RacerX: Effective, Static Detection of Race Conditions and Deadlocks

Dawson Engler and Ken Ashcraft
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

ABSTRACT

This paper describes RacerX, a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. It is explicitly designed to find errors in large, complex multithreaded systems. It aggressively infers checking information such as which locks protect which operations, which code contexts are multithreaded, and which shared accesses are dangerous. It tracks a set of code features which it uses to sort errors both from most to least severe. It uses novel techniques to counter the impact of analysis mistakes. The tool is fast, requiring between 2-14 minutes to analyze a 1.8 million line system. We have applied it to Linux, FreeBSD, and a large commercial code base, finding serious errors in all of them.

Keywords

Race detection, deadlock detection, program checking.

General Terms

Reliability, Verification.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*reliability,validation*; D.4.5 [**Operating systems**]: Reliability—*verification*

1. INTRODUCTION

The difficulty of finding data races and deadlocks is well known. Detecting such errors with testing is hard since they often depend on intricate sequences of low-probability events. This makes them sensitive to timing dependencies, workloads, the presence or absence of print statements, compiler options, or slight differences in memory models. This sensitivity increases the risk that errors will elude in-house regression tests yet make grand entrances when software is released to thousands of users. Further, even if a test case happens to trigger an error it can be difficult to tell that it has happened. Data races in particular are hard to observe since often they quietly violate data structure invariants rather than cause immediate crashes. The effects of these violations only manifest millions of cycles after the error occurred, making it hard to trace back to the root cause.

Because of these problems, many approaches have been developed to catch data races (deadlocks have received less attention). Language-level approaches attempt to simplify concurrency control by providing higher-level constructs that disallow uses that are error-prone. The most successful such effort are monitor-based primitives, which showed up early in systems programming [23] and have recently been incorporated in such languages as Java [18]. Other languages allow programmers to statically bind shared variables to the locks that protect them [9]. However, while languages can ameliorate some of the complexity of concurrency, they do not eliminate its problems. Common errors include simply not protecting shared data consistently, or using too-small critical sections to do so (such that a violated invariant is visible to other threads), or having locking cycles. While recent efforts have made progress in addressing some of these limitations [1, 19], in the end a language approach to protection is rather drastic: all code must be written in the language to get any benefits from it, which prevents system builders from choosing any other language that may better suit their needs. Also, language definitions tend to be relatively slow moving, and thus difficult to adapt as analysis sophistication increases.

As a result, there have been many tools developed to attack the problem: dynamic, post-mortem, and static as well as model checking.

The best known dynamic tool is the Eraser data race detector [29], which dynamically tracks the set of locks held during program execution. Eraser uses these "locksets" to compute the intersection of all locks held when accessing shared state. Shared locations that have an empty intersection are flagged as not being consistently protected. Since set intersection is commutative, Eraser can flag errors irrespective of actual thread interleavings. Choi et al [7] have since improved upon Eraser. Most significantly, they incorporate static analysis to remove unnecessary checks from the runtime analysis. This reduces the runtime analysis overhead from factors of 10 in the original Eraser to 13 - 42 percent. While they do find errors in real programs, it is unclear how effective their solution would be on a multi-million line operating system. Tools similar to Eraser have been developed for Cilk programs [6] and Java [7]. Prior tools [13, 25,

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

27] used a more chancy approach based on Lamport's "happens before" relationship [22] that only checked the scheduling interleavings that occurred while testing.

The strength of dynamic tools is that by operating at runtime they only visit feasible paths and have accurate views of the values of variables and aliasing relations. However, dynamic monitoring has a heavy computational price, making it time consuming to run test cases and impossible on programs that have strict timing requirements. High overheads mean that while in theory such tools can compute arbitrarily precise information, in practice they are limited to what can be computed efficiently (both in time and space). Furthermore, their reliance on invasive instrumentation typically precludes their use on low-level code such as OS kernels, device drivers, and embedded systems, yet these are precisely the applications for which concurrency errors are the most dangerous. Finally, they can only find errors on executed paths. Unfortunately, the number of feasible paths grows roughly exponentially with the size of code. This means that in practice testing can only exercise a tiny fraction of all feasible paths, leaving large systems with a residue of errors that could take weeks of execution to manifest. Even worse, much of the code in a large OS *cannot* be run. The bulk of such code resides in device drivers, and only a small fraction of these drivers can be tested at a typical site, since there is usually a small number of installed devices.

Post-mortem techniques [21] analyze log or trace data after the program has executed in a manner similar to dynamic techniques. While post-mortem analyses can affect performance less than dynamic analyses they suffer from the same limitation as dynamic techniques in that they can only find errors along executed paths.

Another way to find races is to use model checking [8], which is a formal verification technique that can be viewed as a more comprehensive form of dynamic testing. Model checking takes a simplified description of the code and exhaustively tests it on all inputs, using techniques to explore vast state spaces efficiently. It grew out of the need to explore the massive state spaces in hardware circuits, in part to find concurrency errors. It can provide help for finding software races as well. The Java PathFinder [2] and Bandera [10] projects use model checking to find errors in concurrent Java programs. The more specialized Teapot system was developed expressly for finding errors (including concurrency errors) in software-based multiprocessor cache coherence protocols [5]. Unfortunately, while model checking trumps testing in terms of state space exploration, getting a large system into a model checker is still rare. It requires significant effort both to specify the system (which can involve writing an abstract specification in a simplified programming language) and in scaling it down enough to execute in the model checked environment. Model checking an entire system the size of an OS is a long way off.

At the other end of the spectrum are static tools. While these have less precise local information, they can provide significant advantages for large code bases. Unlike a dynamic approach, static analysis does not require executing code: it immediately finds errors in obscure code paths that are difficult to reach with testing. Because they occur offline they can also do analysis impractical at runtime.

Two of the better known static race detection approaches are the Warlock tool for finding races in C programs [30] and the Extended Static Checking [12] (ESC) and ESC/Java [24] tools for Modula-3 and Java respectively, which use a theorem prover to find errors. Burrows et al [3] have since extended ESC/Java to check for stronger properties than unprotected variable accesses, which is the only error most prior tools flag. Unfortunately, in part because of lack of precision at compile time, both Warlock and ESC make heavy use of annotations to inject knowledge into the analysis and to reduce the number of false positives. Anecdotally this caused problems when applying Warlock to large code bases; sophisticated code requires many annotations just to suppress spurious errors. Initial measurements from Flanagan and Freund [15] give a more quantitative feel for the state-of-the-art in annotation-based race checking — they measured an overhead of one annotation per 50 lines of code at a cost of one programmer hour per thousand lines of code [15]. Applying this approach to a several million line operating system would require roughly 100 continuous days of annotations. While there has been work on partially automating the annotation process, current tools still require much manual assistance. For example, Houdini [16] was able to reduce the number of non-concurrency annotations needed on a 36,000 line program from 3679 to 2037.

We want to build tool that effectively finds data races and deadlocks in large, complex systems. The main rules of this game are as follows:

1. The tool should need no annotations other than an indication as to what functions are used to acquire and release locks. We want do not want users to have to invest time annotating large portions of their system. (Though, we do we welcome any annotations they do provide.) In particular it must automatically infer checking information and have automatic ways of handling common false positives without user intervention.
2. The tool must be able to separate out potentially severe errors from the mass of violations it will find. In our experience the key problem with race detection is not the analysis needed to find errors — even simple analysis can find thousands of unprotected shared variable accesses in a large system. Rather, the hard problem is finding those errors that can actually cause problems.
3. The tool must minimize the impact of analysis mistakes and the false positives they cause. Empirically, if the first few errors a user inspects are false positives, or if there are too many false positives in a row, users will stop inspecting a tool's output and even discard the tool.
4. The tool must scale to millions of lines of code both in speed and in its ability to report complex errors.

Ideally, we want users to specify their system's locking functions, run the tool, and immediately find serious errors.

This paper describes RacerX, a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. It aggressively infers checking information such as which locks protect which operations, which code contexts are multithreaded, and which shared accesses are dangerous. It tracks a set of code features which it uses to sort errors from most to least severe. It uses novel techniques to counter the impact of analysis mistakes — for example by selecting the results of those paths that are the most trustworthy, cross-checking decisions in multiple ways, and inferring which semaphores are being used for mutual exclusion as opposed to unilateral synchronization. The tool is fast, requiring somewhere between 2-14 minutes to analyze a 1.8 million line system. We have applied it to Linux, FreeBSD, and a large commercial code base. It has found serious errors in all of them.

The next section gives an overview of RacerX. Section 3 describes the generic lockset analysis. Sections 4—7 describe: the deadlock checker, how we make it more accurate, how we mitigate lockset mistakes, and results. Section 8 describes the race detector and its results. Section 9 concludes.

2. OVERVIEW

This section gives an overview of RacerX and the systems we check. At a high level, checking a system with RacerX involves five phases: (1) retargeting it to system-specific locking functions, (2) extracting a control flow graph from the checked system, (3) running the deadlock and race checkers over this flow graph, (4) post-processing and ranking the results, (5) inspection. The first and last phases are done by the user, the middle three by RacerX.

