrete lists; this representation is then combined with a precondition (that all of its elements are even) before checking the desired property. We present a canonical example of each kind of anti-pattern, along with a repair the programmer could make.

Evaluation. We have implemented SymPro for the Rosette solver-aided language [Torlak and Bodik 2013, 2014], which extends Racket [Racket 2017] with support for verification and synthesis. Our implementation is open-source and integrated into Rosette [Torlak 2018]. To evaluate the effectiveness of our profiler, we performed a literature survey of recent programming languages research, gathering 15 tools built using Rosette. Applying SymPro to these tools, we found 8 previously unknown bottlenecks. Repairing these bottlenecks improved the tools' performance by orders of magnitude (up to 290×), and several developers accepted our patches.

To demonstrate that SymPro profiles are *actionable*, we present detailed case studies on three of these Rosette-based tools, describing how a programmer can use SymPro to iteratively improve the performance of such a tool using language constructs afforded by Rosette and algorithmic changes guided by profile data. To show that SymPro is *explainable*, we conduct a small user study with Rosette programmers, showing that SymPro helps them identify performance bottlenecks in Rosette programs more efficiently than with standard (time-based) profiling tools. Finally, to show that symbolic profiling is *general*, we build a prototype symbolic profiler for Jalangi [Sen et al. 2013], a JavaScript program analysis framework with a symbolic execution pass [Sen et al. 2015], and show that it finds bottlenecks in JavaScript programs that a time profiler misses.

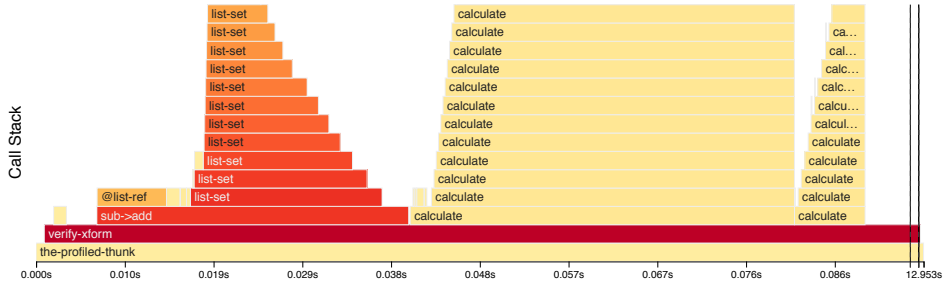
In summary, this paper makes the following contributions:

- *Symbolic profiling.* A new technique, SymPro, for identifying performance bottlenecks in solver-aided code. SymPro is based on a new performance model of symbolic evaluation that tracks the evolution of the symbolic heap and the symbolic evaluation graph.
- *Symbolic evaluation anti-patterns.* A catalog of common performance anti-patterns in solver-aided code, to help with the diagnosis and repair of the identified bottlenecks.
- *Profiler implementations.* To demonstrate generality, we have built both a full-featured implementation of SymPro for Rosette [Torlak and Bodik 2013, 2014], integrated into the latest Rosette release, and a proof-of-concept implementation for Jalangi [Sen et al. 2013].
- *Empirical evaluation.* An extensive empirical evaluation of SymPro's effectiveness on real-world benchmarks, including three detailed case studies showing symbolic profiles are actionable, and a small user study with Rosette programmers showing they are explainable.

The rest of this paper is organized as follows. Section 2 introduces symbolic profiling by working through a small example. Section 3 provides background on symbolic evaluation and catalogs common performance anti-patterns in solver-aided code. Section 4 presents symbolic profiling and our implementations. Section 5 describes three case studies showing SymPro profiles are actionable. Section 6 performs a detailed evaluation of SymPro's explainability, generality, and performance. Section 7 describes related work, and Section 8 concludes.

```
1  (define-values (Add Sub Sqr Nop) ; Calculator opcodes.
2    (values (bv 0 2) (bv 1 2) (bv 2 2) (bv 3 2)))
3
4  (define (calculate prog [acc (bv 0 4)])
5    (cond                        ; An interpreter for
6      [(null? prog) acc]         ; calculator programs.
7      [else                      ; A program is list of
8       (define ins (car prog))   ; '(op) or '(op arg)
9       (define op (car ins))     ; instructions that up-
10      (calculate                ; date acc, where op is
11       (cdr prog)               ; a 2-bit opcode and arg
12       (cond                    ; is a 4-bit constant.
13         [(eq? op Add) (bvadd acc (cadr ins))]
14         [(eq? op Sub) (bvsub acc (cadr ins))]
15         [(eq? op Sqr) (bvmul acc acc)]
16         [else         acc]))]))
17
18 (define (list-set lst idx val) ; Functionally sets
19   (match lst                    ; lst[idx] to val.
20     [(cons x xs)
21      (if (= idx 0)
22          (cons val xs)
23          (cons x (list-set xs (- idx 1) val)))]
24     [_ lst]))
25 (define (sub->add prog idx)        ; Replaces Sub with
26   (define ins (list-ref prog idx)) ; Add if possible.
27   (if (eq? (car ins) Sub)
28       (list-set prog idx (list Add (bvneg (cadr ins))))
29       prog))
30
31 (define (verify-xform xform N) ; Verifies the given
32   (define P                    ; transform for all
33     (for/list ([i N])          ; programs of length N.
34       (define-symbolic* op (bitvector 2))
35       (define-symbolic* arg (bitvector 4))
36       (if (eq? op Sqr) (list op) (list op arg))))
37   (define-symbolic* acc (bitvector 4))
38   (define-symbolic* idx integer?)
39   (define xP (xform P idx))
40   (verify ;  ∀ acc, idx, P. P(acc) = xform(P, idx)(acc)
41    (assert (eq? (calculate P acc) (calculate xP acc)))))
```

Fig. 2. A toy verifier in the Rosette solver-aided language. The performance bottleneck is the `list-set` procedure, originally used by Uhler and Dave to illustrate a symbolic evaluation anti-pattern they encountered when programming in the Smten solver-aided language [Uhler and Dave 2014].



Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
list-set xform.rkt:22 20 calls	3.3	19	2520	60	420	1700
verify-xform xform.rkt:36 1 call	1.4	12868	186	40	40	40
calculate xform.rkt:8 40 calls	1.3	41	380	180	0	280

Fig. 3. Output of the SymPro symbolic profiler when run on the program in Figure 2. Time spent in solver calls is collapsed into the dashed lines. SymPro identifies `list-set` as the performance bottleneck, ranking it highest in the *Score* column and highlighting it red.

2 OVERVIEW

This section illustrates symbolic profiling on a small solver-aided program (Figure 2). The performance bottleneck in this program is a recursive procedure (lines 18–24) used by Uhler and Dave [2014] to describe a common symbolic evaluation anti-pattern. We first show that time-based profiling fails to identify this procedure as the bottleneck, then apply symbolic profiling to identify, diagnose, and fix the issue.

A Small Solver-Aided Program. Our example program (Figure 2) is written in Rosette [Torlak and Bodik 2013, 2014], a solver-aided language that extends Racket with support for verification and

synthesis. Programs written in Rosette behave like Racket programs when executed on concrete values, but Rosette lifts their semantics, via symbolic evaluation, to also operate on unknown *symbolic* values. These symbolic values are used to formulate *solver-aided queries*, such as searching for inputs on which a program violates its specification (verification), or searching for a program that meets a given specification (synthesis). The example implements a tool that verifies optimizations for a toy calculator language.

The `calculate` procedure (lines 4–16) defines the semantics of the calculator language with a simple recursive interpreter. A calculator program is a list of instructions that manipulate a single 4-bit storage cell, `acc`. An instruction consists of a 2-bit opcode and, optionally, a 4-bit argument. The language includes instructions for adding to, subtracting from, and squaring the value in the `acc` cell.

The toy calculator language is also equipped with procedures for optimizing calculator programs. One such procedure, `sub->add` (lines 25-29), takes as input a program and an index, and if the instruction at that index is a subtraction, `sub->add` replaces it with an equivalent addition instruction.

To check that these optimizations are correct, we implement a tiny verification tool, `verify-xform` (lines 31–41), using Rosette's **`verify`** query. The tool first constructs a symbolic calculator program P (lines 32–36) that represents all syntactically correct concrete programs of length N. This is done using Rosette's `(`**`define-symbolic*`** `id type)` form, which creates a fresh symbolic constant of the given `type` and binds it to the variable `id` every time the form is evaluated. Next, the tool applies the `(`**`verify`** `expr)` form to check that the input optimization `xform` preserves the semantics of P for all values of `acc` and the application index `idx`. This form searches for a concrete interpretation of the symbolic constants that violates an assertion encountered during the (symbolic) evaluation of `expr`. As expected, no such interpretation, or *counterexample*, exists for the `sub->add` optimization and programs of length $N \leq 5$.

Performance Bottlenecks. Verifying `sub->add` for larger values of N produces no counterexamples either, but the performance of the `verify-xform` tool begins to degrade, from less than a second for $N = 5$ to a dozen seconds for $N = 20$. While such degradation is inevitable given the computational complexity of the underlying satisfiability query, we have the (usual) practical goal of extracting as much performance as possible from our tool. With this goal in mind, we would like to find out what parts of the code in Figure 2 are responsible for the degradation and how to improve them.

