Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts

Tom Bergan Dan Grossman Luis Ceze

University of Washington, Department of Computer Science & Engineering

{tbergan,djg,luisceze}@cs.washington.edu

Abstract

We describe an algorithm to perform symbolic execution of a multithreaded program starting from an arbitrary program context. We argue that this can enable more efficient symbolic exploration of deep code paths in multithreaded programs by allowing the symbolic engine to jump directly to program contexts of interest.

The key challenge is modeling the initial context with reasonable precision—an overly approximate model leads to exploration of many infeasible paths during symbolic execution, while a very precise model would be so expensive to compute that computing it would defeat the purpose of jumping directly to the initial context in the first place. We propose a *context-specific dataflow analysis* that approximates the initial context cheaply, but precisely enough to avoid some common causes of infeasible-path explosion. This model is necessarily approximate—it may leave portions of the memory state unconstrained, leaving our symbolic execution unable to answer simple questions such as "which thread holds lock A?". For such cases, we describe a novel algorithm for evaluating *symbolic synchronization* during symbolic execution. Our symbolic execution semantics are sound and complete up to the limits of the underlying SMT solver. We describe initial experiments on an implementation in Cloud9.

Categories and Subject Descriptors D.1.3 [*Programming Languages*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic Execution

Keywords static analysis; symbolic execution; multithreading

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2585-1/14/10.
http://dx.doi.org/10.1145/2660193.2660200

1. Introduction

Symbolic execution is a program analysis technique for systematically exploring all feasible execution paths. The idea is to execute programs with symbolic rather than concrete inputs and use an SMT (SAT Modulo Theory) solver to prune infeasible paths. On branches with more than one feasible resolution, the symbolic state is forked and all feasible resolutions are explored. The key advantage of this approach is precision—unlike other techniques, such as abstract interpretation, symbolic execution is generally free of false positives because its semantics are fully precise up to the limits of the underlying SMT solver, and recent advances in SMT solving have made symbolic execution faster and more practical [19, 33].

Symbolic execution has been used to find bugs and generate high-coverage test cases with great success [10, 13, 22, 23, 39]. Unfortunately, building scalable symbolic execution engines is difficult because of path explosion: the number of feasible execution paths is generally exponential in the length of an execution. Path explosion is even worse when symbolic execution is applied to multithreaded programs [9, 27], which suffer from an explosion of possible thread interleavings in addition to the explosion of single-threaded paths. Prior work has dealt with path explosion using summarization [20, 24, 36], path merging [25, 28], search heuristics [8, 10, 31, 34], and partial order reductions [18].

Our goal is scalable symbolic execution of multithreaded programs written in the C language and its derivatives. Our approach is to limit path explosion by symbolically executing relatively small fragments of a program in isolation—this reduces path length, which in turn reduces the potential for path explosion. Rather than exploring ways that program fragments might be selected, this paper focuses on a more basic question: how do we symbolically execute a fragment of a multithreaded program in isolation, *soundly* and *efficiently*? Prior work has largely assumed that symbolic execution will begin at one of a few natural starting points, such as program entry (for whole-program testing) or a function call (for single-threaded unit testing). We do not make such an assumption—we allow program fragments to begin anywhere—so our main challenge is to perform sym-

```
1   global int X,Y
2   global struct Node { Lock lock, int data } nodes[]
3
4   Thread 1                    Thread 2
5     void RunA() {               void RunB() {
6       i = ...                     k = ...
7       Foo(&nodes[i])              Bar(&nodes[k])
8     }                           }
9     void Foo(Node *a) {         void Bar(Node *b) {
10      for (x in 1..X) {           lock(b->lock)
11  ⇒     lock(a->lock)       ⇒    for (y in 1..Y)
12        ...                         ...
```

Figure 1. A simple multithreaded program that illustrates the challenges of beginning symbolic execution at an arbitrary program context. Especially notable are challenges that arise from explicit synchronization and from C-like pointers.

bolic execution of multithreaded programs from *arbitrary* program contexts.

1.1 Problem Statement and Solution Overview

Specifically, we address the following problem: given an *initial program context*, which we define to be a set of threads and their program counters, how do we efficiently perform symbolic execution starting from that context while soundly accounting for all possible concrete initial states? We solve this problem in two parts. First, we use a *context-specific dataflow analysis* to construct an over-approximation of the initial state for the given program context. Second, we integrate that analysis with a novel *symbolic execution semantics* that can execute forward from an abstract initial state, even when precise information about pointers and synchronization is not available.

Constructing an Initial State. The most precise strategy is to symbolically execute all paths from program entry to the initial context, and then use path merging [25] to construct an initial state. This is not scalable—it suffers from exactly the sort of path explosion problems we are trying to avoid. Instead, we must *approximate* the initial state. The least precise approximation is to leave the initial state completely unconstrained, for example by assigning a fresh symbolic variable to every memory location. This is too conservative—it covers many memory states that never occur during any actual execution—and as a result, symbolic execution would investigate many infeasible paths.

Our solution represents a middle ground between the above two extremes: we use a *context-specific* dataflow analysis to construct a sound *over-approximation* of the initial state. We use an over-approximation to ensure that all feasible concrete initial states are included. Our analysis includes constraints on the initial memory state as well as constraints on synchronization, such as *locksets*, that together help symbolic execution avoid infeasible paths.

To illustrate, suppose we are asked to begin symbolic execution from the program context marked by arrows in Figure 1. This context includes two threads, each of which is about to execute line 11. Can lines 11 and 12 of `Foo` execute concurrently with lines 11 and 12 of `Bar`? To answer this we must first answer a different question: does thread t_2 hold any locks at the beginning of the program context (*i.e.*, at line 11)? Here we examine the locksets embedded in our initial state and learn that t_2 holds lock `b->lock`. Next, we ask another question: does `a==b`? Suppose our dataflow analysis determines that `i==k` at line 6, and that `Foo` and `Bar` are called from `RunA` and `RunB` only. In this case, we know that `a==b`, which means that line 11 of `Foo` cannot execute concurrently with line 11 of `Bar`.

Symbolic Execution Semantics. The input to symbolic execution is an abstract initial state constructed by our dataflow analysis. The output is a set of pairs (`path`, `C`), where `path` is a path of execution and `C` is a path constraint: if `C` is satisfied on an initial state, it *must be possible* for execution to follow the corresponding `path` from that initial state, as long as context switches are made in the appropriate places. To support multithreaded programs, we make each `path` a serialized (sequentially consistent) trace of a multithreaded execution.

Our symbolic execution begins from an initial state that is necessarily approximate, leaving us unable to precisely answer simple questions such as "which object does pointer X refer to?" in all cases. For example, suppose our dataflow analysis cannot prove that `i==k` at line 6. In this case, we must investigate two paths during symbolic execution: one in which `a==b`, and another in which `a!=b`. For this reason, the set of paths explored by our symbolic execution may be a *superset* of the set of paths that are actually feasible.

We attempt to avoid infeasible paths using two approaches: first, a novel algorithm for evaluating *symbolic synchronization* (§4.2-4.3), and second, an algorithm for evaluating *symbolic pointers* that incorporates ideas from prior work [12, 15] (§3.2)).

Dealing with Data Races. A key question for any analysis of multithreaded programs is how to deal with data races. Our symbolic semantics treat data races as runtime errors that halt execution. Our dataflow analysis assumes race freedom and ignores data races entirely—we believe this is the only practical approach in the dataflow setting, particularly for C programs, where a conservative analysis is technically required to assume the possibility of "undefined behavior" when it cannot prove that the program is race free [6] (and proving race freedom is incredibly difficult).

Soundness and Completeness. Our symbolic semantics are sound and complete up to the limits of the underlying SMT solver. By *sound*, we mean that if our symbolic execution outputs a pair (`path`, `C`), then, from every concrete initial state that satisfies constraint `C`, concrete execution *must* follow `path` as long as context switches are made just as in `path`. By *complete*, we mean that symbolic execution outputs a set of pairs (`path`, `C`) sufficient to cover *all* possible concrete initial states that may arise during any valid execution of the program. However, our analysis is incomplete in practice: first, SMT solvers are incomplete in practice, and second, the set of feasible paths can be too large to enumerate in practice.

1.2 Applications

Our techniques have a variety of promising applications:

Focused Testing of Program Fragments. We can test an important parallel loop in the context of a larger program. Classic symbolic execution techniques require executing deep code paths from program entry to reach the loop in the first place, where these deep paths may include complex initialization code or prior parallel phases. Our techniques enable testing the loop directly, using a fast and scalable dataflow analysis to summarize the initial deep paths.

Testing Libraries. We would ideally test a concurrent library over all inputs and calling contexts, but as this is often infeasible, we instead might want to *prioritize* the specific contexts a library is called from by a specific program. One such prioritization strategy is to enumerate all pairs of calls into the library that may run concurrently, then treat each pair as a program context that can be symbolically executed using our techniques. Then do the same for every triple of concurrent calls, every quadruple, and so on.

Piecewise Program Testing. Rather than testing a whole program with one long symbolic execution, we can break the program into adjacent fragments and test each fragment in isolation. Such a piecewise testing scheme might enumerate fragments *dynamically* by queuing the next fragments found to be reachable from the current fragment. Fragments might end at loop backedges, for loops with input-dependent iteration counts, producing a set of fragments that are each short and largely acyclic. The key advantage is that we can explore fragments in parallel, as they are enumerated, enabling us to more quickly reach a variety of deep paths in the program's execution. The trade-off is a potential loss of precision, as our dataflow analysis may make conservative assumptions when constructing each fragment's initial abstract state.

Execution Reconstruction. We can record an execution with periodic lightweight checkpoints that include call stacks and little else. Then, on a crash, we can symbolically execute from a checkpoint onwards to reconstruct the bug. Variants of this approach include *bbr* [12] and *RES* [42]. However, *bbr* does not work for multithreaded programs, and both systems have less powerful support for pointers than does our semantics.

Input-Covering Schedules. We have used our symbolic execution techniques as part of an algorithm for finding *input-covering schedules*. This algorithm, which we describe in prior work [2], partitions execution into adjacent fragments and uses symbolic execution to analyze each fragment in isolation. In §7, we evaluate our symbolic execution techniques in the context of this algorithm.