To retarget it to a new system, the user supplies a table specifying functions used to acquire and release locks as well as those that disable and enable interrupts (some might do both). These functions have a set of attributes attached that specify whether they: spin, block, or are "try locks" that return an error code rather than block, as well as whether the locks are recursive, are semaphores, or are read locks. RacerX uses this information to determine the effects of acquiring or releasing a given lock.

In addition, users may optionally provide an annotator routine that marks whether routines are single-threaded, multi-threaded, or interrupt handlers. These are used by the race detector to determine whether accesses need locks. Often systems have naming conventions that can be exploited during this process. This approach has a significant advantage over traditional annotations in that it only requires a small, fixed cost to get results.

In the end, RacerX annotation overhead is modest — less than 100 lines of annotations for millions of lines of checked code. For example, Linux needs 18 lines of lock annotations and 31 lines of function annotations; FreeBSD needs 30 and 36 lines; and System X needs 50 and 52 lines.

The extraction phase iterates over each file in the checked system and extracts a control flow graph (CFG) that is stored in a file. The CFG contains all function calls, uses of global variables, uses of parameter pointer variables and, optionally, uses of all local variables. Any concurrency operation — lock, unlock, interrupt disable and enable – is annotated. The CFG also includes the symbolic information for these objects, such as their names and types, whether a variable access is a read or write, whether a variable is a parameter, whether a function or variable is static, the line number of the enclosing statement, etc. To support flow-sensitive analysis, each statement has a set of pointers to all statements that immediately follow it in the program text.

The analysis phase reads the emitted CFG files for the entire system into memory, constructs a linked whole-system CFG, and then traverses it checking for deadlocks or races. Functions are linked first within their containing file and then globally. The resultant graph will be cyclic if the code contains recursion or loops. The CFG potentially has many different roots, which are functions that have no callers. A typical OS will have many roots, one for each system call. [1]

Given the set of roots, the analysis phase iterates over each and does a flow-sensitive, depth-first, interprocedural traversal of the CFG, tracking the set of locks held at any point. This lockset is similar to that used previously in Eraser [29] and Warlock [30]. At each program statement, the race detection or deadlock checkers are passed the current statement, the current lockset, and other information, which they use to emit error messages and statistics used during the final inspection phase. Since there may be exponentially many paths through this graph, we cache analysis results so that we do not have to re-analyze parts of the CFG that we have reached with the same lockset.

The final phase, inspection, consumes the analysis results and post-processes them to compute ranking information for error messages. It then presents these to the user for manual inspection. Ranking sorts messages based on two features: (1) the likelihood of being a false positive, and (2) the difficulty of inspection. A great deal of our effort has been absorbed by the need for clever ranking. In contrast, the analysis has proven relatively straightforward.

2.1 The systems we check

We checked three systems. The first two are Linux version 2.5.62 and FreeBSD version 5.1 (note we do not check FreeBSD for races). These are large systems written by many different programmers and as such is a good test that our approach can handle different coding and design styles.

We also checked a large commercial operating system (roughly 500K lines of code), which we call "System X." Its development environment differs radically from that of Linux and FreeBSD. System X undergoes regular, uniform testing by a dedicated QA department, the development group is relatively tight-knit and fits within a single building, and all code undergoes a peer review process. While the core Linux kernel undergoes a similar review process (albeit more spatially separated) the bulk of the OS is developed and tested more haphazardly. Using the results from systems developed under such vastly different conditions gives a feeling for the generality of the approach.

We crudely calibrated the quality of System X by running two static checkers from [20]. These flagged uses of freed memory and not releasing an acquired lock. The free checker's error rate was .25% (four free errors out of 1600 checks) and the lock checker's was .4% (two lock errors out of 500 checks). Linux had slightly higher error rates at .3% and .46% respectively.

There was also an unfortunate, practical reason to check a commercial system: while it is simple to determine that a shared access occurs without a lock, it is *extremely* difficult in general to take a large unknown system and understand if an access can actually cause incorrect behavior. Doing so requires reasoning about code invariants and actual interleavings (rather than potential ones), both of which are often undocumented. As a result, a single race condition report can easily take tens of minutes to diagnose. Even at the end it may not be possible to determine if the report is actually an error. In contrast, other types of errors found with static analysis can take seconds to diagnose (e.g., uses of freed variables, not releasing acquired locks). By working with paid developers it was much easier to get confirma-

[1] Note that unreachable functions will also be roots; the race detection throws away static functions with no callers, since they often have many race conditions when detached from their intended caller.

tion of race condition errors; race reports to Linux tend to go unconfirmed, perhaps because the maintainers had the same difficulties in reasoning about them as we did.

3. LOCKSET ANALYSIS

This section describes our generic lockset algorithm, which RacerX uses to detect both deadlocks and races. We compute locksets at all program points using a top-down, flow- and context-sensitive, interprocedural analysis. Top-down because it starts from the root of each call graph and does a depth first search (DFS) traversal down the control flow graph (CFG). Flow-sensitive because we analyze the effects of each path, rather than (say) conflating them at join points. It is context-sensitive because it analyzes the lockset at each actual callsite.

Conceptually, the analysis is simple — a DFS graph traversal that (1) adds and removes locks as needed and (2) calls the race and deadlock checkers on each statement in the flow graph. We cache analysis results at both the statement and function level. Caching works because the analysis is deterministic — two executions that both start from statement s within the CFG with the same lockset l will always produce the same result.

We cache the locksets that have reached each statement in the CFG using a *statement cache*. If we reach statement s with lockset l and l is in s's statement cache, we stop analyzing the current path and backtrack to the last join in the CFG. Otherwise we add l to s's cache and continue.

We memoize the effect of each function f using a *summary cache*. This cache summarizes the effects of each function by recording for each lockset l that entered f the set of locksets $(l_1, \ldots, l_n)$ that was produced (i.e., the union of all locksets that reached each exit point in f). If we reach function f with lockset l and l is in the summary cache, we do not analyze f again, but instead skip over it. Otherwise, we do analyze it, record the locksets it produced $(l_1, \ldots, l_n)$, and add $l \rightarrow (l_1, \ldots, l_n)$ to f's summary cache. In either case we continue the analysis forward within the calling function n different times, once for each lockset $(l_1, \ldots, l_n)$.

The exit locksets include all effects of the function and any functions that it calls. Since the analysis is flow sensitive, a function could produce an exponential number of locksets: one for each different control path interleaving that starts from its entry statement and reaches its exit statements, including all paths in all functions that it calls. In practice their effects are more modest. Most functions either release all locks they acquire or avoid acquiring them at all. Thus, a function that only calls other such functions will produce the same single lockset with which it was entered. Functions that do more interesting locking tend to produce a few different locksets at most.

The core analysis algorithm is given in Figure 1. It has three main parts:

1. `traverse_cfg`, which iterates over all roots in the flow graph and calls `traverse_fn`.
2. `traverse_fn`, which takes a function f and lockset l and analyses it. It first checks to see if the lockset is in the function's summary cache (lines 12-14). If so, it returns the set of all locksets that have reached the exit points of `fn` when analyzed starting with lockset `ls`. Otherwise it processes the function. If it is involved in a recursive call, it breaks the recursion and returns. Otherwise, it marks that it is processing this function (line 19), and then calls `traverse_stmts` on the `fn`'s entry statement. This routine does a DFS traversal of all successors and returns the union of all locksets that reach exit points. Its result is added to `fn`'s summary cache (line 27) and returned (line 28).
3. `traverse_stmt` applies lockset `ls` to statement `s`. As with `traverse_fn` it also does caching, but does not need to track the summary — any cache hit will have been visited by a traversal that did reach an exit point (and produced an exit lockset). If we have not seen this lockset before, we add it to the statement cache and continue. If we have reached an exit statement we return the current lockset (line 43-44). The summary for this path says that if we start with `entry_ls` we will produce `ls`. The code then adds or removes locksets as needed. Lines 51-65 do the bulk of the work. We first build an initial worklist — if the statement `s` is a resolved function call this list contains the exit locksets returned by `traverse_fn`; otherwise it is just the current lockset. We then iterate over each lockset in the worklist, recursively calling `traverse_stmt` on each of `s`'s children. We then return the union of all locksets returned by these traversals.

It may not be obvious why we need to track the initial entry lockset `entry_ls` and store it in the cache (line 36). It is required because different entry locksets can converge to the same lockset within a function. If this convergence happened and we did not record the entry lockset in the cache along with the current lockset, a cache hit in the midst of the function could terminate the computation before the final exit locksets have been found. Adding the entry lockset to the cache prevents such masking.

The basic algorithm is modeled on the analysis frameworks of ESP [11] and the MC system [20], both of which are based in part on RHS [28]. Our implementation is simpler since we specialized it to lockset analysis rather than supporting arbitrary extensions. The analysis is also less sophisticated in that if we enter a function with a new lockset, we compute all exit locksets. A more advanced system would do relaxation computations that would allow it to use the exit locksets produced by different entry locksets. Despite this limitation, the analysis is fast overall.