A first step in investigating the program's performance might be to use Racket's existing profiling support [Barzilay 2017], or any other time-based profiler. The Racket profiler reports that most time is spent in `verify-xform`. It also cannot elide the internal implementation details of Rosette from the profile, and so reports many spurious function calls that do not exist in the program as written. But even a Rosette-aware time-based profiler (essentially, the *Time* column in Figure 3) reports `verify-xform` as the hot spot in this program, with the majority of the execution time spent directly in this procedure—specifically, calling an SMT solver from within the **`verify`** form. While useful, this information is not *actionable*, since it does not tell us where to attempt optimizations. When the solver is taking most of the time, what we want to know is the following: are there are any inefficiencies in the program that are causing the symbolic evaluator to emit a hard-to-solve (e.g., unnecessarily large) encoding?

Symbolic Profiling. To help answer this question, our symbolic profiler, SymPro, produces the output in Figure 3 for $N = 20$. The table at the bottom of Figure 3 identifies `list-set` as the main bottleneck, highlighting it in red and ranking it highest in the *Score* column. The score is computed (as Section 4 describes) from five statistics that quantify the effect of a procedure on the symbolic heap and evaluation graph:

- *Time* is the exclusive wall-clock time spent in a call;
- *Term Count* is the number of symbolic values created during a call;

- *Unused Terms* is the number of those values that do not appear in the query sent to the solver (i.e., the symbolic equivalent of garbage objects);
- *Union Size* is the sum of the out-degrees of all nodes added to the evaluation graph; and,
- *Merge Cases* is the sum of the in-degrees of those nodes.

The chart at the top of Figure 3 visualizes the evolution of the call stack over time. Given this profile, it is easy to see that `list-set` has the largest effect on the heap and the evaluation graph, as well as the size of the final encoding, even though both `verify-xform` and `calculate` are slower. Running the profiler on this benchmark has only minimal overhead: 4% slowdown and 19% additional memory.

Diagnosis and Repair. But why does `list-set` perform poorly under symbolic evaluation, and how can we repair it? The output in Figure 3 shows that `list-set` creates many terms and performs many state merges. As noted by Uhler and Dave [2014] and described in Section 3, the core issue is algorithmic. In particular, the recursive call to `list-set` is guarded by a short-circuiting condition `(= idx 0)` that is symbolic when `idx` is unknown. The symbolic evaluation engine must therefore explore both branches of this conditional, leading to quadratic growth in the symbolic representation (i.e., the term count in Figure 3) of the output list:

```
> (define-symbolic* i integer?)
> (list-set '(1 2 3) i 4)
(list (ite (= 0 i) 4 1)
      (ite (= 0 i) 2 (ite (= 0 (- i 1)) 4 2))
      (ite (= 0 i) 3 (ite (= 0 (- i 1)) 3
                          (ite (= 0 (- i 2)) 4 3))))
```

The solution is to revise `list-set` to recurse unconditionally:

```
(define (list-set lst idx val)
  (match lst
    [(cons x xs)
     (cons (if (= idx 0) val x)
           (list-set xs (- idx 1) val))]
    [_ lst]))

> (list-set '(1 2 3) i 4)
(list (ite (= 0 i) 4 1)
      (ite (= 0 (- i 1)) 4 2)
      (ite (= 0 (- i 2)) 4 3))
```

With this revision, calls to `list-set` add at most $O(N)$ values to the symbolic heap, and the solving time for our verification query is cut in half for $N = 20$.

3 SYMBOLIC EVALUATION ANTI-PATTERNS

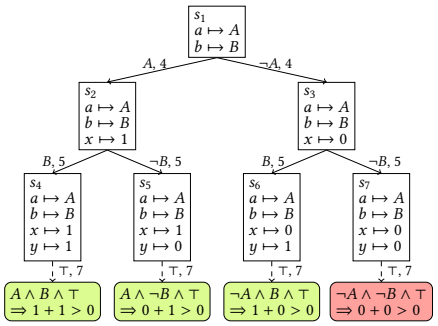
At the core of every symbolic evaluator is a strategy for reducing a program's semantics to constraints, and knowing what programming patterns are well or ill suited to an evaluator's strategy is the key to writing performant solver-aided code. This section presents three common *anti-patterns* that lead to poor performance under most evaluation strategies. We review the space of strategies first, followed by an illustration of each anti-pattern and a potential repair for it.

3.1 Strategies for Reducing Programs to Constraints

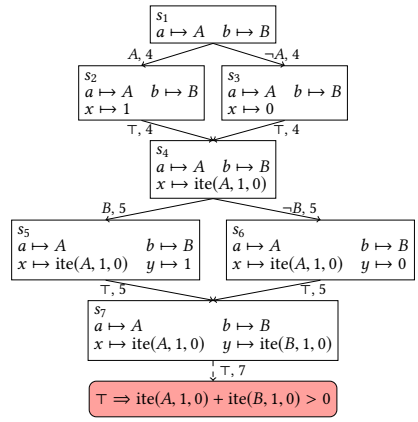
Symbolic evaluation engines rely on two basic strategies for reducing programs to constraints: *symbolic execution* (SE) [Clarke 1976; King 1976] and *bounded model checking* (BMC) [Biere et al. 1999]. There are engines that use just SE [Cadar et al. 2008; Godefroid et al. 2005, 2008] or just BMC [Babić and Hu 2008; Clarke et al. 2004; Xie and Aiken 2005] or a hybrid of the two [Ganai and Gupta 2008; Kuznetsov et al. 2012; Sen et al. 2015; Torlak and Bodik 2014]. We illustrate both SE and BMC on the program in Figure 4a, and briefly review a hybrid approach [Torlak and Bodik 2014]. For a more complete survey, we refer the reader to Torlak and Bodik [2014] or Cadar and Sen [2013].

```
1 (define-symbolic* a boolean?)
2 (define-symbolic* b boolean?)
3
4 (define x (if a 1 0))
5 (define y (if b 1 0))
6
7 (assert (> (+ x y) 0))
```

(a) A program with a simple invalid assertion.



(b) Symbolic execution explores each control flow path through a program separately, resulting in a tree-shaped symbolic evaluation graph.



(c) Bounded model checking merges states from different paths at every control-flow join, resulting in a symbolic evaluation DAG.

Fig. 4. An example of basic symbolic evaluation strategies. Symbolic execution and bounded model checking result in evaluation graphs of different shapes. Edge labels indicate the additional guard and the line of code that caused the transition.

Symbolic Execution. Symbolic execution (SE) reduces a program's semantics to constraints by evaluating and encoding individual paths through the program. Figure 4b shows the *symbolic evaluation graph* (defined in Section 4) created by applying SE to the sample program in Figure 4a. The nodes in the graph are program states, and the edges are transitions between states. Each edge is labeled with a guard and a program location that indicate where, and under what constraint, the transition is taken. The conjunction of all guards along a given path is called a *path condition*. The encoding of the program's semantics is the conjunction of the formulas $pc \Rightarrow \phi$ at the leaves of the symbolic evaluation tree, where pc is the path condition and ϕ is the assertion at the end of that path. This encoding is worst-case exponential in program size, which is the key disadvantage of SE. The crucial advantage of SE is that it maximizes opportunities for concrete evaluation (e.g., line 7 is evaluated concretely along each path), leading to simpler and easier-to-solve queries.

Bounded Model Checking. Bounded model checking (BMC) avoids the exponential explosion of SE by merging program states at each control flow join, as shown in Figure 4c. The resulting encoding of the program's semantics (i.e., the conjunction of the formulas at the leaves of the symbolic evaluation DAG) is polynomial in program size. The disadvantage of BMC, however, is the loss of opportunities for concrete evaluation. In our example, line 7 is evaluated symbolically, producing an encoding that requires reasoning about symbolic integers; the corresponding SE encoding, in contrast, uses only propositional logic. So while BMC encodings are compact, they are also harder to solve in practice.

Hybrid Approaches. Recent symbolic evaluation engines [Sen et al. 2015; Torlak and Bodik 2014] employ a hybrid of SE and BMC to offset the disadvantages of using either strategy on its own. These hybrid approaches generally prefer SE, applying BMC selectively to merge some paths and their corresponding states. For example, Rosette [Torlak and Bodik 2014] performs BMC-style

merging for values of the same primitive type; structural merging for values of the same shape (e.g., lists of the same length); and union-based merging (i.e., SE) for all other values:

```
(define-symbolic* b boolean?)

> (if b 1 0)           ; BMC-style merging
(ite b 1 0)
> (if b '(1 2) '(3 4)) ; structural merging
(list (ite b 1 3) (ite b 2 4))
> (if b 1 #f)          ; union-based merging (SE)
{[b 1] [(! b) #f]}
```

This evaluation strategy produces a compact encoding, like BMC, while creating more opportunities for concrete evaluation, like SE. But careful programming is still needed to achieve good performance, as we show next.