1.3 Contributions and Outline

We propose a framework for solving the basic problem—symbolic execution from arbitrary multithreaded contexts. In particular, we make two primary contributions:

- We propose a way to integrate dataflow analysis with symbolic execution (§2.3, §3.3, §4.4). In particular, our dataflow analysis computes a cheap summary that is used as the starting point for symbolic execution. Deciding *which* dataflow analysis to use is a hard problem, as hundreds have been proposed and the best choice is likely application-dependent. We combine *reaching definitions*, which summarize the state of memory, with *locksets* and *barrier matching*, which summarize the state of synchronization.
- We propose a novel algorithm for evaluating symbolic synchronization (§4.2–§4.3). While prior work has largely focused on path explosion due to branches, we focus on path explosion due to synchronization, which is often symbolic in our context.

Outline. We start with a simple, single-threaded imperative language that has no pointers (§2). We then add pointers (§3) and threads (§4). At each step, we explain how we overcome the challenges introduced by each additional language feature. We then state soundness and completeness theorems (§5), discuss our implementation (§6) and empirical evaluation (§7), and end with related work (§8).

2. A Simple Imperative Language

Figure 2 gives the syntax of *Simp*, a simple imperative language that we use as a starting point. A program in this language contains a set of functions, including a distinguished `main` function for program entry. The language includes function calls, conditional branching, mutable local variables, and a set of standard arithmetic and boolean expressions (only partially shown). We separate side-effect-free expressions from statements. This simple language does *not* include pointers, dynamic memory allocation, or threads—those language features will be added in §3 and §4.

The concrete semantics follow the standard pattern for imperative, lexically-scoped, call-by-value languages. We omit the detailed rules for brevity. We use r to refer to local variables (or "registers"). The metavariables x and y do not appear in the actual concrete language—instead, x and y are used to name *symbolic constants* that represent unknown values during symbolic execution, as described below.

Challenges. Although this language is simple, it reveals two ways in which symbolic execution from arbitrary contexts can be imprecise. Specifically, we use this language to demonstrate imprecision due to unknown calling contexts (§2.2) and unknown values of local variables (§2.3). We also use this language to present basic frameworks that we will reuse in the rest of this paper.

$$
\begin{array}{ll}
r \in \mathit{Var} & \textit{(local variables)} \\
x, y \in \mathit{SymbolicConst} & \textit{(symbolic constants)} \\
f \in \mathit{FName} & \textit{(function names)} \\
i \in \mathbb{Z} & \textit{(integers)}
\end{array}
$$

$$
\begin{array}{l}
v \in \mathit{Value} ::= f \mid i \\
e \in \mathit{Expr} ::= v \mid r \mid x \mid e \wedge e \mid e \vee e \mid e < e \mid \ldots \\
\gamma \in \mathit{StmtLabel} \\
s \in \mathit{Stmt} ::= r \leftarrow e(e^{*}) \\
\qquad\qquad \mid \texttt{br}\ e,\ \gamma_t,\ \gamma_f \\
\qquad\qquad \mid \texttt{return}\ e \\
\mathit{Func} ::= \texttt{func}\ f(r^{*})\{\ (\gamma : s;)^{*}\ \}
\end{array}
$$

Figure 2. Syntax of *Simp*. Asterisks (*) denote repetition.

2.1 Symbolic Semantics Overview

We now describe an algorithm to perform symbolic execution of *Simp* programs. Our algorithm operates over symbolic states that contain the following domains (also illustrated in Figure 3):

- $\overline{\mathcal{Y}}$, which is a stack of local variable bindings. A new stack frame is pushed by each function call and popped by the matching return. Variables are bound to either function arguments (for formal parameters) or the result of a statement (as in $r \leftarrow f()$).
- *CallCtx*, which names the current calling context, where the youngest stack label is the thread's current program counter and older labels are return addresses.
- *path*, which records an execution trace.
- *C*, an expression that records the current path constraint.

Constructing an Initial State. Recall from §1.1 that our job is to perform symbolic execution from an arbitrary program context that is specified by a set of program counters, one for each thread. As *Simp* is single-threaded, the initial program context for *Simp* programs contains just one program counter, γ_0.

Given γ_0, where γ_0 is a statement in function f_0, we must construct an initial symbolic state, S_{init}, from which we can begin symbolic execution. A simple approach is: $path_{init} = empty$; $C_{init} = \text{true}$; $CallCtx_{init} = \{\gamma_0\}$; and $\overline{\mathcal{Y}}_{init} = \{\{r_i \to x_i \mid \forall r_i \in f_0\}\}$. $\overline{\mathcal{Y}}_{init}$ contains one stack frame that maps each $r_i \in f_0$ to a distinct symbolic constant x_i. We describe a more precise approach in §2.3.

Correspondence of Concrete and Symbolic States. Note that we use *symbolic constants*, such as x_i, above, to represent unknown parts of a symbolic state. This allows each symbolic state to represent a *set* of concrete states. Specifically, the set of concrete states represented by S_{init} can be found by enumerating the total set of assignments of symbolic constants x_i to values v_i—each such assignment corresponds to a concrete state in which $x_i = v_i$.

$\overline{\mathcal{Y}}$	:	Stack of $(\mathit{Var} \to \mathit{Expr})$	*(local variables)*
CallCtx	:	Stack of *StmtLabel*	*(calling context)*
path	:	List of *StmtLabel*	*(execution trace)*
C	:	*Expr*	*(path constraint)*

Figure 3. Symbolic state for *Simp*.

Symbolic Execution. At a high level, symbolic execution is straightforward. We begin from the initial state, S_{init}. We execute one statement at a time using $step$, which is defined below. At branches, we use an SMT solver to determine which branch edges are feasible and we fork as necessary. We repeatedly execute $step$ on non-terminated states until all states have terminated or until a user-defined resource budget has been exceeded. We define $step$ as follows, and we also make use of an auxiliary function $eval$ to evaluate side-effect-free expressions:

- $step : (\mathit{State} \times \mathit{Stmt}) \to$ Set of *State*
  Evaluates a single statement under an initial state and produces a set of states, as we may *fork* execution at control flow statements to separately evaluate each feasible branch. The type of each *State* is given by Figure 3.
- $eval : ((\mathit{Var} \to \mathit{Expr}) \times \mathit{Expr}) \to \mathit{Expr}$
  Given $eval(\mathcal{Y}, e)$, we evaluate expression e under binding $\mathcal{Y}$, where $\mathcal{Y}$ represents a single stack frame. We expect that $\mathcal{Y}$ has a binding for every local variable referenced by e. Note that $eval$ returns an *Expr* rather than a *Value*, as we cannot completely reduce expressions that contain symbolic constants.

Our algorithm's final result is a set of *State*s from which we can extract $(path, C)$ pairs that represent our final output. For each such pair, C is an expression that constrains the initial symbolic state, S_{init}, such that when C is satisfied, program execution *must* follow the corresponding *path*.

SMT Solver Interface. Our symbolic semantics relies on an SMT solver that we query using the following interface. The function $isSat(C, e)$, shown below, determines if boolean expression e is satisfiable under the constraints given by expression C, where C is a conjunction of assumptions. In addition to *isSat*, we use *mayBeTrue* and *mustBeTrue* as syntactic sugar, as defined below.

$$
\begin{array}{l}
isSat(C, e) = \text{true iff } e \text{ is satisfiable under } C \\
mayBeTrue(C, e) = isSat(C, e) \\
mustBeTrue(C, e) = \neg mayBeTrue(C, \neg e)
\end{array}
$$

If a query $isSat(C, e)$ cannot be solved, then our symbolic execution becomes *incomplete*. In this case, we concretize enough subexpressions of e so the query becomes solvable and we can make forward progress (similarly to Pasareanu *et al.* [35]).

2.2 Dealing with an Underspecified *CallCtx*

Recall that the initial program context is simply a single program counter, γ_0. If γ_0 is not a statement in the `main` function, then the initial state S_{init} does *not* have a complete call stack. How do we reconstruct a complete call stack?

We could start with a single stack frame and then lazily expand older frames, forking as necessary to explore all paths through the static call graph. However, we consider this overkill for our anticipated applications (recall §1.2), and instead opt to exit the program when the initial stack frame returns. Our rationale is that, for each application listed in §1.2, either the program fragment of interest will be lexically scoped, in which case we never return from the initial stack frames anyway, or complete call stacks will be provided, which we can use directly (*e.g.*, we expect that complete call stacks will be available during execution reconstruction, as in *bbr* [12]).

2.3 Initializing Local Variables with Reaching Definitions

The simple approach for constructing S_{init}, as described above, is imprecise. Specifically, the simple approach assigns each local variable a unique symbolic constant, x_i, effectively assuming that each local variable can start with *any* initial value. This is often not the case. For example, consider thread t_1 in Figure 1. In this example, assuming that `RunA` is the only caller of `Foo`, the value of local variable `a` is known precisely. Even when the initial value of a variable cannot be determined precisely, we can often define its initial value as a symbolic function over other variables.

Our approach is to initialize $\overline{\mathcal{Y}}_{init}$ using an interprocedural dataflow analysis that computes *reaching definitions* for all local variables. We use a standard iterative dataflow analysis framework with function summaries for scalability, and we make the framework *context-specific* as follows: First, we combine a static call graph with each function's control-flow graph to produce an interprocedural control-flow graph, *CFG*. Then, we run a forwards dataflow analysis over *CFG* that starts from `main` and summarizes all interprocedural paths between program entry and the initial program counter, γ_0.

Our dataflow analysis computes assignments that *must-reach* the initial program context. Specifically, we compute a set of pairs $R_{local} = \{(r_i, e_i)\}$, where each r_i is a local variable in $\overline{\mathcal{Y}}_{init}$ such that the assignment $r_i \leftarrow e_i$ must-reach the initial program context. That is, r_i's value at the initial program context *must* match expression e_i. We compute R_{local} using standard flow functions for reaching definitions, then assign each e_i to r_i in $\overline{\mathcal{Y}}_{init}$. Some variables may not have a must-reach assignment—these variables, r_k, do not appear in R_{local}, and they are assigned a unique symbolic constant x_k in $\overline{\mathcal{Y}}_{init}$, as before.