Practical issues. Locksets are interned in a hash table so that we can compare them using pointers. Further, note that only locking operations cause the lockset to grow and shrink. I.e., if a lock name contained in a lockset goes out of scope, the lockset does not change — the fact that a storage location has a value indicating it is locked persists independently of whether this location has a name in the current scope.

For error reporting we obviously need a backtrace of the current callchain along with any interesting events that happened on it (such as lock acquisitions). Getting the trace of all functions above the current call is easy. Unfortunately, we also need a trace of all the events that led to the current lockset, including any calls into children functions that had some effect. Further, since we skip functions that hit in the summary cache, we also have to track the backtrace to all locksets in the summary cache. When reporting an error, we traverse back along the current trace, splicing in summary cache traces at every skipped callsite. Because longer traces are harder to reason about, we pick the smallest trace for

```
// Apply analysis to all roots
void traverse_cfg(roots)
    foreach r in roots
       traverse_fn(r, {});
end

// Compute and return the set of locksets produced
// when entering function fn with lockset ls.  For
// speed we cache a summary.
set of locksets traverse_fn(fn, ls)
  // check fn's summary cache first
  foreach edge x in fn->cache
     if(x->entry_lockset == ls)
          return x->exit_locksets;

  // break recursion arbitrarily
  if(fn->on_stack_p)
     return {};
  fn->on_stack_p = 1;
     // make a new edge.
     x = new edge;
     x->entry_lockset = lockset;
     x->exit_locksets = traverse_stmts(fn->entry,ls,ls);
  fn->on_stack_p = 0;

  // add to summary cache.
  fn->cache = fn->cache U x;
  return x->exit_locksets;
end

// DFS traversal of all statements in fn.  Computes
// the set of locksets produced on exit from each path
// given that we entered fn with lockset entry_ls.
set of locksets traverse_stmts(fn, s, entry_ls, ls)
   // for speed, don't analyze if we've seen already.
   if((entry_ls, ls) in s->cache)
      return {};
   // add to statement cache.  note we need entry_ls
   // here!
   s->cache = s->cache U (entry_ls, ls);

   // summary for this path is: (entry_ls->ls)
   if(s is end-of-path)
         return ls;

   if s is lock
       ls = add_lock(ls, s);
   else if s is unlock
       ls = remove_lock(ls, s);

   // if call is not resolved, process lockset.
   if s is not resolved call
       worklist = { ls };
   // get the set of locksets returned by
   // function when entered with ls.
   else
       worklist = traverse_fn(s->fn, ls);

   // process all locksets in worklist, computed
   // computing set of exit locksets produced by
   // statement children
   summ = {};
   foreach l in worklist
       foreach k in s->kids
          summ = summ U traverse_stmts(fn,k,entry_ls,l);
   return sum;
end
```

Figure 1: Pseudo-code for interprocedural lockset algorithm.

functions that have multiple paths leading to the same exit lockset.

OS code can transfer control between two points without requiring an explicit code path between them. For example, most OSes provide a way to copy data from the user to kernel space and vice versa (e.g., "`copyin`" and "`copyout`" on BSD). If the memory they access is not in core, these routines will take a page fault, implicitly transferring control to the page fault handler. Thus, every callsite that invokes these functions has an implicit call to the page fault handling code. Since page fault handling typically involves a heavy degree of locking it is important not to miss this transfer.

Similarly, there are implicit ordering dependencies between the main bodies of system calls. For example, `open` must occur before `read` or `write` which cannot occur after `close`. [2] Again, compilers cannot see these dependencies.

Fortunately, there is a simple solution. We let the user write "stub" portions of code that encode these dependencies. These are then compiled as normal code, and used to form any necessary edges. For example, the following stub tells RacerX that `copyin` calls `do_page_fault`:

```
int copyin(void *dst, void *src, size_t len) {
     do_page_fault();
 }
```

For the ordering constraint above, we would write code that calls system calls in the orders that we want them analyzed.

Approximations. Our analysis has several important limitations. First, we do not do alias analysis. We crudely represent local and parameter pointer variables by their type name rather than their variable name. For example, a parameter `foo` that is a pointer to a structure of type `bar` will be named "`local:struct bar`." This approximation is mostly conservative in that it can cause false positives for our deadlock detector but will not lead to missed errors. However, it may lead to a lower ranking for errors that involve such pointers than would otherwise be warranted. We plan to use more sophisticated pointer analysis in the future.

Second, we do only simple function pointer resolution. We record all functions ever assigned to a function pointer of a given type (either with an explicit assignment or using a static initialization) and, at each callsite, assume that all of the functions could be invoked. We will not correctly resolve functions that are passed as arguments to a routine that then assigns them to a function pointer. Despite this limit, we resolve most of the function pointers used in practice.

Finally, we have one main speed problem, caused by the fact that OS code tends to have several widely-used functions which can potentially invoke large portions of the OS. For example, kernel memory allocators (such as Linux's `kmalloc`) can invoke an enormous amount of code if memory is low and paging gets triggered, including non-trivial portions of the virtual memory system, file system, and various drivers. This large reach causes problems if the function is called in many different locations with different locks held, since each different entry lockset causes us to reanalyze the function. We take a simple approach to this problem: if the number of distinct entry locksets in a function's summary cache exceeds a fixed limit (100) we skip the function. (This is similar in spirit to PREfix's truncation of path visits [4].)

[2] Of course the initial portion of these routines can run in any order, since the user can call them with impunity; however, out of order calls will fail.

While not entirely satisfactory, this approach does not seem to be that important sources of false negatives. For example, increasing it yields few additional checks. Additionally, in our experience, there is little real interaction between these functions and their callers — code with many callers tends to implement core kernel functionality and rarely calls back into its calling context.

4. DEADLOCK CHECKING OVERVIEW

This section describes the core deadlock checking algorithm and the ranking scheme used to order flagged errors. Deadlock checking has the unfortunate problem that a single mistake can cause an avalanche of false reports. As a result, the bulk of our ingenuity has gone into developing techniques to reduce false positives, which we describe in the subsequent two sections.

4.1 Computing locking cycles

The deadlock detector works in two passes: (1) constraint extraction, which extracts all locking constraints and (2) constraint solving, which does a transitive closure flagging cycles. We discuss each below.

Constraint extraction: is called by the generic analysis engine on every lock acquisition and iterates over every lock in the current lockset, emitting the ordering constraint produced by the current acquisition. For example, if lock a is in the current lockset and lock b has just been acquired, the constraint "$a \rightarrow b$" will be emitted, along with a trace of how to get from the acquisition of a to that of b that includes the number of conditionals, function calls, and degree of aliasing involved.

The most difficult part of constraint extraction is guarding against false constraints caused by invalid locksets. Eliminating false positives consists solely of techniques to eliminate false constraints.

Constraint solving: reads in the emitted locking dependency constraints and computes the transitive closure of all dependencies. It records the shortest path between any cyclic lock dependency, ranks the results, and displays them to the user for inspection. The algorithm computes deadlocks involving $2, \ldots, n$ threads, where n is a user-specified threshold.

As an example, given the constraints $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow a$, the algorithm derives the constraints $a \rightarrow c$, $b \rightarrow a$, $c \rightarrow b$, etc. It will then flag the three-thread cyclic dependency between $a \rightarrow c$ and $c \rightarrow a$ that arises if thread 1 holds a and waits on b, thread 2 holds b and waits on c, and thread 3 holds c and waits on a.

While the core transitive closure algorithm is simple, the need for intelligible error messages requires a fair amount of boilerplate, several times the amount of code needed to do the actual check. We have found this a common problem with program checking tools — detecting an error is often simple, most of the complexity arises in articulating why the tool believes it is error.

4.2 Ranking

Since inspecting deadlock errors can take tens of minutes, effective error ranking is very important. At a high level, ranking error messages consists of sorting them based on the locking constraints that led to them both by how easy these constraints are to inspect, and by how likely they are to be wrong. We rank based on three criteria:

```
ERROR: 2 thread global-global deadlock.
   <rtc_lock>-><rtc_task_lock> occurred 1 time
   <rtc_task_lock>-><rtc_lock> occurred 1 time

<rtc_lock>-><rtc_task_lock> =
  depth = 1:
      linux-2.5.62/drivers/char/rtc.c:rtc_register:723
          ->rtc_register:728

  int rtc_register(rtc_task_t *task) {
    if (task == NULL || task->func == NULL)
        return -EINVAL;
    spin_lock_irq(&rtc_lock);
    if (rtc_status & RTC_IS_OPEN) {
           spin_unlock_irq(&rtc_lock);
           return -EBUSY;
    }
    spin_lock(&rtc_task_lock);
    if (rtc_callback) {
           spin_unlock(&rtc_task_lock);
           spin_unlock_irq(&rtc_lock);
           return -EBUSY;

<rtc_task_lock>-><rtc_lock> =
  depth = 1:
     linux-2.5.62/drivers/char/rtc.c:rtc_unregister:749
        ->rtc.c:rtc_unregister:755

  int rtc_unregister(rtc_task_t *task) {
     spin_lock_irq(&rtc_task_lock);
     if (rtc_callback != task) {
           spin_unlock_irq(&rtc_task_lock);
           return -ENXIO;
     }
     rtc_callback = NULL;
     spin_lock(&rtc_lock);
```

Figure 2: Simple deadlock between two global locks.