3.2 Three Anti-Patterns in Solver-Aided Programs

This section presents three kinds of *anti-patterns* in solver-aided code that lead to poor performance during symbolic evaluation. For each, we show an example of the issue and suggest potential repairs.

Algorithmic Mismatch. As observed by Uhler and Dave [2014], small algorithmic changes can have a large impact on the efficiency of symbolic evaluation. Consider, for example, the `list-set` algorithm in Figure 2 and the revised version presented in Section 2. The revised version is asymptotically better for engines that merge lists (e.g., [Torlak and Bodik 2014; Uhler and Dave 2014]). Yet the original version is asymptotically better when no merging of lists is performed (e.g., [Sen et al. 2015]). Such a *mismatch* between the algorithm and the underlying evaluation strategy can often be remedied with small changes to the algorithm's control flow. Symbolic profiling helps make these changes by informing the programmer of an algorithm's effect on the symbolic heap and the evaluation graph.

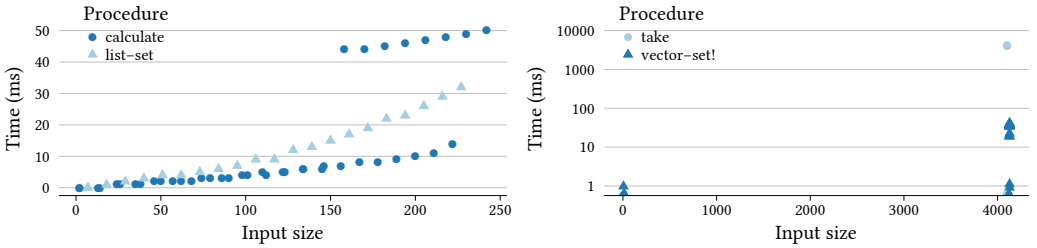
Irregular Representation. Poor choice of data structures is another frequent source of performance problems in solver-aided programs. Some common programming patterns, such as tree manipulations, require careful data structure design (see, e.g., [Chandra and Bodik 2018]) to yield an effective symbolic encoding. In general, performance issues arise when the representation of a data type is *irregular* (e.g., a list of length one *or* two), increasing the number of paths that need to be evaluated to operate on a symbolic instance of that type.

To illustrate, consider the instruction data type for the calculator language from Figure 2. Because an instruction is a list of the form `'(op)` or `'(op arg)`, applying `cadr` to a symbolic instruction at lines 13–14 involves evaluating two paths: one feasible (when the argument is present) and one infeasible (otherwise). Once the algorithmic mismatch in `list-set` is fixed, SymPro identifies this representational issue as the bottleneck by ranking `calculate` and `cadr` highest in the profile. Making the representation more regular—in our case, by replacing line 36 with `(list op arg)`—fixes the problem and leads to an additional 30% improvement in solving time. As this example illustrates, a less space-efficient but more uniform data representation is usually the better choice for symbolic evaluation.

Missed Concretization. In addition to employing careful algorithmic and representational choices, performant solver-aided code is also structured to provide as much information as possible about the feasible choices for symbolic values. Failing to make this information explicit is a common cause of bottlenecks that manifest as large evaluation graphs with many infeasible paths.

For example, consider the following toy procedure:

```
(define (maybe-ref lst idx) ; Return lst[idx]
  (if (<= 0 idx 1)          ; if idx is 0 or 1,
      (list-ref lst idx)    ; otherwise return -1.
      -1))
```



(a) Calculator (Figure 2) (b) Ferrite (§5.1; note logarithmic y-axis)

Fig. 5. Results from our input-sensitive profiling [Coppa et al. 2012] prototype applied to two programs with known bottlenecks. In (a), the outlier results for `calculate` promote it to be the most computationally complex procedure. In (b), the profiler cannot fit a good function for `take`, and so identifies `vector-set!` instead.

Applying this procedure to a list of size N and a symbolic index results in an evaluation graph with $O(N)$ paths, only three of which are feasible. Refactoring the code to make explicit the concrete choices for `idx` leads to an asymptotically smaller evaluation graph and encoding:

```
(define (maybe-ref-alt lst idx)
  (cond [(= idx 0) (list-ref lst 0)]
        [(= idx 1) (list-ref lst 1)]
        [else -1]))

(define-symbolic* idx integer?)
> (maybe-ref '(1 2 3 4 5 6) idx)     ; O(N) encoding
(ite (&& (<= 0 idx) (<= idx 1))
     (ite* (⊢ (= 0 idx) 1) ... (⊢ (= 5 idx) 6))
     -1)
> (maybe-ref-alt '(1 2 3 4 5 6) idx) ; O(1) encoding
(ite (= 0 idx) 1 (ite (= 1 idx) 2 -1))
```

In practice, this anti-pattern shows up in a more subtle form, where the feasible choices for a symbolic value are only known at run time. The fix then relies on the host platform to provide a facility for expressing the set of feasible choices to the symbolic evaluator. We show an example of this more subtle issue and the corresponding fix in Section 5.1.

4 SYMBOLIC PROFILING

This section presents *symbolic profiling*, a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation. As with any profiler, the key choice is what data to measure and where. We first review the space of alternative designs and then present our approach. We define the key parts of our performance model, the symbolic heap and evaluation graph; describe how a symbolic profiler analyzes them; and present two implementations of symbolic profiling.

4.1 Designing a Symbolic Profiler

To help programmers identify performance bottlenecks in the symbolic evaluation of their code, a profiler must satisfy three key objectives. First, its output must be *explainable*: it must provide a few key concepts that programmers can use to understand the behavior of their code under symbolic evaluation, without understanding the implementation details of the underlying engine. Second, its output must be *actionable*, pointing programmers to the root cause of any discovered bottleneck—a location in the code that needs to be repaired to improve performance. Finally, a symbolic profiling technique should ideally be *general*, to handle the wide variety of symbolic evaluation strategies (from SE to BMC) and applications (e.g., bug finding, verification, and synthesis). Drawing on existing profiling and symbolic evaluation research, we evaluated several potential symbolic profiler designs against these criteria before settling on our approach.

Input-Sensitive Profiling. Our first design was based on input-sensitive profiling [Coppa et al. 2012]. For each procedure in a program, input-sensitive profiling estimates its computational complexity as a function of its input size by fitting a function to its observed behavior. For example, such a profiler can determine that a linked-list traversal takes $O(n)$ time. Our intuition was that poor symbolic evaluation performance often comes from program locations that experience high complexity due to path explosion; a symbolic profiler could apply input-sensitive profiling and report the procedures with the worst computational complexity.

We implemented a prototype input-sensitive symbolic profiler to explore this hypothesis. However, we found that the correlation between input size and performance is often poor for code using symbolic evaluation. Minor perturbations in the input can cause the underlying engine to change its evaluation strategy, causing drastic changes in performance that make the estimated computational complexity inaccurate and noisy. For example, Figure 5 shows the results from applying our prototype to the calculator program in Figure 2 and the Ferrite case study in Section 5.1. For the calculator (a), the stratified results for `calculate` identify it as the most computationally complex function, even though `list-set` is the true bottleneck. For Ferrite (b), there is no good function to fit for `take`, and so the profiler prefers functions such as `vector-set!` with more available data. In both cases, noise obscures the true bottlenecks.

Path-Based Profiling. Our second design was inspired by the heuristics used in symbolic evaluation engines (e.g., [Kuznetsov et al. 2012]) to control path explosion. The resulting prototype symbolic profiler ranked functions based on the number of infeasible paths they explored and the total size of the path conditions generated during evaluation. Our intuition was that poorly performing procedures would generate many large, infeasible paths, and be likely candidates for repair.

However, this approach fell short in several ways. First, infeasible paths are not always the source of performance degradation. Applications such as program synthesis intentionally generate many large, feasible paths (e.g., to encode a sketch [Solar-Lezama et al. 2006]), making this analysis ineffective. Second, when they are required, feasibility checks must be discharged by a constraint solver and so are extremely expensive; we observed profiler overheads of 100× on even simple benchmarks. Finally, a path-based profiler does not generalize to BMC-style evaluation, where performance bottlenecks manifest in the creation of large symbolic values during state merging, as illustrated in Section 2 for the `list-set` procedure.

With these experiences in mind, we sought to identify a performance model for symbolic profiling that would offer actionable advice in terms of a few abstract concepts, and accommodate the full spectrum of symbolic evaluation approaches. The remainder of this section describes our chosen design; our case studies in Section 5 and evaluation in Section 6 measure its success against these objectives.

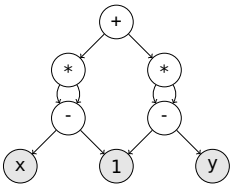
4.2 Instrumenting Symbolic Evaluation

Our key insight is that every symbolic evaluator can be understood in terms of its interaction with two abstract data structures, the symbolic heap and the symbolic evaluation graph, which form our performance model for symbolic profiling. While explicit in our presentation, these data structures are usually implicit in an evaluator's implementation. We therefore also define a simple interface that any evaluator can implement to enable a symbolic profiler to reconstruct both data structures on the fly. The next section shows how to analyze these structures to produce actionable profile data.