As we compute R_{local}, each assignment $r \leftarrow e$ generates a must-reach definition $(r, eval(R_{local}, e))$ on its outgoing dataflow edge. Note that we use $eval$ to reduce expressions. Thus, given $r_1 \leftarrow r_2{+}5$, where $(r_2, x) \in R_{local}$, we generate the definition $(r_1, x{+}5)$ to express that r_1 and r_2 are functions of the same value.

Must-Reach vs. May-Reach. Must-reach definitions provide a sound *over-approximation* of $\overline{\mathcal{Y}}_{init}$, as any variable not included in the must-reach set may have *any* initial value. More precision could be achieved through *may-reach* definitions; however, this would result in a symbolic state with many large disjunctions that are expensive to solve in current SMT solvers [25, 28].

3. Adding Pointers

Figure 4 shows the syntax of *SimpHeaps*, which adds pointers and dynamic memory allocation to *Simp*. As a convention, we use p to range over expressions that should evaluate to pointers.

Memory Interface. We represent pointers as pairs $\texttt{ptr}(l, i)$, where l is the base address of a heap object and i is a non-negative integer offset into that object. Pointers may also be `null`. Pointer arithmetic is supported with the $\texttt{ptradd}(p, e)$ expression, which is evaluated as follows in the concrete language:

$$\frac{eval(\mathcal{Y}, p) = \texttt{ptr}(l, i) \qquad eval(\mathcal{Y}, e) = i'}{eval(\mathcal{Y}, \texttt{ptradd}(p, e)) = \texttt{ptr}(l, i + i')}$$

The heap is a mapping from locations to objects, and each object includes a sequence of fields. Our notion of a field encompasses the common notions of array elements and structure fields. To simplify the semantics, we assume that each field has a uniform size that is big enough to store any value. Following that assumption, we define i to be the offset of the $(i{+}1)$th field (making 0 the offset of the first field), and we define the size of an object to be its number of fields. Our implementation (§6) relaxes this assumption to support variable-sized fields at byte-granular offsets. Heap objects are allocated with `malloc`, which returns $\texttt{ptr}(l, 0)$ with a fresh location l, and they are deallocated with `free`.

Memory Errors. Out-of-bounds memory accesses, uninitialized memory reads, and other memory errors have undefined behavior in C [26]. We treat these as runtime errors in our semantics to simplify the notions of *soundness* and *completeness* of symbolic execution. The details of dynamic detectors for these errors are orthogonal to this paper and are not discussed in detail.

Challenges. In the concrete language, `load` and `store` statements always operate on values of the form $\texttt{ptr}(l, i)$. The symbolic semantics must consider three additional kinds of pointer expressions: $\texttt{ptr}(l, e)$, in which the offset e is symbolic; and x and $\texttt{ptradd}(x, e)$, in which the heap location is symbolic as well.

3.1 Prior Work

A natural approach is to represent the symbolic heap as a single mapping from addresses to values, then use the theory of arrays [19] to reason about reads and writes over this

$$
\begin{array}{rll}
l \in \textit{Loc} & \textit{(heap locations)} & \\
v \in \textit{Value} & ::= \; ... \mid \texttt{null} \mid \texttt{ptr}(l, i) & \\
e, p \in \textit{Expr} & ::= \; ... \mid \texttt{ptr}(l, e) \mid \texttt{ptradd}(p, e) & \\
s \in \textit{Stmt} & ::= \; ... \mid r \leftarrow \texttt{load}\ p \mid \texttt{store}\ p,\ e & \\
 & \mid r \leftarrow \texttt{malloc}(e) \mid \texttt{free}(p) &
\end{array}
$$

Figure 4. Syntax additions for *SimpHeaps*.

$$
\begin{array}{l}
\mathcal{H} \; : \; \textit{Loc} \to \{\text{fields} : (\textit{Expr} \to \textit{Expr})\} \\
\mathcal{A} \; : \; \text{List of } \{\text{x} : \textit{SymbolicConst},\ \text{primary} : \textit{Loc},\ \text{n} : \textit{PtrNode}\}
\end{array}
$$

Figure 5. Symbolic state additions for *SimpHeaps*, including a *heap* ($\mathcal{H}$) and a list of *aliasable objects* ($\mathcal{A}$).

mapping. This approach is common in work on program verifiers [7, 11].

In contrast, some systems use a *separate* array for each memory object. The authors of KLEE observed that symbolic executors often send many small queries to the SMT solver, *e.g.*, to resolve branches, and these small queries can be resolved efficiently using caching [10]. This contrasts with program verifiers, which summarize each program with a single large expression that is sent to an SMT solver just once. KLEE's key insight is that, by assigning each memory object its own array, symbolic memory expressions will naturally avoid mentioning irrelevant parts of the heap, making caching more effective.

Unfortunately, KLEE cannot efficiently resolve pointers with symbolic heap locations (it instead focuses on cases where only the offset is symbolic, as in $\texttt{ptr}(l, x)$). We desire a semantics that assigns each object its own symbolic array, to maintain the caching effectiveness of KLEE, but in a way that enables efficient resolution of symbolic locations. Our approach is a fusion of approaches from *bbr* [12] and Dillig *et al.* [15]: both approaches use conditional aliasing expressions to encode multiple concrete heap graphs into a single symbolic state. However, *bbr* cannot reason about symbolic offsets (like $\texttt{ptradd}(x, y)$) or memory allocations of a symbolic size, and the approach from Dillig *et al.* is not path-sensitive (so it must deal with approximation at control-flow merge points) and it does not exploit the full power of the theory of arrays. Our fused approach overcomes these limitations, as summarized in the following subsection.

3.2 Symbolic Semantics

We now extend our symbolic execution algorithm for *SimpHeaps*. As shown in Figure 5, we add two fields to the symbolic state: a heap, $\mathcal{H}$, which maps concrete locations to dynamically allocated heap objects, and a list $\mathcal{A}$, which tracks aliasing information that is used to resolve symbolic pointers. Key rules for the semantics described in this section are given in Figure 6. The $\overset{\textit{mem}}{\Longrightarrow}$ relation is used by *step* to evaluate memory statements. (Note that memory operations never fork execution in the absence of memory errors, and we elide those error-checking details from this paper.)

Accessing Concrete Locations. We first consider accessing pointers of the form $\texttt{ptr}(l, e)$. In this case, l uniquely names the heap object being accessed, so we simply construct an expression in the theory of arrays [19] to load from or store to offset e of that object's `fields` array.

Accessing Symbolic Locations. Now we consider accessing pointers of the form x and $\texttt{ptradd}(x, e)$. This case is more challenging since the pointer x may refer to an unknown object. Following *bbr* [12], we assign each symbolic pointer x a unique *primary object* in the heap, then use aliasing constraints to allow multiple pointers to refer to the same object. This effectively encodes multiple concrete memory graphs into a single symbolic heap. We allocate the primary object for x *lazily*, the first time x is accessed. In this way, we lazily expand the symbolic heap and are able to efficiently encode heaps with unboundedly many objects.

Stores to x update x's primary object, l_x, and also conditionally update all other objects that x may-alias. For example, suppose pointers x and y may point to the same object. To write value v to pointer x, we first update l_x by writing v to address $\texttt{ptr}(l_x, x_{\textit{off}})$, and we then update l_y by writing the expression $(x = \texttt{ptr}(l_y, x_{\textit{off}}))\ ?\ v : e_{\textit{old}}$ to address $\texttt{ptr}(l_y, e)$, where $e_{\textit{old}}$ is the previous value in l_y (this makes the update *conditional*) and $x_{\textit{off}}$ is a symbolic offset that will be described shortly. Loads of x access $\texttt{ptr}(l_x, x_{\textit{off}})$ directly—since stores update all aliases, it is unnecessary for loads to access aliases as well.

Figure 6 shows the detailed semantics for accessing symbolic pointers of the form $\texttt{ptradd}(x, e_{\textit{off}})$ (we treat x as $\texttt{ptradd}(x, 0)$).

Restricting Aliasing with a Points-To Analysis. Recall that symbolic constants like x represent values that originate in our initial program context. That is, if x is a valid pointer, then x *must* point to some object that was allocated *before* our initial program context. In the worst case, the set of possible aliases includes all primary objects that have been previously allocated for other symbolic pointers. This list of objects is recorded in $\mathcal{A}$ (Figure 5) and kept up-to-date by *addPrimary*.

In practice, we can narrow the set of aliases using a static points-to analysis, as in [15]. On the first access to x, we add the record $\{x, l_x, n_x\}$ to $\mathcal{A}$, where n_x is the representative node for x in the static points-to graph. The set of objects that x *may-alias* is found by enumerating all $\{y, l_y, n_y\} \in \mathcal{A}$ for which n_y and n_x may point-to the same object according to the static points-to graph—this search is performed by *lookupAliases*. Note that, in practice, the search for aliases can be implemented efficiently by exploiting the structure of the underlying points-to graph.