1. Whether the locks involved are local or global. Global lock errors are preferred over local ones, since local errors could be a result of naming mistakes because we replace non-global lock names with their type.
2. The depth of the call chain and the number of conditionals it spans. In general, short call chains with few conditionals are better than longer ones: there is less chance that an analysis mistake happened, and with each additional branch and call, inspecting errors becomes harder.
3. The number of threads involved. Each constraint in the computed cycle adds another thread. Errors with fewer threads are preferred to errors involving many threads because they tend to be: (1) significantly easier to reason about, (2) more likely to happen, and, importantly, (3) require touching far less code to get fixed.

We use these ranking criteria hierarchically sort error messages. We first divide errors into classes based on how many threads were involved, and place the classes in ascending order. We then further subdivide each of these classes based on the number of non-global locks they depend on, and arrange these sub-classes in ascending order. At the end, the first class will contain all errors involving two threads and two global locks, the next class all errors involving two threads and one local lock, and so forth. Finally, we sort the errors within each class by the number of conditionals and call chain depth involved in the shortest lock constraint (each call counts the same as three conditionals).

Figure 2 gives the highest ranked error we have found.

It involves two global locks, both acquired within the same function. The output shown is the raw output from our tool (though the code example has been reformatted). It states that this is a two thread deadlock involving the two global locks rtc_lock and rtc_task_lock, which are acquired in the order rtc_lock→rtc_task_lock once and the order rtc_task_lock→rtc_lock once. Both locks are only involved in this error and the call chains for both have a depth of one — i.e., both acquisitions happen within the same function. The error occurs if thread A enters rtc_register, acquires rtc_lock and thread B enters rtc_unregister and acquires rtc_task_lock. Both threads will then deadlock while they attempt to acquire the other's lock.

Since both routines occur next to each other in the source code, it appears that the programmer might have even done this ordering deliberately, perhaps because it mirrored the fact that register is reversed by unregister and so the locking order should be reversed. Numerous of our other Linux errors followed a similar pattern, where a device registration used exactly the opposite lock ordering of the unregister operation.

5. INCREASING ANALYSIS ACCURACY

This section describes techniques that eliminate two significant sources of false lock dependencies caused by (1) semaphores used to enforce scheduling dependencies (rather than mutual exclusion) and by (2) "release-on-block" locks.

5.1 The rendezvous problem

One of the largest sources of false positives come from the fact that semaphores have two conflated uses: (1) binary semaphores used as locks to provide mutual exclusion or (2) signal-wait semaphores used to implement scheduling dependencies. In the latter case, one thread will wait on a semaphore using the "down" operation (an atomic decrement that sleeps if the counter is zero) until another thread indicates it should proceed using "up" (an atomic increment that may wake a sleeping thread). If we naively always treat a signaling semaphore as a lock, we would flag invalid lockset cycles. Consider the following code:

```
Producer               Consumer
up(s); // signal ready lock(l);
                       down(s); // wait for result
                       unlock(l);
                       ...
                       lock(l);
```

If the semaphore s were indeed a lock, then the consumer's code has a cyclic dependency that can cause a deadlock (similar to the deadlock described in Figure 5). The deadlock occurs if two threads A and B can both run the consumer's code with the following interleaving: (1) A acquires l, acquires s, releases l and (2) B acquires l. Both threads will deadlock waiting on the other's lock.

For similar reasons, signal-wait semaphores can cause false negatives for the race condition checker described in Section 8. In the above code, since the call to "down(s)" is not followed by a matching "up" call, RacerX would believe that the "lock" s is held on every subsequent path, when in fact s provides no mutual exclusion at all.

We use *belief analysis* [14] to distinguish when a semaphore is being used as a lock as opposed to a signal-wait semaphore. The former we track as we do other locks, the latter we ignore. Signal-wait semaphores have two behavioral patterns: (1) they are almost never paired and (2) they have roughly as many unmatched up(s) calls as down(s) calls. In contrast, valid locks have exactly the opposite behavior: (1) they are almost always paired and (2) when not paired, they have many more unmatched lock calls than unlock calls (the most common unpairing is a lock with no matching unlock).

Thus, we have a simple classification problem: given a semaphore s does it behave more like a lock or like a scheduling semaphore? We answer this question by first measuring how often the large supply of known, non-semaphore locks satisfy these tests. We then measure how often s does and use statistical analysis to compute the probability that s belongs to a population similar to the known locks. More precisely, we use the following four-step algorithm:

1. Calculate how often true locks satisfy these two behavioral patterns. We take the entire set of known locks and count the number of lock acquisitions a_{total}, lock releases r_{total}, and unlock errors err_{total}. We then calculate the average ("expected") ratio of releases to acquisitions $t1_{lock} = r_{total}/a_{total}$ and the expected ratio of unlock errors to unlock callsites $t2_{lock} = err_{total}/r_{total}$.

2. Calculate similar values for s: $t1_s = r_s/a_s$ and $t2_s = err_s/r_s$.

3. Use *hypothesis testing* [17] to calculate the probability p we would observe $t1_s$ and $t2_s$ (or a more unlikely event) if s belongs to a population with a true measurement of $t1_{lock}$ and $t2_{lock}$. Details are in [17]. We treat the tests as independent, allowing us to compute p-values p_1 and p_2 for each test in isolation and derive p by multiplying the results: $p = p_1 \cdot p_2$.

4. Discard semaphores below some probability threshold. In our case we set it to $p = 0.17$, but the exact value does not matter much, given the extremes exhibited by either population.

This approach closely follows that of [14]. The main difference is that we extend it to handle multiple tests (this prior work only looked at one).

Table 2 gives a few examples from both extremes, with representative counts. The lack of any release, and a high ratio of unlock errors to acquisitions are good indicators that the semaphore is not being used as a lock. This classification eliminated over twenty false positives out of roughly 40 examined deadlocks from our experiments.

Name	Acq	Rel	Unlock Err
PQFCHBA.TYOBcomplete	5	0	5
event_exit	2	0	9
thread_exit	2	0	1
us_data.dev_semaphore	8	28	2
mm_struct.mmap_sem	141	208	2

Table 2: Three signal-wait semaphores and two locking semaphores classified by our analysis. Signal-wait semaphores behave differently than locking semaphores. Note there are more releases than acquisitions when there are more paths after a lock that contain an unlock.

§ 4.2:	Rank errors involving few threads, local pointer locks, conditionals, and function calls over those with many.
§ 5.1:	Automatically detect and suppress dependencies caused by semaphores used to enforce scheduling constraints.
§ 5.2:	Do not generate constraints between release-on-block locks and lock acquisitions that block.
§ 6.1:	Set lockset to empty after encountering a locking error (e.g., a double lock).
§ 6.2:	Inspect errors found with downward-only propagation over those from upward.
§ 6.4:	Rank errors flagged by both upward and downward lockset propagation over those by only one method.
§ 6.3:	When propagating locksets to callers, pick the lockset that occurred on the most exit edges; break ties by taking the smallest lockset.
§ 6.5:	Use unlockset analysis to remove lockset members not released on any subsequent path.

Table 1: Techniques to reduce deadlock false positives. The first two additionally help ease-of-inspection. The remaining techniques primarily focus on eliminating bogus constraints.

```
  Code                Actual semantics
lock_kernel();      lock_kernel();
      // constraint: BKL->lock
spin_lock(lock);    spin_lock(lock);
      // constraint: lock->sem
      // false constraint: BKL->sem
down(sem);          while(down(sem) would block) {
                         // release BKL on block: no
                         // constraint between BKL
                         // and sem.
                         unlock_kernel();
                         schedule(); // reschedule
                         // constraint: BKL->lock.
                         // possible deadlock.
                         lock_kernel();
                    }
```

Figure 3: Code illustrating problems caused by semaphores.

5.2 Release-on-block (ROB) lock semantics

One unanticipated source of false positives is that many older operating systems such as FreeBSD and Linux use global, coarse-grained locks that have "release-on-block" (ROB) semantics. These are a legacy left from their similar evolution from single-processor (mostly) single-threaded kernels to (somewhat) multi-threaded kernels running on multiprocessors. Initially, their main source of concurrency was interaction between the kernel and device interrupts, which they managed by disabling interrupts. They then introduced multiple kernel threads, managed using a single kernel lock, essentially turning the OS into a single, large monitor. Linux does so using the "big kernel lock" (BKL) released and acquired with calls to `lock_kernel()` and `unlock_kernel()`. FreeBSD similarly has the "`Giant`" lock managed with standard locking calls. Because there was only one such lock, any kernel thread that went to sleep holding it would deadlock the system. Thus, as with normal monitors [23] it was released and then reacquired after blocking.