Symbolic Heap. As an evaluator creates new symbolic values to reflect the program's semantics, it implicitly constructs a *symbolic heap*. For profiling purposes, the symbolic heap is analogous to the concrete heap: procedures that allocate many values on the heap are candidate bottlenecks.

```
1  (define-symbolic* x y integer?)
2  (define dist ; squared dist from <x, y> to <1, 1>
3    (+ (* (- x 1) (- x 1))
4       (* (- y 1) (- y 1))))
```

(a) A program that computes the squared distance between a symbolic and concrete point.



(b) The symbolic heap for the program.

Fig. 6. The *symbolic heap* of a program tracks the structure of allocated symbolic values.

Definition 1 (Symbolic heap). *A* symbolic heap *is a directed acyclic graph* (V, E) *where each vertex* $v \in V$ *is a* term. *A term is either a* concrete constant, *or a* symbolic constant, *or an* expression. *Constants have no outgoing edges. Expressions are labeled with an operator, and have an outgoing edge to each of their subterms. Each symbolic constant and expression is annotated with a program* location $\mathrm{l}(v)$ *that created the term.*

For example, consider the program in Figure 6a. The first line allocates two new symbolic constants x and y of type integer and binds them to the variables `x` and `y`. The second line constructs expressions out of these symbolic constants and the concrete constant `1`. The resulting symbolic heap, shown in Figure 6b, comprises five expressions and three constants. The symbolic constants x and y have locations $\mathrm{l}(x) = \mathrm{l}(y) = 1$, while the expressions have locations $\mathrm{l}(\cdot) = 3$. Most symbolic evaluators use canonicalization (e.g., [Ershov 1958]) to improve sharing of symbolic terms, so only one instance of the terms $(- x\ 1)$ and $(- y\ 1)$ is usually constructed.

Symbolic Evaluation Graph. Where the symbolic heap reflects the flow of data during symbolic evaluation, the *symbolic evaluation graph* captures the control flow. This graph reflects the engine's evaluation strategy—where it explored multiple paths separately, and where those paths were merged together. By analyzing the symbolic evaluation graph, a symbolic profiler can identify candidate bottlenecks with significant branching or merging activity.

Definition 2 (Symbolic evaluation graph). *A* symbolic evaluation graph *is a directed acyclic graph* (V, E) *in which each vertex* $s \in V$ *is a state of the program. Each edge* $(s_i, s_j) \in E$ *is a transition between two program states, and is annotated with a* location $\mathrm{l}(s_i, s_j)$ *reflecting the point in the program that caused the transition, and a* guard constraint $\mathrm{guard}(s_i, s_j)$ *reflecting the condition under which the transition was taken.*

As an example, consider again the program in Figure 4a. Different evaluation strategies will produce different symbolic evaluation graphs for this program. In symbolic execution (Figure 4b), each **`if`** statement will cause execution to diverge into two paths that never meet, and so the definition of `y` (line 5) will execute twice and the assertion (line 7) four times. On the other hand, bounded model checking (Figure 4c) will immediately merge the two paths generated by each **`if`**, and so both the definition of `y` and the assertion will execute only once, on merged states.

Symbolic Profiling Interface. Most symbolic evaluators create the symbolic heap and evaluation graph only implicitly—the heap is implicit in a canonicalization cache, and the graph in the evaluator's control flow. To enable a symbolic profiler to track the evolution of these data structures, we define a generic *symbolic profiler interface* that engines should implement at important points in symbolic evaluation. A symbolic profiler can construct the symbolic heap and evaluation graph by instrumenting calls to this interface. In practice, these calls are often already implemented by

symbolic evaluators, simplifying adoption. Section 4.4 describes implementations of the interface in two different symbolic evaluators.

Definition 3 (Symbolic profiler interface). *The* symbolic profiler interface *comprises five instrumentation points that a symbolic evaluator should implement to expose profiling data:*

- new(x, l) *Allocate a fresh symbolic constant named x at program location l.*
- new$(op, x_1, \ldots, x_n, l)$ *Allocate a new expression $op(x_1, \ldots, x_n)$ at program location l, where each x_i is a concrete constant or a previously allocated symbolic term.*
- step$(s_0, \langle g_1, e_1\rangle, \ldots, \langle g_n, e_n\rangle)$ *Starting from program state s_0, evaluate each program expression e_i (annotated with a corresponding program location l_i) under the guard g_i. Return a list of the resulting states $s_1, \ldots, s_k$, where $k \geq n$.*
- merge$(s_1, \ldots, s_m, l)$ *Merge the states $s_1, \ldots, s_m$ at program location l using the evaluator's merging strategy, returning a new list of states $s'_1, \ldots, s'_j$, where $1 \leq j \leq m$.*
- solve(x, l) *Call a constraint solver at program location l to determine the satisfiability of the expression x.*

The two new calls in the profiler interface construct the symbolic heap. The symbolic evaluator invokes new each time it allocates a new symbolic term—either a fresh symbolic constant or an expression. The profiler adds the corresponding new node to the symbolic heap, with edges to the relevant (immediate) subterms if the new term is an expression.

The step and merge calls in the profiler interface reconstruct the symbolic evaluation graph. The symbolic evaluator calls step to evaluate a set of expressions (e.g., two branches of a conditional) under disjoint and exhaustive guards. It calls merge to merge a set of states, usually at a control-flow join point. For example, at line 4 in Figure 4a, the evaluator invokes step$(s_1, \langle a, \texttt{1}\rangle, \langle \neg a, \texttt{0}\rangle)$, which adds the edges $\langle s_1, s_2\rangle$ and $\langle s_1, s_3\rangle$, with the guards a and $\neg a$, to the evaluation graph. From this point, different evaluation strategies result in different calls to the profiler interface. A symbolic execution engine (Figure 4b) never calls merge, instead evaluating each path separately by calling step twice for line 5 (once per path). A bounded model checker (Figure 4c) immediately calls merge$(s_2, s_3, 4)$ to merge the two states at line 4, producing a single new state s_4. In either case, by instrumenting the step and merge calls, a symbolic profiler can reify the (otherwise implicit) evaluation graph.

Finally, the solve call in the interface allows a profiler to determine which parts of the symbolic heap flow to a constraint solver. The symbolic evaluator invokes solve(x, l) whenever it solves a constraint x, either to check the feasibility of a path condition or to discharge a solver-aided query. In the next section, we use this data to analyze the symbolic heap for terms unseen by the solver, which can indicate wasted allocations.

4.3 Analyzing a Symbolic Profile

Our symbolic profiler, SymPro, analyzes the symbolic heap (Definition 1) and evaluation graph (Definition 2) in three ways to present suggestions to users: computing *summary statistics* about each procedure in the program; determining *data flow* through the program to identify wasted allocations and work; and *ranking* procedures based on these two analyses to identify the most likely bottlenecks.

Summary Statistics. SymPro computes four summary statistics about each procedure call:

- *Time* is the exclusive wall-clock time spent in the call;
- *Term count* is the number of symbolic terms added to the symbolic heap;
- *Union size* is the sum of the out-degrees of all nodes added to the symbolic evaluation graph;
- *Merge cases* is the sum of the in-degrees of those nodes.

These statistics summarize the key aspects of symbolic evaluation: the time spent in each procedure; the size of the symbolic state allocated; how many times path splitting (symbolic execution) was performed; and how many times merging (bounded model checking) occurred.

Data Flow. In addition to computing the summary statistics, SymPro uses the symbolic heap and the solve instrumentation to determine which terms in the heap are "used" by the program. To a first approximation, terms in a solver-aided program are not useful if they are never sent to the underlying constraint solver as part of a feasibility check or a solver-aided query. SymPro exploits this observation to produce an analysis of *unused terms* in the program. For each solve(x, l) call made by the symbolic evaluator, SymPro computes the set of all terms in the symbolic heap that are transitively reachable from x. Any term y that is in none of these transitive closures is unused: it is not part of any constraint sent to the solver.

Unused terms indicate either dead code (terms that were created but never used) or simplification by the symbolic evaluator. For example, consider the following program:

```
(define-symbolic* x boolean?) ; add x to the heap
(define A (or x (not x)))     ; add ¬x to the heap
> A                           ; but simplify x ∨ ¬x
#t
> (solve (assert A))          ; both x and ¬x unused
```

SymPro would report the terms x and $\neg x$ as unused, because the evaluator simplified them out of the query sent to the solver. Both causes of unused terms represent optimization opportunities: dead code should be removed, while excessive simplification suggests redundancy that a better algorithm or encoding could eliminate.

Ranking. Based on the summary statistics and data flow analysis, SymPro ranks each procedure in the program to suggest the most likely bottlenecks to the user. It first normalizes each statistic (time, term count, union size, merge count, and unused terms) to the range 0–1. Then it assigns each procedure a score by summing the normalized statistics. This score, a number between 0 and the number of statistics, is a simple ranking of which procedures do the most symbolic work. Our case studies in Section 5 and evaluation in Section 6 show this ranking is highly effective for navigating symbolic profiles. We also experimented with a machine-learned ranking scheme, training a binary classifier (a support vector machine) to identify bottlenecks using some of the benchmarks from Table 2 as training data. The resulting classifier had high recall but poor precision, identifying many false positive bottlenecks. For that reason, and because our manual ranking scheme is easier to explain, SymPro uses that scheme as the default.