We use a *field-sensitive* points-to analysis so we can additionally constrain the offset being accessed. For each symbolic pointer x, we query the points-to analysis to compute

Symbolic heap interface, including conditional *put*:

$$\frac{(l, \{\text{fields}\}) \in \mathcal{H} \qquad \textit{read}(\text{fields}, e_{\mathit{off}}) = e}{\textit{heapGet}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}})) = e} \qquad \frac{\begin{array}{c}(l, \{\text{fields}\}) \in \mathcal{H} \qquad \textit{read}(\text{fields}, e_{\mathit{off}}) = e_{\mathit{old}} \qquad e_{\mathit{val}} = e_{\mathit{cond}}\ ?\ e : e_{\mathit{old}} \\ \textit{write}(\text{fields}, e_{\mathit{off}}, e_{\mathit{val}}) = \text{fields}' \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\text{fields}'\}]\end{array}}{\textit{heapPut}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}}), e_{\mathit{cond}}, e) = \mathcal{H}'}$$

$$\boxed{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \textit{Stmt} \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}'; C'; \mathcal{A}'}$$

Load/store of a concrete location:

$$\frac{\begin{array}{c}\textit{eval}(\mathcal{Y}, p) = \mathtt{ptr}(l, e_{\mathit{off}}) \\ \textit{heapGet}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}})) = e\end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{load}\ p \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}; \mathcal{Y}[r \mapsto e]; C; \mathcal{A}} \qquad \frac{\begin{array}{c}\textit{eval}(\mathcal{Y}, p) = \mathtt{ptr}(l, e_{\mathit{off}}) \qquad \textit{eval}(\mathcal{Y}, e) = e' \\ \textit{heapPut}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}}), \text{true}, e') = \mathcal{H}'\end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{store}\ p,\ e \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}; C; \mathcal{A}}$$

Load/store of a symbolic location:

$$\frac{\begin{array}{c}\textit{eval}(\mathcal{Y}, p) = \mathtt{ptradd}(x, e_{\mathit{off}}) \\ \textit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \mathtt{ptr}(l_x, x_{\mathit{off}})) \\ \textit{heapGet}(\mathcal{H}', \mathtt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}})) = e\end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{load}\ p \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto e]; C'; \mathcal{A}'}$$

$$\frac{\begin{array}{c}\textit{eval}(\mathcal{Y}, p) = \mathtt{ptradd}(x, e_{\mathit{off}}) \quad \textit{eval}(\mathcal{Y}, e) = e' \\ \textit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \mathtt{ptr}(l_x, x_{\mathit{off}})) \\ \textit{lookupAliases}(\mathcal{A}', x) = \{l_1 ... l_n\} \\ \textit{heapPut}(\mathcal{H}', \ \mathtt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}}), \text{true}, \ e') = \mathcal{H}''_0 \\ \textit{heapPut}(\mathcal{H}''_0, \ \mathtt{ptr}(l_1, x_{\mathit{off}} + e_{\mathit{off}}), (x = \mathtt{ptr}(l_1, x_{\mathit{off}})), e') = \mathcal{H}''_1 \\ \cdots \\ \textit{heapPut}(\mathcal{H}''_{n-1}, \mathtt{ptr}(l_n, x_{\mathit{off}} + e_{\mathit{off}}), (x = \mathtt{ptr}(l_n, x_{\mathit{off}})), e') = \mathcal{H}''_n\end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{store}\ p,\ e \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}''_n; \mathcal{Y}; C'; \mathcal{A}'}$$

Allocate and free:

$$\frac{l = \textit{fresh loc} \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\lambda i.\mathtt{undef}\}]}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{malloc}(e_{\mathit{size}}) \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto \mathtt{ptr}(l, 0)]; C; \mathcal{A}} \qquad \frac{\textit{true}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{free}(p) \overset{\textit{mem}}{\Longrightarrow} \mathcal{H}; \mathcal{Y}; C; \mathcal{A}}$$

Figure 6. Representative rules from the symbolic heap semantics. In these rules, $\mathcal{Y}$ refers to the current stack frame (namely, the youngest stack frame in $\overline{\mathcal{Y}}$), and $\textit{read}(A, e_{\mathit{off}})$ and $\textit{write}(A, e_{\mathit{off}}, e_{\mathit{val}})$ are standard constructors from the theory of arrays (*e.g.*, see [19]).

a range of possible offsets for x, and then construct a fresh symbolic constant x_{off} that is constrained to that range. (This is the same x_{off} used above in the discussion of loads and stores.) For example, if x is known to point at a specific field, then x_{off} is fixed to that field. If a range of offsets cannot be soundly determined, x_{off} is left unconstrained.

Heap Invariants. On the first access of symbolic pointer x, *addPrimary* allocates a primary object at l_x, appends the record $\{x, l_x, n_x\}$ to $\mathcal{A}$, and enforces a *heap invariant* that we describe now.

Suppose the first access of x is a load, and suppose that x may-alias some other symbolic pointer y. For soundness, we *must* ensure that every load of x satisfies the following invariant: $x = y \implies \mathtt{load}(x) = \mathtt{load}(y)$. Making matters more complicated is the fact that we may have performed stores on y before our first access of x—we must ensure that these stores are visible through x as well. Our approach is to define the initial fields of l_x as follows:

$$\begin{aligned}&\textit{Initial}\ \mathtt{fields}\ \textit{of}\ l_x \\ &\quad \equiv\ (x = \mathtt{ptr}(l_y, x_{\mathit{off}}))\ ?\ \mathtt{fields}_y : \textit{fresh}\end{aligned} \tag{1}$$

where $\mathtt{fields}_y$ is the current fields array of object l_y, which is the primary object for y, and where *fresh* is a symbolic array that maps each field $\textit{fresh}(i)$ to a fresh symbolic constant—this represents the unknown initial values of l_x in the case that x and y do not alias. In general x may have more than one alias, in which case we initialize the `fields` of l_x similarly to the above, but we use a chain of conditionals that compares x with all possible aliases.

Memory Allocation. Semantics for $\mathtt{malloc}(e_{\mathit{size}})$ are shown in Figure 6. Since each object has its own symbolic `fields` array, we naturally support allocations of unbounded symbolic size.

Memory Error Checkers. Since this paper elides memory error-checking details, we treat $\mathtt{free}(p)$ as a no-op in Figure 6.

Briefly, to detect memory errors, we might add *size* and *isLive* attributes to each object in $\mathcal{H}$. On $\mathtt{malloc}(e)$, we would set $\textit{size} = e$ and $\textit{isLive} = \text{true}$. On $\mathtt{free}(p)$, we would *conditionally* free all objects that p may-alias by conditionally setting $\textit{isLive} = \text{false}$ in all aliases, much in the same way that $\texttt{store}(p, e)$ conditionally writes e to all aliases of p. Error checkers such as out-of-bounds and use-after-free would then ensure that, for each access at $\texttt{ptr}(l, e_{\mathit{off}})$, $0 \le e_{\mathit{off}} < \mathcal{H}(l).\mathit{size}$ and $\mathcal{H}(l).\mathit{isLive} = \text{true}$.

Compound Symbolic Pointer Expressions. Figure 6 shows rules for load and store statements where the pointer p evaluates to an expression of the form $\texttt{ptr}(l, e)$, x, or $\texttt{ptradd}(x, e)$, but the result of $eval(\mathcal{Y}, p)$ can also have the form $\mathit{read}(\mathit{fields}, e_{\mathit{off}})$. This form appears when a pointer is read from the heap, since all heap accesses use the theory of arrays.

The difficulty is that there may be multiple possible values at e_{off}. For example, if *fields* is $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, x')$, then we cannot evaluate this address without first resolving the symbolic pointers x and x'. Further, the values written by *write* can contain conditional expressions due to the conditional store performed by *heapPut*. So, in general, the fields array might include a chain of calls as in the following: $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, e'' \mathbin{?} x' : x'')$.

Our approach is to walk the call chain of *write*s to build *guarded expressions* that summarize the possible values at offset e_{off}. If the value stored by a *write* is a conditional expression, we also walk that conditional expression tree while computing the guarded expressions. This gives each guarded expression the form $e_{\mathit{grd}} \to p$, where each p has the form x, $\texttt{ptradd}(x, e)$, or $\texttt{ptr}(l, e)$. In the above example, we build guarded expressions $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge e'') \to x'$, and $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge \neg e'') \to x''$, and $(e_{\mathit{off}} \ne e'_{\mathit{off}} \wedge e_{\mathit{off}} = 1) \to x$, and so on down the call chain.

We then execute the memory operation on this set of guarded expressions. For stores, we evaluate each guarded expression independently: given $e_{\mathit{grd}} \to p$, we evaluate p using the rules in Figure 6, but we include e_{grd} in the condition passed to *heapPut*. For loads, we use the rules in Figure 6 to map each pair $e_{\mathit{grd}} \to p$ to a pair $e_{\mathit{grd}} \to e$, where e is the value loaded from pointer p. We then collect each $e_{\mathit{grd}} \to e$ into a conditional expression tree that represents the final value of the load. Continuing the above example, if the values at x, x', and x'' are v, v', and v'', respectively, then a load of the above example address would return the following conditional expression tree: $(e_{\mathit{off}} = e'_{\mathit{off}}) \mathbin{?} (e'' \mathbin{?} v' : v'') : (e_{\mathit{off}} = 1 \mathbin{?} x : \_)$.

Function Pointers. At indirect calls, we first use a sound static points-to analysis to enumerate a set of functions F that might be called, then we use *isSat* to prune functions from F that cannot be called given the current path constraint, and finally we fork for each of the remaining possibilities.

Example. Figure 7 demonstrates our heap semantics on a simple program fragment that is shown in the left column. For this example, we use an initial symbolic state in which local variables r_x, r_y, and r_z are assigned fresh symbolic constants x, y, and z, respectively.

The second column shows the symbolic state after executing the store instruction. As we have not yet seen symbolic pointer x, we construct a new primary heap object, l_x, and then add corresponding entries to both the heap ($\mathcal{H}$) and the aliasable objects list ($\mathcal{A}$).

The third column shows the results of executing two versions of the load instruction. In the first version, our static points-to analysis determines that r_x and r_y never alias, so we can simply add a new primary object l_y to the heap. We use a fresh symbolic constant (y_0) to represent the unknown initial contents at location l_y.

In the second version, our points-to analysis concludes that r_x and r_y may alias. Now, when constructing the new object l_y, we must ensure the following heap invariant: $x = y \implies \texttt{load}(y) = 5$. We do this by conditionally initializing l_y to the current value of l_x when $x = y$ (recall Equation (1)). Hence, in this case, the final value of r_z will be a conditional expression, as shown in Figure 7.

3.3 Initializing the Heap with Reaching Definitions

The initial symbolic state (S_{init}) actually contains an *empty* heap that is expanded lazily, as described above. As the heap graph expands, newly uncovered objects are initially *unconstrained*, as represented by the *fresh* symbolic array allocated for each primary object (recall Equation (1), above). This approach can be imprecise for the same reasons discussed in §2.3. We improve precision using reaching definitions, as follows.