Since then, these systems have moved towards more fine-grained locking, but their single ROB lock remains active and intermixed with the newer additions. Unfortunately, this intermixing causes two problems. First, for the programmer, it makes it easy to cause deadlocks when a thread acquires the single-kernel lock, acquires another lock, and then sleeps. Second, for the tool, it causes many bogus constraints to be emitted.

Figure 3 gives example code illustrating both of these problems. The code acquires the BKL (using `lock_kernel()`), acquires `lock` and then attempts to acquire `sem`. If the acquisition of `sem` would block, the locking code will release the BKL and switch to another thread. When the blocked thread is subsequently rescheduled, it will attempt to reacquire the BKL and then, if that succeeds, attempt to acquire `sem`. Because of ROB semantics, this final acquisition does not form a dependency between BKL and `sem` and cannot cause a deadlock: another thread holding `sem` while attempting to acquire the BKL will eventually succeed since the thread that holds the BKL will keep releasing it when it fails to acquire `sem` [26]. Thus, we do not emit a constraint when a thread holds a ROB lock and attempts to acquire a lock that blocks (rather than spins). This change eliminated a large number of false positives.

While ROB semantics prevent deadlocks with blocking locks, they make deadlocks with spinlocks more difficult to avoid. The release and acquisition of the BKL in the above code can cause a deadlock if another thread holds the BKL while attempting to acquire `lock`. In general, such a locking cycle arises whenever a thread has acquired a ROB lock, acquired a spinlock, and then blocks. Since many operations cause rescheduling — blocking memory allocation, kernel page-faults on user data, explicit sleeps — correctly using the single ROB lock with other spinlocks is difficult.

While these types of deadlocks are hard for the programmer to avoid, they are especially easy for the tool to flag: emit an error whenever a lockset that has a dependency from a ROB lock to a spinlock reaches a potentially blocking operation. Although this check is not difficult to build, we instead check a more stringent rule used by Linux, FreeBSD, and OpenBSD to prevent such deadlocks: do not call a blocking operation with a spinlock held or interrupts disabled. A 50-line extension to our deadlock checker flags the shortest path between such acquisitions and any blocking operation. It finds hundreds of errors in the latest version of Linux.

Figure 4 gives a representative example where the routine `atm_ioctl` calls `put_user` with the `atm_dev_lock` spinlock held. The routine `put_user` puts data at a user-specified address and can block if a page fault occurs. This routine is a good a good example of the value of automatic checking — `atm_ioctl` has over 20 such errors. These errors are potential security holes since the user can cause a page fault at will. We do not discuss these results further: our checker is conceptually no different than that described by in previous work [20], though it is a completely different implementation.

```
//linux-2.5.62/net/atm/common.c:556:atm_ioctl:ERROR:BLOCK
// calling blocking function <put_user> w/ lock held!
     spin_lock (&atm_dev_lock);
     vcc = ATM_SD(sock);
     switch (cmd) {
     case SIOCOUTQ:
        ....
       ret_val =  put_user(...); // ERROR: can block.
```

Figure 4: Example security deadlock: a call to put_user (which can block) with a spinlock held.

6. HANDLING LOCKSET MISTAKES

Deadlock false positives are caused by false locking constraints, most from analysis mistakes. This section describes four techniques we use to mitigate the effects of such mistakes.

The single most significant cause of invalid locksets are intra- and interprocedural false paths. Almost all false constraints arise from a data-dependent lock release, either parameter-controlled locking or, more commonly, correlated branches such as the following:

```
void foo(int x) {
  if(x)
      lock(l);
  ...
  if(x)
      unlock(l);
}
```

Without path-sensitive analysis (which we do not do) RacerX will believe there are four paths through `foo`, one of which acquires lock `l`, but does not release it. On this invalid path, the analysis will exit function `foo` with an invalid lockset containing `l`, which will promptly begin generating spurious constraints. Our interprocedural analysis makes these mistakes worse because it will aggressively push such bogus locksets over as large an extent of code as possible.

While we intend to add path sensitivity to reduce the effects of this problem, the problem is undecidable in general and often difficult in practice. Instead we have used our experience examining false error reports to design simple, novel propagation techniques that minimize the propagation of invalid locksets, often limiting them to the actual function that caused the problem. Importantly, these techniques should work just as well with stronger analysis and other checking system.

6.1 Cutting off lock-error paths

We prune the lockset on paths that contain a locking error — double lock acquisitions, lock releases without a prior acquisition, or failure to release an acquired lock. Either the locking error is valid, in which case the system is in an inconsistent state and subsequent reports are neither surprising nor trustworthy, or the analysis has made a mistake, which will have similar results. In either case we want to suppress the effects of these mistakes. In the case of double locking and releases of unacquired locks, when run in deadlock-mode RacerX sets the current lockset to empty, thereby eliminating any constraints that straddle the locking error. [3] (In lock-error mode, it will emit a message so the user can inspect such locking errors.) Initially, we detected the third error, unreleased locks, when we finished processing a given root. We emitted an error for each held lock and demoted constraints emitted on that path. This case has since been subsumed by unlockset analysis (§ 6.5).

In a sense, lock acquisitions and releases can be viewed as crude programmer-supplied annotations indicating which code paths they believe must be taken. Paths that contain matching lock-unlock annotation pairs are likely valid and should be trusted; those with mismatches likely invalid and should not be.

6.2 Downward-only lockset propagation

A significant source of false positives for upward propagation occur when it falsely believes that a lock is held on function exit when it is not. Obviously, if we only propagate locksets downward from caller to callee but never upward back to the caller we eliminate this problem. Errors from this method tend to be the most reliable. Further, in our experience, they are also easier to reason about since they involve looking at one related call chain rather than the up and down progress through various subsystems and their abstractions.

We almost always use downward-only propagation to get to the first round of deadlock errors. However, this approach is vulnerable to false negatives, since it does not propagate the effects of locking "wrapper functions" (whose sole purpose is to acquire or release locks) up to their callers. Thus, after inspecting these results, we then do a second pass using upward propagation.

6.3 Selecting the right summary

Not all function summaries are equally trustworthy. An effective way to reduce false constraints is to only propagate the most plausible ones. We describe two different summary propagation strategies to isolate the effects of bad locksets.

Majority summary selection: after looking at around fifty locking errors similar to the code example at the beginning of this section we had the obvious-in-hindsight realization that since locking mistakes tend to occur on a minority of paths — most paths are correct, a few are wrong — this pattern can be exploited to select the most reasonable lockset from our summary cache. Rather than following all locksets a function call generates we instead take the one produced by the largest number of exit points within the function. Our assumption is that the lockset that reaches the most different exit edges within a function is the one that most accurately describes the actual locking interface of the function.

Minimum-size summary selection: almost all false positives come from paths that fail to release locks — such errors will cause the lockset on these paths to be one larger in size than on well-formed paths. This leads to the second realization that the effects of these errors go away if we simply select the smallest lockset exiting a given function. This insight can also be added to the majority summary selection scheme above as a means to break ties.

While summary selection is simple, it makes a significant difference in practice. We expect it will handle similar problems we have encountered in other checkers.

[3] Naively, we might instead chose to stop following the path. While this decision works if the path is truly infeasible, it may not work when the locking error was caused by an analysis mistake — if all paths to this program point have such mistakes, we will abort all of them, leaving the subsequent code path unexplored.

6.4 Constraint intersection

Downward-only and upward lockset propagation have are vulnerable to different problems: downward propagation to wrapper functions, upward propagation to invalid summaries that fool their selection criteria. We can exploit these different weaknesses to determine the most reliable lock dependencies.

In general, the more different ways to compute the same result the more confidence one has in its correctness. Locking dependencies are no different. If both methods state that the dependency $a \rightarrow b$ exists, we have more confidence in it than if $a \rightarrow b$ was produced by just one of the methods. We use this to rank different errors based on the number of different methods that agreed on the involved dependencies.

In practice, we first examine errors based on constraints derived from both methods, then the errors from downward propagation, then the errors from upward.

6.5 Unlockset analysis

Unlockset analysis is a simple yet effective method for removing invalid locks from a lockset. We developed it after attempting to find deadlocks in FreeBSD. While the techniques described so far were sufficient to handle Linux and System X, they left a large number of false positives on FreeBSD, mainly due its more complex control flow determining lock acquisition and release.

Unlockset analysis allowed us to eliminate by far the majority of the false constraints this complexity caused. It came from three related observations. First, when emitting a dependency $a \rightarrow b$, almost all false positives are due to mistakes about a. Intuitively, this makes sense — propagating a from its acquisition site to the acquisition site of b requires making at least several, and potentially many analysis decisions correctly. In contrast, we are naturally protected against most mistakes about b since the dependency $a \rightarrow b$ is emitted immediately at b's acquisition site, with little opportunity for a mistake. Thus, a good way to cut down on false constraints is to only leave those locks in the lockset which we have a high degree of confidence that they are acquired.

Second, after examining the false positives caused by false constraints $a \rightarrow b$, we noticed that in almost all cases, the code after the acquisition of b had no unlock of a. I.e., most false constraints are caused when an acquired lock "goes too far" and is propagated past all of its releases.