4.4 Implementation

We have implemented the symbolic profiler interface (Definition 3) in two different symbolic evaluators—a fully featured implementation for the Rosette solver aided language [Torlak and Bodik 2013, 2014], and a proof of concept one for the Jalangi JavaScript analysis framework [Sen et al. 2013, 2015].

Rosette. Our Rosette profiler instruments several key points in Rosette's evaluation engine, most of which are directly analogous to the calls in the profiler interface. To implement the new interface, we instrument Rosette's term creation cache, which performs hash-consing to canonicalize terms. To implement step, we record the creation of *symbolic unions*, which Rosette uses to track the multiple possible values of a variable during symbolic evaluation. Finally, to implement merge and solve, we instrument Rosette's corresponding `merge` and `solver-check` procedures. This instrumentation changes only 21 lines of the Rosette engine implementation. The code for the SymPro analyses

comprises 1,000 lines of Racket and 1,400 lines of TypeScript. The Rosette profiler is open-source and integrated into the latest Rosette release [Torlak 2018].

Jalangi. Jalangi [Sen et al. 2013] uses a symbolic execution engine called MultiSE [Sen et al. 2015] to provide concolic test generation for JavaScript. We modified MultiSE to implement the symbolic profiler interface as follows. To implement new, we track calls to the constructors of symbolic term objects (strings, numbers, and booleans). MultiSE rewrites JavaScript programs with additional control flow to implement step, and so we track each time a new path is generated from a branch in the program. To instrument merge, we modify MultiSE's implementation of *value summaries*, which are lists of guard-value pairs reflecting the possible values of each variable. Section 6.4 presents our results with this proof-of-concept profiler.

4.5 Discussion

Two kinds of performance issues are outside the scope of symbolic profiling, which focuses on analyzing the behavior of symbolic evaluation. First, if the bottleneck is in constraint solving, a symbolic profiler can report that solving is taking the most time, but it cannot identify the cause of the issue. Second, while bottlenecks in concrete execution can be identified by symbolic profiling (which includes a measure of execution time), they will be ranked below symbolic evaluation bottlenecks because they cause no activity in the symbolic heap and evaluation graph.

Some bottlenecks can be repaired in multiple ways at different locations within a program; when symbolic profiling identifies such a bottleneck, it may not suggest the easiest location to repair. For example, consider this program, with a symbolic boolean input `b` passed to `outer`:

```
(define (outer b)
  (when b
    (inner)))
(define (inner)
  ...)
```

Suppose the `inner` function has side effects (e.g., mutating global variables) that result in a bottleneck. Symbolic profiling will identify `outer` as the bottleneck, but the issue could be repaired by modifying either `outer` (to not call `inner` under a symbolic path condition) or `inner` (by mutating less global state). As another example, irregular data representations (Section 3.2), such as the one on line 36 of Figure 2, are most easily repaired where the data is constructed, even though symbolic profiling will identify the location the data is used (lines 13–14 of Figure 2) as the bottleneck.

5 ACTIONABILITY: CASE STUDIES

To demonstrate that SymPro produces *actionable* profiles, we performed a series of case studies on real-world Rosette programs. We collected a suite of benchmarks by performing a literature survey of all papers citing Rosette [Torlak and Bodik 2014]. This suite, shown in Table 1, comprises all 15 tools that were open source (or that the authors made available to us) and that ran on the latest Rosette release.

We applied SymPro to each benchmark and used it to identify 8 performance bottlenecks summarized in Table 2. This section presents our results, with three in-depth case studies and brief overviews of four other findings. In each case study, we highlight a bottleneck found by SymPro, relate it to the anti-patterns of Section 3, and present repairs. Six of the eight bugs we found were in code bases with which we were not previously familiar. Section 6 evaluates SymPro against our other design criteria, *explainability* and *generality*.

		Time		Peak Memory	
Benchmark	LoC	Time (sec)	Slowdown	Memory (MB)	Overhead
Bagpipe [Weitz et al. 2016]	3317	16.1	51.1%	314	30.9%
Bonsai [Chandra and Bodik 2018]	641	55.1	22.1%	341	128.7%
Cosette [Chu et al. 2017b]	2709	12.8	7.4%	296	17.6%
Ferrite [Bornholt et al. 2016]	350	21.5	2.7%	690	5.0%
Fluidics [Willsey et al. 2018]	145	17.7	5.7%	198	18.9%
GreenThumb [Phothilimthana et al. 2016]	934	2358.5	0.1%	2258	0.0%
IFCL [Torlak and Bodik 2014]	574	96.4	0.7%	248	28.7%
MemSynth [Bornholt and Torlak 2017]	3362	24.0	45.7%	349	33.5%
Neutrons [Pernsteiner et al. 2016]	37317†	45.3	14.8%	1702	98.4%
Nonograms [Butler et al. 2017]	6693	15.1	3.1%	300	18.6%
Quivela [Amazon Web Services 2018]	5946	78.6	1.4%	496	20.0%
RTR [Kazerounian et al. 2018]	2007	374.6	12.6%	822	35.5%
SynthCL [Torlak and Bodik 2014]	3732	27.7	61.2%	445	133.2%
Wallingford [Borning 2016]	3866	7.9	2.4%	618	86.3%
WebSynth [Torlak and Bodik 2014]	2057	14.2	47.7%	467	122.8%

† Includes a 36,847-line Racket file automatically generated from the software being verified, which SymPro must instrument.

Table 1. Rosette benchmarks used in our evaluation. LoC is lines of code. Performance results show the overhead of SymPro's analysis as the average of five runs; 95% confidence intervals for overhead are < 5 pp.

5.1 File System Crash-Consistency

Ferrite [Bornholt et al. 2016] is a tool for reasoning about crash safety of programs running on modern file systems, which offer only weak consistency semantics. It consists of a verifier and a synthesizer. Given a *litmus test* program (i.e., a small, straight-line sequence of system calls), and a specification of crash safety for it, the verifier checks whether the program satisfies the safety specification under the relaxed semantics of a file system such as ext4, even in the face of crashes. If not, the synthesizer attempts to repair the program by inserting barriers (i.e., calls to `fsync`).

Ferrite represents files as a backing store (a list of bytes) together with the length of the file:

```
(struct file (contents length) #:transparent)
(define BLOCK_SIZE 4096)
(define F (file (make-list BLOCK_SIZE #x00) 0))
```

To model a write to the file `F`, which persists only if the system does not crash, Ferrite introduces a symbolic boolean value *crash?* to represent a non-deterministic crash:

```
(define N 2)
(define-symbolic* crash? boolean?)
(unless crash?          ; If not crashed
  (match-define (file contents length) F)
  (define new-contents  ; write 0x1 to first N bytes
    (append (make-list N #x01) (drop contents N)))
  (set! F (file new-contents (+ length N))))
```

To check the safety specification, Ferrite retrieves the final contents of the file:

```
(define cnts (take (file-contents F) (file-length F)))
(assert (or (equal? cnts '()) (equal? cnts '(1 1))))
```

This implementation is sufficient to verify crash safety at small block sizes (e.g., 32 bytes). But since many crash consistency bugs rely on boundary conditions around the size of disk blocks, Ferrite sets `BLOCK_SIZE` to a realistic value for a modern device (here, 4 kB). With this block size, even simple litmus tests cannot be verified (or repaired) in reasonable time.

Identifying the Bottleneck. SymPro identifies the call to `take` in the final step above as the source of poor performance. It ranks `take` high based on its large number of created symbolic terms and

Program	Anti-Pattern	Description	Speedup
Bonsai	Irregular representation	Shape of tree data structure is enumerated multiple times (§5.4)	1.35×
Cosette	Missed concretization	Possible table sizes are enumerated in a nested loop (§5.2)	> 6×[†]
	Algorithmic mismatch	Inefficient reduction builds a complex intermediate list (§5.2)	75×
Ferrite	Missed concretization	Length of an array is merged despite few feasible values (§5.1)	24×
Fluidics	Irregular representation	Grid data structure implemented with nested mutable vectors (§5.4)	2×
Neutrons	Irregular representation	Log of possible paths is maintained symbolically (§5.3)	290×
Quivela	Missed concretization	Object references are merged and obscure dynamic dispatch (§5.4)	29×
RTR	Algorithmic mismatch	Unnecessary fold over list of symbolic length (§5.4)	6×

[†] Without the repair, Cosette does not terminate within one hour.

Table 2. Summary of performance bottlenecks found by applying SymPro to the benchmarks in Table 1, together with the speedups obtained by repairing them.

the fact that almost none of those terms reach the solver. In contrast, a time-based profiler ranks the subsequent `equal?` call as the hottest method.

Diagnosing the Bottleneck. The root cause of this issue is a *missed concretization*. Rosette merges the second input to `take`, representing the length of the file, into a symbolic term of the form (ite *crash?* 0 2). When `take` receives a symbolic length argument, it performs symbolic execution, generating one path per potential length of the returned list. Since the input list `(file-contents F)` has length `BLOCK_SIZE` = 4096, the `take` call generates 4097 distinct paths, each with a path condition of the form (ite *crash?* 0 2) = n for $0 \leq n \leq 4096$. All but two of these paths are infeasible.