We extend the reaching definition analysis from §2.3 to also compute a set of heap writes that *must-reach* the initial program context. Specifically, we compute a set of pairs $R_{\mathit{heap}} = \{(p_i, e_i)\}$, where the heap location referenced by p_i *must* have a value matching e_i in the initial state. We use standard flow functions to compute R_{heap} and we use a static points-to analysis to reason about aliasing.

Then, we modify *addPrimary* to exploit R_{heap}. Specifically, when adding a primary object l_x for symbolic pointer x, we append the following invariant to the current path constraint, *C*:

$$\bigwedge_{(\texttt{ptradd}(x, e_{\mathit{off}}), e_{\mathit{val}}) \in R_{\mathit{heap}}} \mathit{read}(\mathit{fresh}, x_{\mathit{off}} + e_{\mathit{off}}) = e_{\mathit{val}}$$

Above, we enumerate all pairs $(p, e_{\mathit{val}}) \in R_{\mathit{heap}}$ where p has the form $\texttt{ptradd}(x, e_{\mathit{off}})$ or x (which we treat like $\texttt{ptradd}(x, 0)$). For each such pair, we emit the constraint $\mathit{fresh}(x_{\mathit{off}} + e_{\mathit{off}}) = e_{\mathit{val}}$, where *fresh* is the initial symbolic array for l_x as shown in Equation (1).

4. Adding Threads and Synchronization

Figure 8 gives syntax for *SimpThreads*, which adds shared-memory multithreading and synchronization to *SimpHeaps*.

Threads. *SimpThreads* supports cooperative thread scheduling with `yield()`, which nondeterministically selects an-

Program fragment:

store r_x, 5
$r_z \leftarrow$ load r_y

Initial symbolic heap and stack:

$\mathcal{H} = \{\}$
$\mathcal{A} = \{\}$
$\overline{\mathcal{Y}} = \{r_x \mapsto x, r_y \mapsto y, r_z \mapsto z\}$

After the store:

$\mathcal{H} = \{\mathbf{l_x} \mapsto \{\mathbf{5}\}\}$
$\mathcal{A} = \{(\mathbf{x}, \mathbf{l_x}, \mathbf{n_x})\}$
$\overline{\mathcal{Y}} = \{r_x \mapsto x, r_y \mapsto y, r_z \mapsto z\}$

After the load (if x cannot alias y):

$\mathcal{H} = \{l_x \mapsto \{5\}, \mathbf{l_y} \mapsto \{\mathbf{y_0}\}\}$
$\mathcal{A} = \{(x, l_x, n_x), (\mathbf{y}, \mathbf{l_y}, \mathbf{n_y})\}$
$\overline{\mathcal{Y}} = \{r_x \mapsto x, r_y \mapsto y, \mathbf{r_z} \mapsto \mathbf{y_0}\}$

After the load (if x may alias y):

$\mathcal{H} = \{l_x \mapsto \{5\}, \mathbf{l_y} \mapsto \{(\mathbf{x} = \mathbf{y}\,?\,\mathbf{5} : \mathbf{y_0})\}\}$
$\mathcal{A} = \{(x, l_x, n_x), (\mathbf{y}, \mathbf{l_y}, \mathbf{n_y})\}$
$\overline{\mathcal{Y}} = \{r_x \mapsto x, r_y \mapsto y, \mathbf{r_z} \mapsto (\mathbf{x} = \mathbf{y}\,?\,\mathbf{5} : \mathbf{y_0})\}$

Figure 7. Example of our heap semantics on a simple program fragment. The local variables r_x and r_y are pointers to integers, while r_z is an integer. **Bolded** expressions denote updates at each step of execution.

other thread to run. Cooperative scheduling with `yield` is sufficient to model any data race free program. As with other memory errors (recall §3), data races have undefined behavior in C [6, 26] and are runtime errors in *SimpThreads*. Hence, cooperative scheduling is a valid model as we can assume that all *SimpThreads* programs are either data race free or will halt before the first race.

New threads are created by `threadCreate`(e_f, e_{arg}). This spawns a new thread that executes the function call $e_f(e_{arg})$, and the new thread will run until e_f returns. As *SimpThreads* uses cooperative scheduling, the new thread is *not* scheduled until another thread yields control.

Synchronization. We build higher-level synchronization objects such as barriers, condition variables, and queued locks using two primitive parts: cooperative scheduling with `yield`, which provides simple atomicity guarantees, and FIFO wait queues, which provide simple notify/wait operations that are common across a variety of synchronization patterns. Wait queues support three operations: `wait`, to yield control and move the current thread onto a wait queue; `notifyOne`, to wake the thread on the head of a wait queue; and `notifyAll`, to wake all threads on a wait queue.

We use these building blocks to implement standard threading and synchronization libraries such as POSIX threads (pthreads). To aid our symbolic semantics, we assume synchronization libraries have been instrumented with the *annotation functions* listed in Figure 8. Annotation functions are no-ops that do not actually perform synchronization—they merely provide higher-level information that we will exploit, as described later (§4.3, §4.4). The example in Figure 9 demonstrates how to annotate an implementation of pthreads' mutexes. We have written the example in a pseudocode that uses memory operations resembling those in *SimpThreads*.

Note that wait queues are named by pointers. There is an implicit wait queue associated with every memory address—no initialization is necessary. For example, Figure 9 uses the implicit wait queue associated with `&m->taken`. The `futex()` system call in Linux uses a similar design. The reason for naming wait queues by an address rather than an integer id will become clear in §4.3.

$s \in$ *Stmt* ::= ... | `threadCreate`(e_f, e_{arg}) | `yield()`
| `wait`(p) | `notifyOne`(p) | `notifyAll`(p)
synchronization annotations
| `acquire`(p) | `release`(p)
| `barrierInit`(p, e) | `barrierArrive`(p)

Figure 8. New statements for *SimpThreads*.

```
pthread_mutex_lock(mutex *m) {
  while (load ptradd(m, $i_{taken}$))        // while (m->taken)
    wait(ptradd(m, $i_{taken}$));            //   wait(&m->taken)
  store ptradd(m, $i_{taken}$) 1;            // m->taken = 1
  acquire(m);                                // acquire(m)
}
pthread_mutex_unlock(mutex *m) {
  store ptradd(m, $i_{taken}$) 0;            // m->taken = 0
  release(m);                                // release(m)
  notifyOne(ptradd(m, $i_{taken}$));         // notify(&m->taken)
  yield();                                   // yield()
}
```

Figure 9. Pseudocode demonstrating how pthreads' mutexes might be implemented in *SimpThreads*.

Challenges. The primary new challenge introduced by *SimpThreads* is the need to reason about symbolic synchronization objects. Our approach includes a semantics for symbolic wait queues (§4.2) and a collection of synchronization-specific invariants (§4.3) that exploit facts learned from our context-specific dataflow analysis (§4.4).

4.1 Symbolic Semantics

We now extend our symbolic execution algorithm to support *SimpThreads*. As illustrated in Figure 10, we modify $\overline{\mathcal{Y}}$ and *CallCtx* to include one call stack per thread, and we modify *path* to record a multithreaded trace. We add the following domains to the symbolic state:

- T^{Curr}, which is the id of the thread that is currently executing.
- T^{E}, which is the set of enabled threads, *i.e.*, the set of threads not blocked on synchronization. This includes T^{Curr}.
- *WQ*, which is a list that represents a global order of all waiting threads. Each entry of the list is a pair (p, t) sig-

$\overline{y}$	:	***ThreadId*** $\rightarrow$ Stack of $(Var \rightarrow Expr)$	*(local variables)*
CallCtx	:	***ThreadId*** $\rightarrow$ Stack of *StmtLabel*	*(calling contexts)*
path	:	List of (***ThreadId***, *StmtLabel*)	*(execution trace)*
T^{Curr}	:	*ThreadId*	*(current thread)*
T^{E}	:	Set of *ThreadId*	*(enabled threads)*
WQ	:	List of (*Expr*, *ThreadId*)	*(global wait queue)*
$\mathtt{L}^{+}$	:	*ThreadId* $\rightarrow$ Set of *Expr*	*(acquired locksets)*
$\mathtt{B}^{\mathtt{cnts}}$	:	*Expr* $\rightarrow$ Set of *Expr*	*(barrier arrival cnts)*

Figure 10. Symbolic state for *SimpThreads*, with modifications to *SimpHeaps* **bolded**, above the line, and additions shown below.

nifying that thread t is blocked on the wait queue named by address p. The initial *WQ* can either be empty (all threads enabled) or non-empty (some threads blocked, as described in §4.2).

- $\mathtt{L}^{+}$, which describes a set of locks that *may* be held by each thread and is derived from `acquire` and `release` annotations.
- $\mathtt{B}^{\mathtt{cnts}}$, which describes a set of possible arrival counts for each barrier and is derived from `barrierInit` annotations.

$\mathtt{L}^{+}$ and $\mathtt{B}^{\mathtt{cnts}}$ are both *over-approximations*. They are initialized as described in §4.4 and they are used by invariants described in §4.3.

Symbolic Execution. Our first action during symbolic execution is to invoke $step(S_{init}, \mathtt{yield}())$, where `yield` forks execution once for each possible $T^{Curr} \in T^{E}$. This gives each thread a chance to run first. Note that context switches (updates to T^{Curr}) occur *only* either explicitly through `yield`, or implicitly when the current thread exits or is disabled through `wait`. Note also that execution has deadlocked when T^{E} is empty and *WQ* is non-empty.