Third, as a result of the prior observations, we had unconsciously evolved a pattern of inspecting deadlock errors caused by dependencies where the lock a was explicitly released after the second lock b was acquired. These errors are the easiest to inspect, since they demonstrate that the lock a was held past the acquisition of b. The code for `rtc_register` in Figure 2 is a good example. First it acquires `rtc_lock`, then `rtc_task_lock`, then it subsequently releases `rtc_lock`, strongly implying it was held. Thus, when ranking errors we would like to emit those first that are explicitly paired with a unlock.

Unlockset analysis exploits all of these observations. Intuitively it is simple: at program statement s, remove any lock l in the current lockset if there exists no successor statement s' reachable from s that contains an unlock of l. If l reaches no unlock after statement s on any subsequent path then it is almost certain our analysis has made a mistake. Conversely, if it does reach an unlock then it becomes more likely that the lock is indeed held. The nice thing about this technique is that it tightly fits the lockset to the set of locations that the analysis seems able to handle. For example, consider the locksets obtained in the code given at the beginning of this section if we drop all locks that will not reach a subsequent release:

```
1: void foo(int x) {  // unlockset
2:   if(x)            {   }
3:       lock(l);     { l }
4:   ...              { l }
5:   if(x)            { l }
6:       unlock(l);   { l }
7: }                  {   }
```

As with the original analysis, the lockset will contain `l` at lines 3, 4, and 5. However, unlike the original analysis, `l` is eliminated on the false path of the check at line 5 since it reaches no subsequent release. The lockset is exactly what RacerX would compute if it could suppress the infeasible path. The nice thing it that unlockset analysis is complementary to traditional analysis.

Unlockset analysis runs in two passes. Its implementation is similar to that of liveness analysis, which is used to determine if there is a path from statement s that can reach a use of variable v. The first pass is a backwards analysis that runs before deadlock checking. For each statement s it computes s's *statement unlockset*, which is the set of unlocks reachable using a downward traversal starting from s. We compute statement unlocksets using a backwards pass from the leaves of the CFG reached during lockset analysis back up to the root(s). It is essentially the inverse of lockset analysis: as unlocks are encountered, they are added to the unlockset, as locks are encountered, the unlock they reverse is removed. A statement's unlockset is the union of all of its children's unlocksets, plus the effect of any lock or unlock that it itself contains. Additionally, if statement s is a call to resolved function f, we add the unlockset of f's entry block to s's. Similar to lockset analysis, we do not insert unlocks for which there was no preceding lock. Each statement persistently records its unlockset for use by the second pass.

The second pass is a forward analysis that runs during deadlock checking and computes the dynamic *call unlockset*, which is specific to each call chain. The call unlockset is the union of the statement unlocksets of each callsite in the current call chain. For example, assume we are analyzing function h reached by the call chain $f \rightarrow g \rightarrow h$. The call unlockset used to analyze h is computed by taking the union of the statement unlocksets at the callsite of (1) f's call to g and (2) g's call to h. We use the call unlockset to support context sensitivity. I.e., different callers $c1$ and $c2$ may release different locks after the call to h returns — these releases should just be added to the unlockset of each callsite and this call to h rather than to all callsites. In general, we compute the call unlockset as follows. Before processing a root we initialize the current call unlockset to the empty set. When the analysis follows a resolved function call to h at statement s in function g, we calculate the new call unlockset used to analyze h by taking the union of the original unlockset used in g and s's statement unlockset.

Deadlock checking eliminates invalid locks in the lockset l at statement s by intersecting l with s's statement unlockset and the current call unlockset.

System	Confirmed	Unconfirmed	False
System X	2	3	7
Linux 2.5.62	4	8	6
FreeBSD	2	3	6

Table 3: Deadlock bugs in System X, Linux, and FreeBSD. We separate confirmed from unconfirmed bugs since these messages can be hard to diagnose correctly in someone else's system.

7. DEADLOCK RESULTS

Table 3 summarizes the errors found in System X, Linux, and FreeBSD. We found two confirmed deadlock errors in System X and three unconfirmed. There were seven false positives, three because of a complex check that acquired locks in different orders depending on their global ordering. Initially, there were over 20 false positives, but these were eliminated by the analysis of signal-wait semaphores described in Section 5.1

Four errors in Linux were confirmed and patched. The others looked almost certain. There were six false positives. Since deadlock errors tend to take a while to inspect, we still have over fifty uninspected error reports for Linux.

Finally, there were two confirmed errors in FreeBSD (both fixed), and three unconfirmed, though they seem probable. The six false positives come from complicated locking protocols. A very large number of false positives were eliminated with the unlockset analysis in Section 6.5.

We describe two of the more interesting errors below.

A surprising error: a single code path introduces a cyclic dependency: (1) code acquires `a`, acquires `b`, releases `a`, and (2) reacquires `a`. The last reacquisition introduces a circularity that can cause deadlock if another thread "slips in" and acquires `a` between steps (1) and (2) and then waits on `b`.

Figure 5 gives an example taken from the commercial code System X. Here a thread enters the routine `FindHandle` while holding `scsilock` and acquires `handleArrayLock`. It then calls `Validate`, which can potentially sleep using the routine `CpuSched_Wait`. Unfortunately, `CpuSched_Wait` will release `scsiLock` before blocking and then reacquire it when waking. This sequence creates a locking cycle.

While obvious in hindsight, this was not a type of deadlock we had initially expected. The first error report that showed such a case was greeted with silence for 30 seconds or so, then surprised laughter.

The most complex error: we only inspected a small subset of all deadlocks emitted by the tool. The error in Figure 6 was the most complex. It requires three threads, involves one global and two non-global locks, and one deep call chain. The deadlock occurs in the following case:

1. Thread 1 executes `igmp_timer_expire` in the IPV4 networking code where it acquires `im`→`lock` and then follows a six-level call chain to the routine `inet_select_addr` where it tries to acquire the global lock `inetdev_lock`.

2. Thread 2 executes the routine `inet_select_addr` where it acquires the global lock `inetdev_lock` and tries to get the lock `in_dev`→`lock`.

3. Thread 3 executes the routine `igmp_heard_query` where it acquires `in_dev`→`lock` and then calls `igmp_mod_timer` where it tries to get `im`→`lock`.

```
// Entered holding scsiLock
int FindHandle(int handleID) {
   prevIRQL=SP_LockIRQ(&handleArrayLock,SP_IRQL_KERNEL);
   //find the right handle
   Validate(handle);
   handle->reserved = 1;
   SP_UnlockIRQ(&handleArrayLock, prevIRQL);
   ...
// Entered holding scsiLock->handleArrayLock
int Validate(handle) {
   ASSERT(SP_IsLocked(&scsiLock));
   while (adapter->openInProgress) {
       // BUG: Releases scsilock!
       CpuSched_Wait(&adapter->openInProgress,
                 CPUSCHED_WAIT_SCSI, &scsiLock);
       SP_Lock(&scsiLock);
```

Figure 5: An unexpected deadlock pattern: an acquired lock is released and then reacquired by the same thread, forming a cycle.

This deadlock takes a while to diagnose, even when told exactly what the problem was by a static tool. It is difficult to imagine finding this error without such hand-holding.

8. RACE DETECTION

This section describes the race checker in RacerX. Race detection is significantly harder than deadlock detection. It requires having good answers to the following questions:

1. Is the lockset valid? The problems caused by invalid locksets can be even worse than in the deadlock checker. Most paths have relatively few locking operations. Thus, for deadlock detection, an invalid lockset often not encounter subsequent lock operations, rendering the mistake harmless. In contrast, there are many variable accesses, which invalid locksets will immediately hit.

2. Can the code containing an unprotected access run concurrently with another thread that accesses the same shared state? Many parts of an OS, such as initialization code, are effectively single threaded. Others, may have very restricted forms of concurrency, such as a device driver whose global state is private to a single hardware device, and thus need only manage concurrent accesses made by its interrupt handling code.

3. Does the access actually need to be protected? Many concurrent, unprotected accesses to shared state are perfectly harmless. They may involve reads of data that no other thread will write [29], or simple assignments that are carefully ordered to provide a coherent picture of the state. More difficult, we must differentiate pointers that point to shared state from those that only refer to local state. We must identify calls to functions that require locks (even if we do not have their source code).

Below we give an overview of the basic algorithm and describe the techniques RacerX uses to answer these questions.