Repairing the Bottleneck. To repair the program, we recover the feasible concrete values for the file's length in two steps. First, we remove the `#:transparent` annotation from the definition of the `file` data type, to prevent structural (field-wise) merging of files. Instead, Rosette will use symbolic unions (Section 3.1) to merge files. Second, we use Rosette's `for/all` annotation to evaluate the `take` call separately for each value in the symbolic union `F`:

```
(define contents
  (for/all ([f F])
    (take (file-contents f) (file-length f))))
```

The `for/all` annotation is Rosette's *symbolic reflection* facility [Torlak and Bodik 2014], which allows programmers to control path splitting and merging. By default, Rosette evaluates the arguments to `take` first, merges the results, and then applies `take` once to the merged value. The `for/all` annotation tells Rosette to instead apply `take` to each possible value of `F` separately and then merge the results. This repair speeds up Ferrite by 24×, enabling it to replicate—in just a few minutes—a complex ext4 delayed allocation bug in Google Chrome [Boichat 2015].

5.2 SQL Query Equivalence Verification

Cosette [Chu et al. 2017a,b] is an automated prover for deciding the equivalence of two SQL queries. It uses Rosette to search for small counterexamples to equivalence, and Coq to construct proofs of equivalence if no counterexample is found.

Cosette's counterexample finder works by constructing a symbolic representation of a SQL table as a bag of tuples. Both the multiplicity of each tuple and its constituent elements are symbolic values. To execute a query against a table, Cosette constructs a new table in which the multiplicity of each tuple reflects the semantics of the query. For example, the result of executing the query `SELECT A FROM table WHERE C="a"` on a table is another table:

A	B	C	#
e_0	e_1	e_2	c_0
e_3	e_4	e_5	c_1

$\Longrightarrow$

A	#
e_0	`(if (= `e_2 `"a") `c_0 ` 0)`
e_3	`(if (= `e_5 `"a") `c_1 ` 0)`

To check if two queries are equivalent, Cosette executes each query on the same symbolic table, constructs a constraint asserting the two resulting tables are different, and solves this constraint using Rosette. Cosette makes extensive use of advanced Rosette features, including `eval` of dynamically generated code, making manual reasoning about performance particularly challenging.

A recent change to Cosette adjusted its encoding of SQL `WHERE` clauses to accommodate a richer subset of SQL's filter syntax. Previously, Cosette implemented filtering by removing the appropriate tuples from the bag; the change instead filters by setting those tuples' multiplicities to zero. After making this change, a Cosette benchmark that previously returned in under 15 seconds no longer returned within an hour. Our initial investigation showed the SMT solver was never called, suggesting the bottleneck was in symbolic evaluation, but offered no further details.

Identifying the Bottleneck. To identify the source of this bottleneck, we used SymPro's support for streaming profile data during execution. The streaming profiler applies the analyses in Section 4 incrementally as the symbolic heap and symbolic evaluation graph evolve, and periodically sends the resulting data to the profiler interface. For Cosette, the profiler implicated the following call to the `filter` function:

```
(map (lambda (t)
       (sum (filter (lambda (r) (eq? t r)) table)))
     table)
```

The profiler ranked these `filter` calls far above any other calls in the program due to their high number of new terms allocated on the symbolic heap and large numbers of merges in the symbolic evaluation graph.

Diagnosing the Bottleneck. This bottleneck is caused by a combination of two issues, a *missed concretization* and an *algorithmic mismatch*, which manifest as two distinct sources of path explosion.

The missed concretization is due to `table` being a symbolic union, reflecting the table's value along several control-flow paths generated by symbolic execution. The nested use of `table` thus creates quadratic path explosion—for each path in `table` explored when calling `map`, the evaluator explores every path in `table` when evaluating the inner `filter`.

The algorithmic mismatch is due to using `filter` to create an intermediate list just to `sum` its contents. The predicate used by `filter` depends on symbolic state, and so there are $O(2^N)$ paths for the return value of `filter`, as in the toy example from Figure 1. The `sum` procedure must then run once for each such path.

Repairing the Bottleneck. An easy repair for the missed concretization is to apply symbolic reflection:

```
(for/all ([table table])
  (map (lambda (t)
         (sum (filter (lambda (r) (eq? t r)) table)))
       table))
```

Here, the `for/all` evaluates its body once for each path in `table`. During each such evaluation, `table` is bound to a single concrete value rather than a union, avoiding the first source of path explosion. With this repair, the problematic benchmark completes within 10 minutes—better than non-termination but still worse than the original version of Cosette.

To repair the algorithmic mismatch, we avoid building the intermediate list with `filter`. Instead, the procedure passed to `map` performs a fold over `table` to sum the values that satisfy the `filter` predicate. With this additional repair, the problematic benchmark completes in 8 seconds—faster than even the original Cosette implementation. We reported the regression to the Cosette developers, and they accepted our patch.

5.3 Safety-Critical System Verification

Neutrons [Pernsteiner et al. 2016] is a tool for verifying the safety of a radiotherapy system in clinical use. The system is controlled by a large program written in the EPICS dataflow language [EPICS 2017]. Neutrons provides a symbolic interpreter for EPICS programs, and a verifier (built with Rosette) to check that EPICS programs satisfy key safety properties. The Neutrons verifier is used for active development of the system's software, so its performance is important for developer use.

Identifying and Diagnosing the Bottleneck. We used SymPro to profile the Neutrons symbolic interpreter, and found a bottleneck with the interpreter's tracing feature. As the interpreter executes an EPICS program, it records each executed instruction in a trace—a list of executed instructions—which is used to visualize counterexamples:

```
(define (record-trace msg)
  (set! trace (append trace (list msg))))
```

However, since this call is made with a symbolic path condition, Rosette must merge the new and existing values of `trace` when performing the mutation. This leads to excessive path creation and merging, since Rosette will track each potential length of `trace` separately by symbolic execution, and the length of `trace` depends upon the execution path. In essence, `trace` has an *irregular representation*. SymPro identifies this tracing procedure as the key bottleneck.

Repairing the Bottleneck. To improve this program, we observe that tracking the shape of the trace is unnecessary for counterexample visualization. For each executed instruction, we need only record the path condition that was true when the instruction executed, together with the instruction:

```
(define (record-trace msg)
  (raw-set! trace (append trace (list (cons (pc) msg)))))
```

Here, `(pc)` retrieves the current path condition, and `raw-set!` is Racket's unlifted implementation of `set!` that overwrites `trace` without any merging. The `trace` is now a list of every instruction executed by *any* possible interpretation of the EPICS program. When using this trace to visualize a counterexample, we simply hide any instruction whose corresponding path condition is not satisfied by the counterexample. This program transformation—which essentially adjusts the `trace` list to always have a concrete length—improves Neutrons' verification performance by 290× on a representative example. We reported this issue to the Neutrons developers, and they accepted our patch.

5.4 Other Findings

Our other findings in Table 2 include examples of all three anti-patterns presented in Section 3.2.

Type System Soundness Checking. Bonsai [Chandra and Bodik 2018] is a synthesis-based tool for checking the soundness of type systems. It uses a novel tree representation for type checking, and has been used to replicate a soundness bug in the Scala type system. We applied SymPro to Bonsai and found two *irregular representation* issues. First, Bonsai represents trees as nested lists; since the trees have unknown size, these lists are merged into a symbolic union. When the tree is used multiple times during the same type checking call, the symbolic evaluator enumerates the members of this union once per use and merges the results. Instead, we used Rosette's `for/all` facility to perform this enumeration only once, as done in the Cosette case study. Second, each (recursive) type checking step can return either a subtree or a boolean (in case of failure), which Rosette will always merge into a symbolic union due to their different types. Instead, we used multiple return values to separate the returned boolean failure flag from the returned subtree. Together, these changes improved Bonsai's performance by 35% when checking the Scala type system.

Cryptographic Protocol Verification. Quivela [Amazon Web Services 2018] is a tool for verifying the security of cryptographic protocols. It takes as input an implementation and a specification

of a cryptographic protocol, along with a series of refinement steps between them, and checks that each refinement is valid. We applied SymPro to Quivela and identified a *missed concretization* issue. Quivela represents protocols in a simple object-oriented language in which all method calls are virtual; each object can store references to other objects, which Quivela represents as integer addresses. Because these references are integers, the symbolic evaluator's default strategy is to merge them. But the merged references obscure the targets of virtual method calls forcing the engine to evaluate many infeasible paths, as in the Ferrite case study. We modified Quivela to instead track references concretely, by wrapping references into an opaque structure type that cannot be merged. This change improved Quivela's verification performance by up to 29× on small benchmarks, and allowed it to quickly verify larger protocols that previously caused out-of-memory failures.