4.2 Symbolic Wait Queues

We now give symbolic semantics for the three FIFO wait queue operations, `wait`, `notifyOne`, and `notifyAll`. When a thread t calls `wait`(p), we remove t from T^{E} and append the pair (p, t) to *WQ*. When t is notified, we remove it from *WQ* and add it to T^{E}. *Which* threads are notified is answered as follows:

notifyOne(p). Any thread in *WQ* with a matching queue address may be notified. Let (p_1, t_1) be the first pair in *WQ* and let (p_n, t_n) be the last pair. We walk this ordered list and fork execution up to $|WQ| + 1$ times. The possible execution forks are given by the following list of path constraints:

$$\begin{array}{rl} (1) & p_1 = p \\ (2) & p_1 \neq p \wedge p_2 = p \\ & \dots \\ (n) & p_1 \neq p \wedge p_2 \neq p \wedge \dots \wedge p_n = p \\ (n+1) & p_1 \neq p \wedge p_2 \neq p \wedge \dots \wedge p_n \neq p \end{array}$$

In the first fork, we notify t_1, in the second, we notify t_2, and so on, until the nth fork, in which we notify t_n. In the final fork, no threads are notified. Only a subset of these forks may be feasible, so we use *isSat* to prune forked paths that have an infeasible path constraint. In particular, if there exists an i where $p_i{=}p$ *must* be true on the current path, then all forks from $(i{+}1)$ onwards are infeasible and will be discarded. Further, as in §3, we increase precision by using a static points-to analysis to determine when it *cannot* be true that $p_i{=}p$. These semantics are simple but reveal a key design decision: by folding all concrete wait queues into a single global queue, *WQ*, we naturally allow each wait queue to be named by *symbolic* addresses.

notifyAll(p). Any subset of threads in *WQ* may be notified. We first compute the powerset of *WQ*, $\mathcal{P}(WQ)$, and then fork execution once for each set $S \in \mathcal{P}(WQ)$. Specifically, on the path that is forked for set S, we notify all threads in S and apply the following path constraint:

$$\bigwedge_{(p_i,t_i)\in WQ} \begin{cases} p_i = p & \mathtt{if}\ (p_i, t_i) \in S \\ p_i \neq p & \mathtt{otherwise} \end{cases}$$

This forks execution $2^{|WQ|}$ ways, though we expect that *isSat* and a points-to analysis will prune many of these in practice.

Initial Contexts with a Nonempty *WQ*. Suppose we want to analyze an initial program context in which some subset of threads begin in a *waiting* state, but we do not know the order in which the threads began waiting. One approach is to fork for each permutation of the wait order, but this is inefficient. Instead, our approach is to add *timestamp counters*. First, we tag each waiting thread with a timestamp derived from a global counter that is incremented on every call to `wait`, so that thread t_1 precedes thread t_2 in *WQ* if and only if t_1's timestamp is less than t_2's timestamp.

Then, we set up the program context so that each waiting thread begins with the call to `wait` it is waiting in. Before beginning normal symbolic execution, we execute these `wait` calls in any order, using the semantics for `wait` described above, but with one adjustment: we give each waiting thread t_i a symbolic timestamp, represented by the symbol x_i, and we bound each $x_i < 0$ so these waits occur before other calls to `wait` during normal execution. We say that $x_i < x_k$ is true in the concrete initial state when t_i and t_k are waiting on the same queue and t_i precedes t_k on that queue.

Next, we update the semantics of `notifyOne`. If there are n threads in *WQ* and w of those threads are *initial waiters*, meaning they have symbolic timestamps, then `notifyOne` uses the following sequence of path constraints, where $1 \leq i \leq w$:

$$\begin{array}{rl} (i) & p_i = p \wedge \left(\bigwedge_{1\leq k\leq w, k\neq i} (p_k = p) \Rightarrow (x_i < x_k) \right) \\ (w+1) & p_1 \neq p \wedge p_2 \neq p \wedge \dots \wedge p_w \neq p \wedge p_{w+1} = p \\ & \dots \\ (n) & p_1 \neq p \wedge p_2 \neq p \wedge \dots \wedge p_n = p \\ (n+1) & p_1 \neq p \wedge p_2 \neq p \wedge \dots \wedge p_n \neq p \end{array}$$

The first w constraints handle the cases where an initial waiter is notified. We can notify initial waiter t_i if it has a matching queue address, $p_i{=}p$, and it precedes all other initial waiters t_k with a matching address. The cases for $w{+}1$ and above are as before.

4.3 Synchronization Invariants

The semantics described above are sound, but approximations in the initial state can cause symbolic execution to explore infeasible paths. In an attempt to avoid infeasible paths, we augment the path constraint with higher-level program invariants. Specifically, we propose a particularly high-value set of *synchronization invariants*. We focus on synchronization invariants here since the novelty of our work is symbolic exploration of multithreaded programs with symbolic synchronization. More generally, high-level invariants always help reduce explosion of infeasible paths and we could easily integrate programmer-specified invariants.

We cannot apply synchronization invariants without first identifying synchronization objects. Ideally we would locate such objects by scanning the heap, but our core language is untyped, so we cannot soundly determine the type of an object by looking at it. (This conservatively models our target language, C, where potentially unsafe type casts are prevalent.) Instead, we apply invariants when synchronization functions are called. For example, we instrument the implementation of `pthread_mutex_lock(m)` to apply invariants to `m` as the first step before locking the mutex. The rest of this section describes the invariants we have found most useful.

Locks. As illustrated in Figure 9, locks can be modeled by an object with a `taken` field that is non-zero when the lock is held and zero when the lock is released. Suppose a thread attempts to acquire a lock whose `taken` field is symbolic: execution must fork into two paths, one in which `taken`=0, so the lock can be acquired, and another in which `taken`≠0, so the thread must wait. One of these paths may be infeasible, as illustrated by Figure 1, so we need to further constrain lock objects to avoid such infeasible paths.

We use *locksets* to constrain the `taken` field of a lock object. Given a symbolic state with locksets $\mathtt{L}^+$ and a pointer p to some lock object, the lock's `taken` field can be non-zero only when there exists a thread T and an expression e, where $e \in \mathtt{L}^+(T)$, such that $e = p$. This invariant is expressed by the following constraint, where e_i ranges over all locks held by all threads:

$$(\mathtt{taken} = 0) \;\Leftrightarrow\; \left(\bigwedge_{e_i \in \mathtt{L}^+(*)} e_i \neq p \right)$$

Our dataflow analysis computes $\mathtt{L}^+$ for the initial symbolic state (§4.4). We keep $\mathtt{L}^+$ up-to-date during symbolic execution using the `acquire` and `release` annotations: on $\mathtt{acquire}(p)$ we add p to $\mathtt{L}^+(T^{Curr})$, and on $\mathtt{release}(p)$ we remove e from $\mathtt{L}^+(T^{Curr})$ where e must-equal p on the current path.

Barriers. A pthreads barrier can be modeled by two fields, `expected` and `arrived`, and a wait queue, where `arrived` is the number of threads that have arrived at the barrier, the barrier triggers when `arrived=expected`, and the wait queue is used to release threads when the barrier triggers.

Suppose a program has N threads spin in a loop, where each loop iteration includes a barrier with `expected=`N. Now suppose we analyze the program from an initial context where the barrier is unconstrained. When the first thread arrives at the barrier, execution forks at the condition `arrived=expected`. In the true branch we set `arrived=0` and notify the queue, and in the false branch we increment `arrived` and wait. This repeats for the other threads, and an execution tree unfolds in which we explore $O(2^N)$ paths through a code fragment that has exactly one feasible path.

We compute invariants for both of these fields. Bounds for `arrived` can be determined by examining *WQ*: the number of threads that have arrived at a barrier is exactly the number of threads that are waiting on the barrier's wait queue. Let q be the wait queue address used by the barrier and let C be the current path constraint. We compute conservative lower- and upper-bounds for `arrived`. The lower-bound L is the number of pairs $(p, t) \in WQ$ for which $mustBeTrue(C, p{=}q)$, and the upper-bound H is the number of pairs for which $mayBeTrue(C, p{=}q)$. Given these bounds, the invariant is $L \leq \mathtt{arrived} \leq H$.

A barrier's `expected` count is specified during barrier initialization, *i.e.*, when `pthread_barrier_init` is called. Each symbolic state contains a $\mathtt{B}^{\mathtt{cnts}}$ that maps barrier pointers p to a set of expressions that describes the set of possible `expected` counts for all barriers pointed-to by p. So, we can use $\mathtt{B}^{\mathtt{cnts}}$ directly to construct an invariant for `expected`:

$$\bigwedge_{p' \in \mathtt{B}^{\mathtt{cnts}}} \left(\bigvee_{e \in \mathtt{B}^{\mathtt{cnts}}(p')} (p' = p) \Rightarrow (\mathtt{expected} = e) \right)$$

$\mathtt{B}^{\mathtt{cnts}}$ is computed for the initial state (see §4.4) and does not change during symbolic execution—when evaluating a call to `pthread_barrier_init` during symbolic execution, we write to the barrier's `expected` field directly, making $\mathtt{B}^{\mathtt{cnts}}$ irrelevant for this case.

Other Types of Synchronization. The invariant described above for a barrier's `arrived` field is more generally stated as an invariant on the size of a given wait queue, making it applicable to other data structures that use wait queues, such as condition variables and queued locks.

Why Wait Queues are Named by Address. For standard synchronization objects such as barriers, condition variables, and queued locks, different objects do not share the same wait queue. For example, notifying the queue of lock L should not notify threads waiting at any other lock. By using the address of L to name L's wait queue, we state this invariant implicitly.

For contrast, suppose we instead named wait queues by an integer id. We would be forced to add a `queueId` field to each lock, then state the following invariant: $\forall p_1, p_2 : (p_1 = p_2) \Leftrightarrow (id_1 = id_2)$, where p_1 and p_2 range over the set of pointers to locks, and where id_1 and id_2 are the `queueId` fields in p_1 and p_2, respectively. Stating this as an axiom would require enumerating the complete set of pointers to locks, which can be extremely inefficient.