8.1 Race detection overview

At a high level, the race checker is called by the lockset analysis on each statement. It uses the current lockset and internal bookkeeping data structures to determine if it should emit an error or not. As with the deadlock detector, given lockset analysis, the checking logic itself is simple: by far the majority of effort goes into ranking error

```
thread 1: <local:ip_mc_list.lock>->inetdev_lock =
  depth = 7:
     // linux-2.5.62/net/ipv4/igmp.c
     void igmp_timer_expire(...) {
        ...
        spin_lock(&im->lock);
        im->tm_running=0;
        ...
        err = igmp_send_report(...);
        ->ip_route_output_key:2149
          ->__ip_route_output_key:2142
            ->ip_route_output_slow:1988
               ->fib_semantics.c:__fib_res_prefsrc:638
                 ->devinet.c:inet_select_addr:759
                              read_lock(&inetdev_lock);
      ...
thread 2: <inetdev_lock-><local:ip_mc_list.lock> =
  depth = 1:
      // linux-2.5.62/net/ipv4/devinet.c
      u32 inet_select_addr(...) {
        ...
        read_lock(&inetdev_lock);
        in_dev = __in_dev_get(dev);
        if (!in_dev)
                goto out_unlock_inetdev;
        read_lock(&in_dev->lock);
      ...
thread 3: <local:in_device.lock>-><local:ip_mc_list.lock>
  depth = 2:
    // linux-2.5.62/net/ipv4/igmp.c
    static void igmp_heard_query(...) {
      ...
      read_lock(&in_dev->lock);
      for (im=in_dev->mc_list; im; im=im->next) {
          ...
          igmp_mod_timer(im, max_delay);
            ->igmp_mod_timer:165
               spin_lock_bh(&im->lock);
```

Figure 6: Complex deadlock involving three threads, non-global locks, and one deep callchain. Since fixed in Linux.

messages. Ranking for race detection is even more important since we must simultaneously guard against mistakes (1) in false paths, (2) in determining if code can run concurrently, and (3) in determining if the access needs protection. The checker can be run in three modes, from least to most precise:

1. Simple checking: only flags global accesses that occur without any lock held (i.e., the lockset is empty when the access occurs). Because it does not depend on resolving lock names correctly or inferring that a pointer can reference shared state, it eliminates several sources of false positives and its results are relatively robust. We always run this mode first.
2. Simple statistical: infers which non-global variables and functions must be protected by some lock.
3. Precise statistical: infers which specific lock protects an access and flags when an access occurs when the lockset does not contain that lock. We run this mode last, since it is the most sensitive to analysis mistakes.

We use a set of heuristics to identify and rank likely races. Since different error messages will satisfy different heuristics, we need a way to compare them. We do so using a scoring function, which is the usual way to map non-numeric attributes to a numeric value suitable for comparison:

1. Each heuristic has a point value associated with it, positive when the heuristic correlates with a race, negative if it does not. Table 4 summarizes these heuristics and their default scores; the following four subsections describe the heuristics in more detail.
2. We compute a score for each error message by summing the point values of each heuristic it satisfies.
3. We sort all messages based on their point value.
4. We then use deterministic ranking (as in Section 4.2) to sort messages with the same point value.

The downside of this approach is that assigning points to heuristics is ad hoc. Unfortunately, since the heuristics do not typically directly dominate one another, there fundamentally does not seem to be an alternative. Fortunately, ranking does not seem overly sensitive to the exact scoring used. Further, the user can customize the scores to values more to their taste, and we could (but do not) use a learning algorithm to fit them more to the data.

The next four subsections describe how we determine: (1) if a lockset is valid, (2) if a context is multithreaded, (3) if an unprotected access is unsafe, and (4) if an access should be protected by a specific lock. We then describe some techniques to detect false negatives and finally results.

8.2 Is the lockset valid?

Deadlock false positives happened when the lockset contained invalid locks. Race detection false positives happen when the lockset does not contain all valid locks. We use the following techniques described in the deadlock checking sections to handle this problem, though some have to be changed to conservatively include locks rather than conservatively omit them. First, we use summary selection (§ 6.3) to decide what locksets to propagate to callers, though we bias towards picking the lockset containing the most entries rather than those with the least. Second, we use intersection ranking (§ 6.4) and rank race conditions found by both downward and upward propagation over those found with upward alone. Finally, we use the automatic classification of scheduling semaphores (§ 5.1) to eliminate false negatives.

8.3 Is code multithreaded?

We have two methods of determining if code is multithreaded: (1) multithreading inference and (2) programmer-written automatic annotators that use system-specific knowledge to mark known single and multithreaded code.

Multithreaded inference: is a simple, automatic technique that uses a form of belief analysis [14] to infer whether a programmer believes code is multithreaded. It relies on the fact that in general programmers do not do spurious locking. Locks are acquired and released to protect something. Thus, any concurrency operation implies that the programmer believes the surrounding code is multithreaded. These operations include locking, as well as calls to library routines that provide atomic operations such as `atomic_add` or `test_and_set`. (From this perspective, concurrency calls can be viewed as carefully inserted annotations specifying that code is multithreaded.)

RacerX marks a function as multithreaded if concurrency operations occur (1) anywhere within its body, or (2) anywhere above it in the current callchain. Note that concurrency operations below it do not necessarily imply that the function itself is multithreaded. For example, it could

§ 8.2 Is the lockset valid?
Rank errors flagged by both downward and upward propagation highest.
Order errors with the same score based on number of conditionals and call chain depth.
Count visits to statement with empty lockset versus not.
+2 if always visited with empty lockset.
+1 if more visits with empty lockset than not.
§ 8.3: Is code multithreaded?
Were there concurrency operations in this function or earlier in callchain?
Rank all `true` cases before `false` ones.
Data shared with interrupt handler.
+2 if written in interrupt handler.
+1 if read by interrupt handler.
Count modifications n on different roots.
+2 if $n > 2$.
+1 if $n > 1$.
§ 8.4 Does X need to be protected?
Count number of times that X was the first, last or only object in a critical section.
+4 If only object > 1 times, +2 if 1 time.
+1 If first object > 0 times.
+1 If last object > 0 times.
Compute z-test statistic based on count of how often protected with any lock versus not protected.
+2 If $z > 2$.
−2 If is non-global and $z < -2$.
Count the number n of unprotected variables in the non-critical section.
+2 If $n > 4$.
+1 If $n > 1$.
Non-atomic updates: writes to > 32-bits or bitfields.
+1.
Access was a write.
+1.

Table 4: Subset of ranking and scoring criteria used to rank race conditions warnings. Ranking computes the total score for each message and then does a descending sort.

be calling library code that always conservatively acquires locks. RacerX computes this information in two passes. First, it walks over all functions, marking them as multithreaded if they do explicit concurrency operations within their body. Second, when doing the normal lockset computation, it tracks if it has hit a known multithreaded function and, if so, adds this annotation to any error emitted.

Programmer-written annotators: As stated in Section 2 clients can iterate over RacerX's representation of the checked system's functions marking them as: (1) single threaded (such as initialization code), (2) functions that should be ignored, (3) multithreaded (such as system call entry points) (4) interrupt handlers, which can run at any point unless interrupts are explicitly disabled, and (4) functions that should be ignored. RacerX skips the first two categories and promotes errors in the latter two. As in the automatic inference, reports originating from multithreaded code paths are more likely than those coming from possibly single-threaded code paths.

Annotators can exploit naming conventions and other system knowledge to label functions. For example, the following is our Linux annotator that marks all functions beginning with "`sys_`" (i.e., system calls) as multithreaded:

```
// client annotator called by RacerX
void mark_entry_routines(struct flist *fl) {
   for(struct fn *f = fn_begin(fl); f; f = fn_next(f))
       if(strncmp(f->name, "sys_", 4) == 0)
            f->multithreaded_p = 1;
}
```

The fact that annotators can automatically classify large numbers of functions based on programmer knowledge has been a big win.

RacerX automatically propagates these annotations to other "equivalent" functions by exploiting the fact that in systems code, functions assigned to the same function pointer typically belong to an equivalence class in that they all implement the same interface (e.g., `open`, `read`, `write`, `close`) [14]. Thus, if a function `f` marked with some annotation ("multithreaded," "single," etc.) is assigned to function pointer `fp`, RacerX automatically propagates its annotation to all other functions assigned to the same function pointer.

8.4 Does X need to be protected?

We take three approaches to answering this question: (1) eliminating accesses unlikely to be dangerous, (2) promoting accesses that have a good chance of being unsafe, and (3) inferring which variables programmers believe must not be accessed without a lock. We describe each below.

Ignored accesses: we ignore variables read but never written, since they require no concurrency control. We optionally demote variables written but rarely or never read since they are often statistics variables that the programmer explicitly tolerates lost updates on. Warnings about them are often unwelcome and can hide more important errors. We avoid flagging races on variables still private to a given thread by stripping out all references to newly allocated data. We avoid flagging common harmless initializations by demoting assignments of 0 and 1 to variables. Finally, we demote errors where data appears to be written only during initialization and only read afterwards. We do so by counting the number of different roots that reach a write to a given variable; variables written only from a single root are likely candidates and their errors demoted.

Non-atomic modifications: We also want to detect accesses that have a high probability of being unsafe so that we can promote them. We have found several heuristics that work well. First, we favor errors that write data over read errors, since there are many more ways that reads can be safe. Second, we rank each error on an unprotected code path (a "non-critical-section") by the total number of other accesses to writable shared state. If there are many variables on this path that could be written by other threads then it is almost certain a thread executing this path will see an inconsistent view of the world. (Figure 7 contains an example.) As a special case, we explicitly flag variables that cannot be read or written atomically such as bit fields and 64-bit variables on 32-bit machines. Unprotected accesses to these variables can result in surprising values, which is rarely the intent.