Microfluidics Control Synthesis. Fluidics [Willsey et al. 2018] is a prototype tool for synthesizing programs that control a digital microfluidics array, used for executing biological wet-lab protocols. It takes as input the initial arrangement of samples on the array, and the desired final arrangement (potentially including mixtures of the samples), and synthesizes a series of movement and mixing instructions that produce the desired outcome. We applied SymPro to Fluidics and identified an *irregular representation* issue. Fluidics represents the state of the array as a two-dimensional vector of vectors, indexed by y and then x coordinates. However, this nested structure makes updates to the array expensive: because vectors are mutable data structures, the inner vector must be duplicated for each update to correctly track later mutations. Replacing the nested data structure with a flat one-dimensional vector improves Fluidics' performance by 2×, allowing it to synthesize more complex control programs and reason about larger microfluidics arrays.

Refinement Type Checker for Ruby. RTR [Kazerounian et al. 2018] is a type checker for a new refinement type system for Ruby. It takes as input a Ruby program translated to a Rosette-based intermediate verification language, and checks that user-specified refinement types hold in the (translated) program. The RTR verification language reflects Ruby's object structure and control-flow constructs. We applied SymPro to RTR and identified an *algorithmic mismatch* issue in the way RTR initializes new Ruby objects. In Ruby, an array initialization supplies a length together with an anonymous function (a "block") defining the value at each index:

```
Array.new(5){ |i| i*2 }
#=> [0, 2, 4, 6, 8]
```

RTR represents arrays as a pair of a vector (holding the array's contents) and an integer (holding the array's actual length). To support bounded verification, array lengths can be symbolic. RTR's array initialization creates a separate vector/integer pair for each possible length of the array, taking quadratic time. SymPro identifies the array initialization procedure as the bottleneck. We repaired this issue by initializing a concrete vector of length equal to the upper bound on the symbolic length; since RTR already tracks each list's length separately, the extraneous elements can simply be ignored. This repair improved RTR's performance on its slowest benchmark (Matrix) by 6×, from 6.1 minutes to 61 seconds, and reduces its peak memory usage by 3×. RTR's developers accepted our patch.

6 EXPLAINABILITY, GENERALITY, AND PERFORMANCE: EXPERIMENTS

To evaluate the performance, explainability, and generality of symbolic profiling, we sought to answer four research questions:

(1) Is the overhead of symbolic profiling reasonable for development use?
(2) Is the data collected by SymPro necessary for correctly identifying bottlenecks?
(3) Are programmers more effective at identifying bottlenecks with SymPro?
(4) Is SymPro effective at profiling different symbolic evaluation engines?

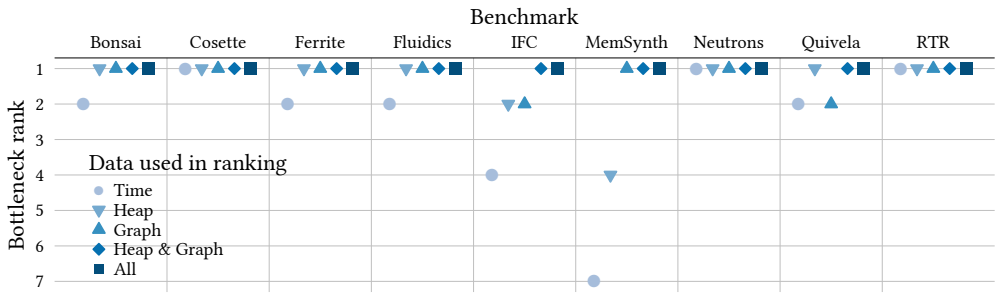


Fig. 7. Sensitivity of profiler rankings to the data sources used in ranking. Each point is the rank of the bottleneck function in a profile of the program when using the specified data in the ranking function. Benchmarks are those in which we found new bottlenecks (Table 2) or had previously known bottlenecks.

The first two questions address the key performance aspects of SymPro—run-time overhead and the necessity of the collected data for generating actionable feedback. The third question evaluates the explanatory power of SymPro's profiles. The fourth question assesses the generality of our approach. We use the Rosette profiler to investigate the first three questions, and the Jalangi profiler for the fourth. We find positive answers to all four questions.

6.1 Is the overhead of symbolic profiling reasonable for development use?

Table 1 shows the time and memory overheads for SymPro on a collection of real-world Rosette programs. All results were collected using an AMD Ryzen 7 1700 eight-core processor at 3.7 GHz and 16 GB of RAM, running Racket v6.12. For each benchmark we report the average overhead across five runs; 95% confidence intervals are below 5 percentage points for all overhead results.

Overall, SymPro slows applications by 0.1%–61.2% (geometric mean 16.9%), and increases peak memory use by 0.0%–133.2% (geometric mean 45.6%). These overheads are reasonable for development use, and are better than other tracing-based profiling tools. For example, the Racket version of profile-guided metaprogramming [Bowman et al. 2015] averages 4–12× slowdown, and input-sensitive profiling [Coppa et al. 2012] averages 30× slowdown for C programs. The highest overheads occur for benchmarks with many short-lived recursive calls. It would be possible to implement a sampling-based profiler if this overhead were to become unacceptable.

6.2 Is the data collected by SymPro necessary for correctly identifying bottlenecks?

To understand the importance of the data SymPro gathers, we performed a sensitivity analysis using all benchmarks in which we identified new bottlenecks (Table 2), as well as a collection of benchmarks with previously known bottlenecks. For each benchmark, we manually investigated its profile to identify a single procedure we believe should be ranked as the primary cause of poor performance. We then varied the data available to SymPro, giving it access to only wall-clock time, only the symbolic heap, only the symbolic evaluation graph, or combinations of the three components.

Figure 7 shows the results of the sensitivity experiment. For each benchmark, the y-axis measures the ranking of the known bottleneck when using only the specified source of profiling data. These results have three key highlights. First, timing data ● alone (i.e., the time spent in each procedure) identifies only three of nine bottlenecks. Second, no single data source is sufficient to identify the key bottleneck in every benchmark. While the symbolic heap ▼ and evaluation graph ▲ are each more effective than time alone, both are required ◆ to correctly rank all bottlenecks. Third, once both the symbolic heap and evaluation graph are available, including timing data ■ does not improve the quality of the rankings. However, SymPro still includes timing data in rankings, to help profile the parts of programs that do not perform symbolic evaluation.

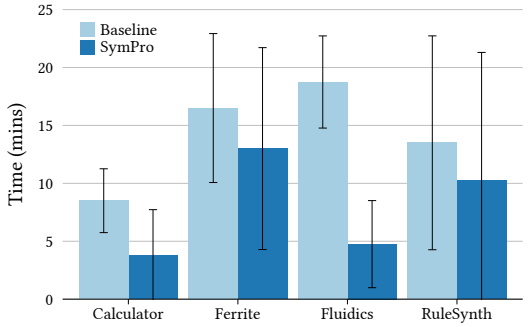


Fig. 8. Average time taken for users to identify the performance issue in four benchmarks, with or without SymPro. Error bars are 95% confidence intervals for $n = 4$ users.

6.3 Are programmers more effective at identifying bottlenecks with SymPro?

To help understand how effective SymPro is in real-world use, we conducted a small user study with Rosette programmers. Our study had eight graduate student participants, who each had previous Rosette experience ranging from "a few hours" to multiple published papers using Rosette. We first provided each participant a short tutorial on how to use both SymPro and existing Racket performance tools (the `time` form and the built-in Racket profiler [Barzilay 2017]). We then asked each participant to study four benchmarks—three realistic solver-aided tools and a simple calculator program—and identify (but not repair) the primary performance bottleneck. For each benchmark, each participant was randomly assigned to either the baseline group (which had access to any tool except SymPro) or the SymPro group (which had access to SymPro as well). To help control for learning effects, each participant saw the benchmarks in a random order, and had at most 20 minutes to analyze each benchmark.

Quantitative Results. Figure 8 shows the average time taken for users to identify the performance issue in each benchmark. SymPro improves the identification time for every benchmark, though due to the small sample size ($n = 4$ for each treatment), we do not claim statistical significance. There were 6 cases where a user in the baseline group failed to find the issue in a benchmark within the 20 minutes available; no users in the SymPro group ever reached this time limit.

Qualitative Observations. Given the limited size of our study, its main value was in the qualitative observations reported by the participants. Users with access to SymPro said it gave "insight into what Rosette is actually doing" which they lacked from other tools. One user said that SymPro was "extremely useful for investigating a performance issue," and that they could "see how I would optimize my own code using the [symbolic] profiler." Users generally reported they thought the symbolic profiler would be even more successful when run against their own code, because they "know what to ignore."

We found that users were most successful when using SymPro to conduct an initial investigation. SymPro's data analysis generally directed users to fruitful locations in the code to inspect more quickly than either manual exploration or analysis by existing performance tools. While we did not require users to identify potential repairs to the performance issues they found, they were more willing and able to do so voluntarily when using SymPro, suggesting a better understanding of the code.

6.4 Is SymPro effective at profiling different symbolic evaluation engines?

In addition to the Rosette profiler evaluated above, we also built a prototype symbolic profiler for the Jalangi dynamic analysis framework [Sen et al. 2013], as Section 4.4 describes. We applied the profiler to the three slowest publicly available benchmarks reported by Sen et al. [2015]. For each benchmark, we ran both the symbolic profiler and a traditional time-based profiler to identify hotspots, and compared the results. The symbolic profiler added only negligible overhead ($< 1\%$).