4.4 Approximating the Initial State of Synchronization

We update the context-specific dataflow framework introduced in §2.3 to support multiple threads. Specifically, we apply the dataflow framework as described in §2.3 to each thread, separately, and then combine the per-thread results to produce a multithreaded analysis. We perform the following analyses for *SimpThreads*:

Reaching Definitions. We update the reaching definitions analysis described in §3.3 to support multiple threads. Importantly, since we analyze each thread in isolation, we must reason about cross-thread interference. Our approach is to label memory locations in R_{heap} as either *conflict-free* or *shared*. A location is conflict-free if it is provably thread-local (via an escape analysis) or if all writes to the location must-occur before the first call to `threadCreate`—the second case captures a common idiom where the main thread initializes global data that is kept read-only during parallel execution. Shared locations may have conflicts—we reason about these conflicts using *interference-free regions* [16].[1]

Locksets. We use a lockset analysis to compute $\mathtt{L}^+(T)$, the set of locks that *may* be held at thread T's initial program counter. Our analysis uses relative locksets as in RELAY [40]: each function summary includes two sets, $\mathtt{L}_f^+$ and $\mathtt{L}_f^-$, where $\mathtt{L}_f^+$ is the set of locks that function f *may acquire* without releasing, and $\mathtt{L}_f^-$ is the set of locks that f *always releases* without first acquiring.

The key difference between our implementation and RELAY's is that we compute may-be-held sets while RELAY computes must-be-held sets. This reflects differing motivations: as a static race detector, RELAY wants to know which locks *must* be held to determine if accesses are properly guarded, but we want to know which locks *may* be held to determine when two `lock()` calls may need to be serialized (as motivated by Figure 1). Hence, our $\mathtt{L}^+$ and $\mathtt{L}^-$ are may-acquire and must-release, while those used by RELAY are must-acquire and may-release.

Barrier Expected Arrivals. To compute $\mathtt{B}^{\mathtt{cnts}}$, we simply enumerate all calls to $\mathtt{barrierInit}(p, e)$ that might be performed on some path from program entry up to the initial context, and for each such call, we add e to the set $\mathtt{B}^{\mathtt{cnts}}(p)$. This can be viewed as *may-reach* analysis applied to each barrier's `expected` field.

[1] A formal description of the full reaching definitions analysis is given in the first author's Ph.D. thesis [1].

Merging Thread-Local Analyses. Note that each of the above three analyses is purely thread-local—we must merge these analyses into a summarized initial program state.

For reaching definitions, the local variable definitions (R_{local}) are trivially separate for each thread, so they can be applied directly to the thread-local stacks as described in §2.3. For the heap definitions, each thread-local analysis produces a set of definitions R^T_{heap}, and all of these definitions must be merged. As it turns out, we can simply union all R^T_{heap} into a single summarized R_{heap}. This follows from the following observations: first, our dataflow analysis ignores data races (recall the discussion in §1.1), and second, our use of interference-free regions guarantees that no location p will be defined in both R^{T1}_{heap} and R^{T2}_{heap} unless $T1 = T2$ or there is a data race on p (see [16] for a proof).

For locksets, each thread-local analysis for thread T computes $\mathtt{L}^+(T)$, and each of these are trivially merged into $\mathtt{L}^+$. Similarly, we construct a summarized barrier expected arrivals ($\mathtt{B}^{\mathtt{cnts}}$) by taking the union of all $\mathtt{B}^{\mathtt{cnts}}_T$ that were computed by thread-local analyses.

Barrier Matching. A large class of data-parallel algorithms use barriers to execute threads in lock-step. For example, a program might execute the following loop in N different threads, where each iteration happens in lock-step:

```
for (i=0; i < Z; ++i) { barrierArrive(b); ... }
```

Suppose we are given an initial program context in which each thread begins inside this loop. In this case, since the loop runs in lock-step, we know that all threads must start from the same dynamic loop iteration, so we can add a constraint that equates the loop induction variable, `i`, across all threads. This constraint is included in the initial path constraint, $S_{init}.C$.

This is the *barrier matching* problem: given two threads, must they pass the same sequence of barriers from program entry up to the initial context? Solutions have been proposed—we adapt the algorithm from Zhang and Duesterwald [43], which builds *barrier expressions* to describe the possible sequence of barriers each thread might pass through. Two threads are barrier-synchronized if their barrier expressions are compatible.

Zhang and Duesterwald's algorithm does not support our use case directly because it cannot reason about loops with input-dependent trip counts. So, we extend that algorithm by computing a symbolic trip count for each loop node in a barrier expression. Two loops match if their symbolic trips counts *must* be equal. We compute trip counts using a standard algorithm, but we discard trip counts that depend on *shared* memory locations (recall the definition of *shared*, from above). To determine if the trip count can be kept, we compute a backwards slice of the trip count expression and ensure that slice does not depend on any *shared* locations.

5. Soundness and Completeness

Our symbolic execution algorithm is sound, and it is complete except when the SMT solver uses concretization to make progress through an unsolvable query (recall §2.1). We make this claim only for programs with a correctly implemented pthreads library; otherwise, the invariants from §4.3 would be incorrect. Our theorem relies on a notion of correspondence between concrete and symbolic states—because the heap is expanded lazily in the symbolic semantics, this notion relies on *partial equivalence* and is somewhat technical. We give the full concrete semantics, a full definition of correspondence, and a proof of the theorem in an expanded version of this paper [3]. Our proof applies to our symbolic semantics only—it implicitly assumes that the initial symbolic state is a correct over-approximation for the given program context.

Definition 1 (Correspondence of concrete and symbolic states)**.** *We say that symbolic state S_S* models *concrete state S_K* under *constraint C if there exists an assignment Σ that assigns all symbolic constants in S_S to values such that (a) Σ is a valid assignment under the constraint C, and (b) the application of Σ to S_S produces a state that is* partially-equivalent *to S_K (as defined in the expanded version of this paper).*

Theorem 1 (Soundness and completeness of symbolic execution)**.** *Consider an initial program context, an initial concrete state S_K for that context, and an initial symbolic state S_S:*

- ***Soundness:*** *If symbolic execution from S_S outputs a pair (p, C), then for all S_K such that S_S models S_K under C, concrete execution from S_K must follow path p as long as context switches happen exactly as specified by path p.*
- ***Completeness:*** *If concrete execution from S_K follows path p, then for all S_S such that S_S models S_K under S_S.C, symbolic execution from S_S will either (a) output a pair (p, C), for some C, or (b) encounter a query that the SMT solver cannot solve.*

6. Implementation

We implemented the above algorithms on top of the Cloud9 [9] symbolic execution engine, which is in turn based on KLEE [10]. Cloud9 symbolically executes C programs that use pthreads and are compiled to LLVM [30] bitcode (Cloud9 operates directly on LLVM bitcode). Where a points-to analysis is needed, we use DSA [29].

The C language allows casts between pointers and integers. This is not modeled in our semantics but is partially supported by our implementation. Our approach is to represent each pointer expression p like any other integer expression. Then, at each memory access, we analyze p to extract $(base, offset)$ components. For example, our implementation represents `int *p=&a[x*3]` as $p = a + 4 \cdot (x \cdot 3)$, and to access p we transform it to $\texttt{ptr}(a, 12 \cdot x)$. We determine that a is the *base* address by exploiting LLVM's simple type system to learn which terms are used as pointers.

The precise semantics of integer-to-pointer conversions in C are implementation-defined (§6.3.2.3 of [26]). Our implementation does not support programs that use integer arithmetic to jump between two separately-allocated objects, such as via the classic "XOR" trick for doubly-linked lists. Such programs are not amenable to garbage collection for analogous reasons [5], even though such programs are supported by some C implementations.

7. Evaluation

We ran our experiments on an 8-core 2.4GHz Intel Xeon E5462 with 10GB of RAM. Although our testbed is multi-core, Cloud9 is not designed to exploit multiple cores. We ran one experiment at a time to minimize memory pressure and other system effects that could add experimental noise. We selected applications from standard multithreaded benchmark suites [4, 41] to cover a range of parallelism styles, including fork-join parallelism, barrier-synchronized parallelism, task parallelism, and pipeline parallelism.

Evaluation: Infeasible Paths. Recall (§1.1) that our approach lies on a spectrum between a *naïve* approach, which approximates the initial state very conservatively by leaving all memory locations unconstrained, and a *fully precise* approach, which constructs a perfectly precise initial state using an intractably expensive analysis.

We first compare the *naïve* approach with our approach: *how many fewer infeasible paths do we explore?* We answer this question for a given program context C by exhaustively enumerating all paths reachable from C up to a bounded depth. Any path that is enumerated by the *naïve* approach, but not by our approach, *must* be an infeasible path that our approach has avoided. We use a bounded depth to make exhaustive exploration feasible.

Table 1 summarizes our results. Each row summarizes experiments for a unique program context. For each benchmark application, we manually selected one or two program contexts in which at least two threads begin execution from the middle of a core loop. Column 2 shows the number of threads used in each initial context, and Column 3 shows the maximum number of conditional branches executed on each path during bounded-depth exploration.

Columns 4 and 7 show the number of paths explored by our fully optimized approach (*Full*) and the *naïve* approach, respectively. To further characterize our approach, we also ran our approach with optimizations disabled: *-RD* disables reaching definitions (§2.3, §3.3, §4.4) and *-SI* disables synchronization invariants (§4.3, §4.4). Our approach explores significantly fewer infeasible paths compared to the *naïve* approach, and a comparison across Columns 4–7 shows that each optimization is essential.

It is difficult to compare our approach with the *fully precise* approach, as the *fully precise* approach is intractable.

Program Context			Num Paths				Avg IPS					Exec Time in *isSat*					inf.
	thr	br	*Full*	*-RD*	*-SI*	*N*	*WP*	*Full*	*-RD*	*-SI*	*N*	*WP*	*Full*	*-RD*	*-SI*	*N*	pths
blackscholes	4	20	763	1087	765	1087	927	176	1171	178	1206	75%	93%	65%	93%	65%	–
dedup-1	5	10	103	122	863	971	4731	72	49	67	64	3%	30%	62%	36%	51%	–
dedup-2	5	12	458	550	1811	1904	4692	45	26	39	32	5%	35%	64%	30%	59%	–
lu-1	4	22	681	1026	1133	1864	3997	93	170	64	107	2%	55%	16%	75%	57%	625
lu-2	4	18	554	1400	1290	4680	3860	80	136	105	162	32%	57%	23%	56%	26%	380
pfscan	3	18	246	246	3785	3785	6250	5368	5650	5254	5503	17%	28%	25%	15%	13%	–
streamcluster	3	11	60	617	229	1004	5382	161	59	7	19	15%	9%	35%	74%	31%	48

Table 1. Results for manually-selected program contexts. *Full* is our fully optimized approach, and *N* is the *naïve* approach.

For `lu` and `streamcluster`, we have manually inspected the paths explored by our approach (Column 4) and estimated, through our best understanding of the code, how many of those paths are infeasible (Column 18). Sources of infeasible paths include the following: Both programs assign each thread a unique *id* parameter (*e.g.*, by incrementing a global counter), but we are unable prove that these *id*s are unique across threads. We suspect that a similar situation causes infeasible paths in other applications, but we have not quantified this precisely. Further, they performs calls of the form `pthread_join(t[i])`—we are unable to prove that each `t[i]` is a valid thread id, so we must fork for (infeasible) error cases.

Evaluation: Performance. Columns 8–12 show the average number of LLVM instructions executed per second (IPS), and Columns 13–17 show the percentage of total execution time devoted to *isSat*. The two metrics are correlated, as slower *isSat* times lead to lower IPS. *Full* uses more precise constraints than the *naïve* approach, but this does not necessarily lead to higher IPS for *Full*. Namely, precise and simple constraints such as $x = 5$ lead to high IPS, but precise and complex constraints can lead to low IPS—the latter effect has been observed previously [25, 28].

To further understand the overheads of our approach, we symbolically executed multiple *whole program* paths that each begin at program entry and pass through the initial context (*WP* in Columns 8 and 13). Although *Full* can be an order-of-magnitude slower than *WP*, many paths explored by *WP* visit 100s of branches before reaching the initial context, suggesting that exhaustive summarization of all paths from program entry is infeasible—approximating the initial context is *necessary*. Further profiling shows that much of our overhead comes from resolving symbolic pointers: LLVM's `load` and `store` instructions typically comprised 15% to 50% of total execution time in *Full*, but $< 5\%$ in *WP*.

Lastly, we tried disabling our use of a points-to analysis to restrict aliasing (§3.2, §4.2). With this optimization disabled, each symbolic pointer was assigned 100s of aliases, leading to large heap-update expressions and poor solver performance—so slow that on most benchmarks, throughput decreased to well under 5 IPS. Hence, we consider this optimization so vital that we left it enabled in all experiments.

Evaluation: Input-Covering Schedules. We evaluate how well our techniques support an algorithm for finding *input-covering schedules* [2]. The algorithm partitions execution into *epochs* of bounded length and uses symbolic execution to enumerate a set of input-covering schedules for each epoch. The beginning of each epoch is defined by a multithreaded program context with fully specified call stacks, so our techniques are directly applicable.

We ran the algorithm (from [2]) on the programs shown in the table below, using our approach at various optimization levels, along with the *naïve* approach. We report the number of schedules enumerated by that algorithm and the algorithm's total runtime. Similarly to the infeasible paths evaluation above, if a schedule is enumerated with the *naïve* approach, but not our approach, then it must be an *infeasible* schedule. The results show, again, that our techniques are essential: the *naïve* approach suffers from slower algorithm runtimes and more infeasible schedules.

Program		NumSchedules / RunningTime			
	thr	*Full*	*-RD*	*-SI*	*N*
fft	2	2/ 9s	2/12s	2/ 10s	2/ 11s
lu	4	3/ 6s	23/14s	1550/396s	1976/202s
pfscan	2	455/24s	455/28s	2245/ 78s	2273/ 80s

8. Related Work

We have discussed related work throughout the paper. We stress that prior approaches to path explosion, such as summarization [20, 24, 36], heuristics [8, 31, 34], path merging [25, 28], and partial order reductions [14, 18], are completely orthogonal to our symbolic execution semantics and could be profitably incorporated along with our ideas. We believe our integration of dataflow analysis with symbolic execution is novel, and §7 shows that our choice of analysis represents an essential sweet spot in our context. However, we do not claim that our dataflow analysis is *powerful* in a novel way. We refer to Rinard for a thorough survey of dataflow analysis of multithreaded programs [37].

The recently proposed idea of *micro execution* [21] has a similar goal to our work—"virtual" execution (which includes symbolic execution) from arbitrary program contexts—but the MicroX system does not reason as precisely about the symbolic heap, and most importantly, it does not consider the effect of the surrounding program context (making it equivalent to the naïve approach we compare with in §7).

Some additional prior work (not yet discussed) has investigated symbolic pointers. Otter [32] appears to use a similar conditional-aliasing approach with lazy initialization, but its approach is not explained formally. SAGE [17] soundly handles interior pointers, similarly to our semantics, but does not support symbolic object sizes or symbolic base locations. CUTE [38] symbolically executes C unit tests and supports symbolic pointer inputs, but it can reason about only those aliasing relationships that are explicitly stated in program branches, such as `if(p1!=p2)`. Pex [39] exploits static types (not available to us) and cannot reason soundly when type casts are involved (our approach is sound on an untyped language). We originally implemented the algorithm from Khurshid *et al.* [27], which explores each possible memory graph via aggressive forking, but this suffered from unacceptably extreme path explosion.

Table 2 gives a brief comparison of our approach as compared to the most related prior approaches.

9. Conclusions

We proposed the "symbolic execution from arbitrary contexts" problem and described a solution. In solving this problem, we found it profitable to integrate dataflow analyses with symbolic execution. Specifically, our evaluation in §7 showed that two classes of dataflow analyses are particularly profitable: *reaching definitions*, to summarize the state of memory in a general way, and *locksets* and *barrier matching*, to summarize the state of synchronization objects in a specific way. In broader terms, we believe that practical solutions to this problem *must* construct the initial symbolic state using a scalable analysis of some sort, and we have shown that scalable dataflow analyses can be a good fit.

Our solution gains scalability at the cost of precision. We believe this trade-off is necessary, as constructing a fully-precise summary of the initial context would be prohibitively expensive. Hence, unlike classic approaches to symbolic execution, which are fully precise because they explore feasible paths only, our approach is imprecise and can explore *infeasible paths*—this can lead to *false positives* in verification tools, testing tools, and other analyses that build on our symbolic execution techniques.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We also thank the members of the Sampa and PLSE groups at UW for their feedback and fruitful discussions. This work was supported in part by the National Science Foundation CAREER award 0846004, a Google Ph.D. Fellowship awarded to Tom Bergan, and a gift from Microsoft.

References

[1] T. Bergan. *Avoiding State-Space Explosion in Multithreaded Programs with Input-Covering Schedules and Symbolic Execution*. PhD thesis, Computer Science Dept., University of Washington, Seattle, WA, March 2014.

[2] T. Bergan, L. Ceze, and D. Grossman. Input-Covering Schedules for Multithreaded Programs. In *OOPSLA*, 2013.

[3] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. Technical Report UW-CSE-13-08-01, Univ. of Washington.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[5] H.-J. Boehm. Simple Garbage-Collector-Safety. In *PLDI*, 1996.

[6] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[7] S. Böhme and M. Moskal. Heaps and Data Structures: A Challenge for Automated Provers. In *Proceedings of the 23rd International Conference on Automated Deduction*, 2011.

[8] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, 2008.

[9] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[11] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A Reachability Predicate for Analyzing Low-Level Software. In *TACAS*, 2007.

[12] A. Cheung, A. Solar-Lezama, and S. Madden. Partial Replay of Long-Running Applications. In *FSE*, 2011.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*, 2011.

[14] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded Partial-Order Reduction. In *OOPSLA*, 2013.

[15] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *PLDI*, 2011.

[16] L. Effinger-Dean, H.-J. Boehm, P. Joisha, and D. Chakrabarti. Extended Sequential Reasoning for Data-Race-Free Programs. In *Workshop on Memory Systems Performance and Correctness*, 2011.

[17] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *ISSTA*, 2009.

[18] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[19] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.

[20] P. Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[21] P. Godefroid. Micro Execution. In *ICSE*, 2014.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

System	Can start from the middle of execution?	Supports multithread programs?	Supports symbolic ptrs? base	offset
KLEE [10]	no	no	yes *(slow)*	yes
Cloud9 [9]	no	yes	yes *(slow)*	yes
Otter [32]	yes *(naïve)*	no	yes	yes
bbr [12]	yes *(naïve)*	no	yes	yes
MicroX [21]	yes *(naïve)*	yes	no	yes
This paper	yes *(dataflow)*	yes	yes	yes

Table 2. Comparison with a few key related symbolic execution tools. For *yes* answers in the second column, we also summarize (in parentheses) the method used to compute the initial symbolic state. In the last two columns, we summarize how well each system supports symbolic *base* and *offset* components of a symbolic pointer.

[23] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*, 2008.

[24] P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *ISSTA*, 2011.

[25] T. Hansen, P. Schachte, and H. Sondergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *Intl. Conf. on Runtime Verification (RV)*, 2009.

[26] ISO. *C Language Standard, ISO/IEC 9899:2011*. 2011.

[27] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, 2003.

[28] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[29] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[30] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, 2004.

[31] Y. Li, Z. Su, L. Wang, and X. Li. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA*, 2013.

[32] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Analysis of Multithreaded Programs. In *Static Analysis Symposium (SAS)*, 2011.

[33] L. D. Moura and N. Bjrner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[34] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, 2007.

[35] C. S. Pasareanu, N. Rungta, and W. Visser. Symbolic Execution with Mixed Concrete-Symbolic Solving. In *ISSTA*, 2011.

[36] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing Procedures in Concurrent Programs. In *POPL*, 2004.

[37] M. Rinard. Analysis of Multithreaded Programs. In *Static Analysis Symposium (SAS)*, 2001.

[38] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *FSE*, 2005.

[39] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Tests and Proofs (TAP)*, 2008.

[40] J. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *FSE*, 2007.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[42] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated Debugging for Arbitrarily Long Executions. In *HotOS*, 2013.

[43] Y. Zhang and E. Duesterwald. Barrier Matching for Programs With Textually Unaligned Barriers. In *PPoPP*, 2007.