Red-Black Tree. The red-black tree benchmark implements a self-balancing binary search tree with integer keys. The symbolic version of the benchmark inserts five unknown, symbolic keys into the binary search tree. The time-based profiler identifies an internal key-comparison function as the only hotspot in the benchmark. But the symbolic profiler helps pinpoint why the key comparison is slow: it identifies the tree's `insert` procedure as being responsible for the creation of most symbolic state (due to branching), and reports that the key comparison creates very large terms on the symbolic heap. Guided by this profile, we modified the key comparison function to be branch-free, which improved the benchmark's performance by 2×. The profiler also suggests that path pruning with the SMT solver is ineffective on this benchmark: most paths are generated by the key-comparison function, but they are always feasible. Surprisingly, we found that the tree's rebalancing operations were not the sources of expensive symbolic operations.

Calculator Parser. The calculator parser benchmark implements a simple grammar for arithmetic expressions, and attempts to parse an expression from a symbolic input. The time-based profiler identifies the function `getsym`, which generates the next character of symbolic input, as the bottleneck. The symbolic profiler instead identifies `accept` and its callers, which interpret the output of `getsym` and form the core of the parser. In particular, the symbolic profiler identifies the grammar's "factor" production as being a bottleneck due to a large number of branches. Inspecting this function, we found most branches perform similar work, and so we refactored it to move that work outside of the branches. This small refactoring improved the benchmark's performance by 1.8×.

Binary Decision Diagram. The binary decision diagram (BDD) benchmark constructs a BDD with three unknown, symbolic operations (that can be either $\wedge$ or $\vee$), each of which operates on two unknown, symbolic operands. The time-based profiler can only identify the top-level driver function of this benchmark as a potential hotspot. The symbolic profiler is more effective, identifying an internal hash table and the BDD `put` operation as the sources of symbolic complexity. We replaced the hash table with a linked list, improving performance by 10%. With this repair, the profiler now identifies `get` as the bottleneck instead of `put` (as we would expect, since `get` must now search the list). While a linked list is clearly less efficient for concrete code, it is more amenable to verification, and so this transformation may be preferable for verifying *clients* of the BDD library. In general, we expect SymPro to be useful for developing *models* of libraries and frameworks, which are simplified implementations intended for verification purposes and used by automated verification tools [Cadar et al. 2008].

7 RELATED WORK

Optimizing Symbolic Evaluation. A high-performance symbolic evaluation engine must make good decisions about when to merge states from different paths. Query count estimation (QCE) is a heuristic for estimating the number of paths that will be created by merging at a given program point [Kuznetsov et al. 2012]. A QCE engine merges states only if the "hot" variables in each branch are the same, or are already symbolic. The "hot" variables are identified heuristically; a variable v is hot if many additional paths are likely to be generated by making v symbolic. In essence, QCE is a heuristic for predicting the shape of the symbolic evaluation graph. SymPro, in contrast, tracks the shape of the graph and lets the programmer use this information to guide symbolic evaluation.

In addition to improving the performance of symbolic evaluation at the engine level (through better strategies and encodings), prior work has also proposed making improvements at the program level. Wagner et al. [2013] advocate for a special compiler optimization mode tuned for emitting code amenable to symbolic execution, avoiding program transformations that exhibit poor behavior under symbolic evaluation. Cadar [2015] presents a collection of program transformations (both semantics-preserving and -altering) designed to enable scalable symbolic execution. SymPro is an ideal companion to these approaches: when automated optimizations fail (as Cadar shows is often the case), a profiler can help identify potential bottlenecks for manual repair.

Profiler-Aided Development. Recent research has focused on how profiling information should be integrated into development workflows. For example, the *optimization coach* [St-Amour et al. 2012] feature of Racket communicates successful and failed compiler optimizations to programmers, while profile-guided meta-programming [Bowman et al. 2015] integrates profiling data into the source-to-source transformations in Racket's macro system. One important property of a profiler is that its advice must be *actionable*: optimizing the functions it suggests as hot should improve execution time. The Coz causal profiler [Curtsinger and Berger 2015] achieves this by performing experiments at run time. To determine if a function f is hot, Coz simulates optimizing f by artificially slowing down every other function in the program. We took inspiration from all three of these techniques when designing SymPro.

Interactive Profiling. Ammons et al. [2004]'s Bottlenecks tool is an interactive interface for profile data. Profilers implement a common interface defined by Bottlenecks, which then layers a command-line user interface on top of the generated data. Through that interface, Bottlenecks suggests interesting profile points using navigation heuristics that skip over uninteresting data; for example, procedures with little exclusive time are likely less interesting than their callees, so navigation "zooms" over these points. Ammons et al. used Bottlenecks to find 14 performance issues in IBM's WebSphere Application Server, and improve its throughput by 23%. SymPro's user interface (Figure 3) and its common symbolic evaluator interface (Definition 3) both take influence from Bottlenecks.

8 CONCLUSION

This paper presented symbolic profiling, a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation. Symbolic profiling makes explicit the key resources—the symbolic heap and evaluation graph—that programmers must manage to create performant solver-aided applications. These resources form a new performance model of symbolic evaluation that is actionable, explainable, and general. Our case studies show that symbolic profiling produces actionable profiles. Guided by these profiles, we identified, diagnosed, and repaired performance bottlenecks in published, state-of-the-art solver-aided tools, obtaining orders-of-magnitude speedups. Our experiments show that symbolic profiles have high explanatory power, helping programmers understand what the symbolic evaluator is doing, and that our profiling approach generalizes to different symbolic evaluation engines. As programmers increasingly apply solver-aided automation to new domains, symbolic profiling can help them more quickly reach the scale they need to solve real-world problems.

ACKNOWLEDGMENTS

We thank Dan Grossman, Xi Wang, and the anonymous reviewers for their feedback on this work; Eunice Jun and Calvin Loncaric for help with user study design; and the participants in our user study. This work was supported in part by DARPA under agreement number FA8750-16-2-0032, by the National Science Foundation under grant CCF-1651225, by the joint Intel–NSF CAPA research center, by the Alfred P. Sloan Foundation, and by a Facebook PhD Fellowship.

REFERENCES

Amazon Web Services. 2018. Quivela. (2018). https://github.com/awslabs/quivela

Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. 2004. Finding and Removing Performance Bottlenecks in Large Systems. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. Oslo, Norway, 170–194.

Domagoj Babić and Alan J. Hu. 2008. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany, 211–220.

Eli Barzilay. 2017. Profile: Statistical Profiler. http://docs.racket-lang.org/profile/. (2017).

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Amsterdam, The Netherlands, 193–207.

Nicolas Boichat. 2015. Issue 502898: ext4: Filesystem corruption on panic. (June 2015). https://code.google.com/p/chromium/issues/detail?id=502898.

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 83–98.

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, 467–481.

Alan Borning. 2016. Wallingford: Toward a Constraint Reactive Programming Language. In *Proceedings of the Constrained and Reactive Objects Workshop (CROW)*. Málaga, Spain.

William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. 2015. Profile-guided Meta-programming. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, 229–239.

Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 239–254.

Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG)*. Hyannis, MA, USA.

Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 906–909.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 209–224.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

Kartik Chandra and Rastislav Bodik. 2018. Bonsai: Synthesis-Based Reasoning for Type Systems. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), 62:1–62:34.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017a. Cosette. (2017). http://github.com/uwdb/Cosette

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017b. Cosette: An Automated Prover for SQL. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems (CIDR)*. Chaminade, CA, USA.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Barcelona, Spain, 168–176.

Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2, 3 (1976), 215–222.

Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 89–98.

Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, USA, 184–197.

EPICS. 2017. Experimental Physics and Industrial Control System. (2017). http://www.aps.anl.gov/epics/

A. P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (1958), 3–6.

Malay Ganai and Aarti Gupta. 2008. Tunneling and slicing: Towards scalable BMC. In *Proceedings of the 45th Design Automation Conference (DAC)*. Anaheim, CA, USA, 137–142.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL, USA, 213–223.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Los Angeles, CA, USA, 269–290.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 89–98.

Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, Vol. 2. Toronto, ON, Canada, 23–41.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 297–310.

Racket 2017. The Racket Programming Language. (2017). https://racket-lang.org

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russian Federation, 488–498.

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 842–853.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, USA, 404–415.

Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *Proceedings of the 27th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Tuscon, AZ, USA, 163–178.

Emina Torlak. 2018. Rosette. (2018). http://github.com/emina/rosette

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. Indianapolis, IN, USA, 135–152.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541.

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-Based Search. In *Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, USA, 157–176.

Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. -Overify: Optimizing Programs for Fast Verification. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, NM, USA.

Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Amsterdam, The Netherlands, 765–780.

Max Willsey, Luis Ceze, and Karin Strauss. 2018. Puddle: An Operating System for Reliable, High-Level Programming of Digital Microfluidic Devices. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Wild and Crazy Ideas Session*. Williamsburg, VA, USA.

Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Long Beach, CA, USA, 351–363.