Programmer beliefs: We rank unprotected accesses especially high if the programmer appears to believe that the variable or routine should be protected. The first effective

```
/* unprotected access to
    vars=[logLevelPtr,
        _loglevel_offset_vmm,
        (*theIOSpace).enabledPassthroughPorts,
        (*theIOSpace).enabledPassthroughWords]
   [nvars=4] [modified=1] [has_locked=1] */
    LOG(2,("IOSpaceEnablePassthrough 0x%x count=%d\n",
          port, theIOSpace->resumeCount));
    theIOSpace->enabledPassthroughPorts = TRUE;
    theIOSpace->enabledPassthroughWords |= (1<<word);
```

Figure 7: Simple race: four unprotected accesses to global data on a path that had previously seen locking. (Note, the first two occur in the `LOG` macro.)

way to infer such beliefs is to realize that the first, last, or only shared data in a critical section are special. As noted above, programmers do not write redundant critical sections. Thus, if a variable or function call is the only piece of potentially shared state within the critical section, then we have strong evidence that the programmer thinks (1) the state should be protected and (2) that the acquired lock does so. Similarly, the first and last pieces of shared state in a critical section are also noteworthy (although to a lesser degree), since programmers often acquire a lock on the first shared state access that must be protected and release it immediately after the last one. (Another way to look at it is that programmers do not make critical sections gratuitously large).

The second way we use statistical analysis to infer if a programmer appears to believe an access should be protected. Oversimplified, it works by assuming all functions and all pointer variables must be protected by a lock. It then counts how many times they are accessed with a lock held versus not. Routines and types the programmer believes should be protected will have a relatively high number of locked uses and few unlocked uses. Identically to [14] (and similarly to § 5.1) we then use the z-test statistic to determine how likely it is that these counts were due to chance (i.e., the programmer protected the variable or function coincidently rather than with intent). The reader does not need to understand the details, other than we use the formula:

$$z = (s/n - p_0)/\sqrt{p_0 \times (1 - p_0)/n}$$

Here s is the number of protected accesses, n the total number of accesses, and p_0 represents how often that we expect a function or variable that does not need to be protected will be protected coincidently. (We set $p_0 = 0.8$, which reflects that such coincidental protections are frequent.) A high positive value of z implies that the number of protected accesses was more than that we would expect from chance.

An example error: Figure 7 shows a race that we give a high ranking to based on the features discussed in this section. The output is the unprocessed error message emitted by RacerX (a post-processing pass can make the output prettier, but more verbose). Properly interpreted they indicate that this error should be ranked high in the error messages. The main features:

1. The field `nvars=4` indicates there are four accesses to shared state in this "non-critical section." While there are safe ways to update a single variable without locks while sharing it with others, modifying and reading a set of variables is almost certainly an error. The aggregate set will take on values impossible under any valid sequential interleaving. In this case, we can also lose an update to the field `enabledPassthroughWords`.
2. The field `has_locked=1` specifies that locking was used earlier in the call chain, suggesting that we are in a multithreaded context.
3. The field `modified=1` specifies that at least one of the shared variables was modified.

This is a real error that has since been fixed. It was caused because the developer forgot to adapt this function when porting the code to SMP. When the system is run on a uniprocessor, there is no problem. When it is run on an SMP machine, things get more exciting.

8.5 Does `X` need to be protected by `L`?

We infer whether a given lock protects a variable using a statistical approach similar to that in the previous section. The inference works in two passes. The first does a local pass over the entire system counting (1) the total number of accesses to a variable or routine and (2) the number of times these accesses held a specific lock. The second pass filters the candidate locks for each variable by picking a single "best" lock out of all the candidates and then doing an interprocedural pass checking with it. Since we can make mistakes in deciding that lock `l` protects `x`, we rank errors where *no* lock is held over those where the wrong one was. The main refinements in practice are (1) deciding what to count and (2) some evidence is more compelling than others.

What to count. we have made two mistakes in deciding what to count. The first mistake is to record whether or not a given statement accesses a variable or routine with a lock held every time the statement is encountered during analysis. Doing so has two problems. First, the number of times a statement is hit has little to do with anything other than that there are many paths to it that acquire different locks (and thus defeat caching). Second, a single piece of code really only gives one piece of evidence that the programmer believed a lock protected a variable. The number of times it is hit during analysis is irrelevant. Thus, we only allow each point in the CFG to make exactly one contribution to a lock count.

A second mistake is to not take strong steps to suppress coincidental lock pairings. For example, if we call `foo` with lock `l` held it is generally a bad idea to mark accesses to variables in `foo` as being protected by `l` — it will tend to access many related things, which will thus have high numbers of successful accesses under `l`, polluting the checking results. Similarly, if variable `v` is accessed within a critical section we only count it once, not matter how many times it was accessed. Not suppressing multiple counts causes problems because a few critical sections may make heavy use of a given variable that is (coincidentally) then viewed as heavily protected by a lock and its errors then ranked highly.

What evidence really matters. As in the previous subsection, we give special weight to the first, last, and only shared data in a critical section, since they imply that the programmer believes that the critical section's lock protects them. These are also great examples to display during inspection since they make it very clear to a user of RacerX what exactly is being protected.

An example error: Figure 8 gives a simple example of this. There were 37 accesses to `serial_out` with the argu-

```
// ERROR: linux-2.5.62/drivers/char/esp.c:
//    2313:block_til_ready: calling <serial_out-info>
//    w/o cli!
restore_flags(flags); // re-enable interrupts
...
// non-disabled access to serial_out-info!
serial_out(info, UART_ESI_CMD1, ESI_GET_UART_STAT);

// Example 1 drivers/char/esp.c:1206
save_flags(flags); cli();
/* set baud */
serial_out(info, UART_ESI_CMD1, ESI_SET_BAUD);
serial_out(info, UART_ESI_CMD2, quot >> 8);
serial_out(info, UART_ESI_CMD2, quot & 0xff);
restore_flags(flags);

// Example 2: rivers/char/esp.c:1426
cli();
info->IER &= ~UART_IER_RDI;
serial_out(info, UART_ESI_CMD1, ESI_SET_SRV_MASK);
serial_out(info, UART_ESI_CMD2, info->IER);
serial_out(info, UART_ESI_CMD1, ESI_SET_RX_TIMEOUT);
serial_out(info, UART_ESI_CMD2, 0x00);
sti();
```

Figure 8: Error caught by inferring the serial_out must be called with interrupts disabled. There were 28 places where the routine serial_out was used as the first or last statement in a critical section.

ment info with interrupts disabled, in contrast there was only one non-disabled use. The routine-argument pair was the first statement of a critical section 11 times and the last one 17 times. Even knowing nothing about the system, simply looking at the examples makes it obvious that the programmer is explicitly disabling interrupts before invoking this routine. Having such examples makes inspection much easier. In practice we almost always look at errors that have such features before those that do not.

8.6 How to find false negatives?

While eliminating false positives is important, so is removing false negatives. Below we describe three methods we use to guard against different types of false negatives.

Statistical inference of locking functions. Obviously, the more locking functions RacerX knows of, the more effective it will be. While it uses interprocedural analysis to propagate information across procedure boundaries, it requires knowing a set of "root" locking functions. Although many such functions are well-known, large systems have a startling number of special-purpose locking routines. To infer these functions we use a variant of the belief analysis used in [14] to infer paired functions. To determine if functions a and b should be paired, we count how often they are, how often they are not, and rank the results using the z-test statistic. We modify this approach to give high priority to functions implemented in the same file and functions that have suggestive names ("lock," "unlock," "disable," etc.). Users can inspect derived pairs either as a means to ensure that they did not miss any, or as a way to simply infer all of them from the source code. There were forty such functions in System X, including four that we had missed, despite working with its source code in collaboration with core implementors for over eight months.

Errors in lockset analysis. Static checkers have the invidious problem that any errors in their analysis that cause false negatives are silent. Some cleverness is needed to find ways to detect such silent failures. We have three main ways.

System	Bug	Unconfirmed	Benign	False
System X	7	4	13	14
Linux 2.5.62	3	2	2	6

Table 5: Race results for System X and Linux. Benign errors are races that are not considered important enough to fix (e.g., statistics variables).

First, many systems, such as FreeBSD and System X, make heavy use of locking assertions which we can check statically. The most prevalent example are places where the programmer asserts that a given lock is held:

```
ASSERT(Is_Locked(&lock));
```

RacerX checks its lockset each time it reaches such assertions and emits an error if the lock is not in the lockset: such missing locks signal either programmer or analysis mistakes.

Second, we exploit the fact that we can check for other errors. Flagging lock pairing errors gives us an easy way to ensure that our system correctly matches locks.

Third, we again exploit the fact that programmers do not gratuitously do locking operations. The intent of a critical section is to protect something. Thus, we can detect false negatives in RacerX by counting how many shared variables each critical section contains. Critical sections with zero count either indicate the programmer made a mistake (e.g., a typo in a variable name) or that our state tracking is broken. We found eight errors in the RacerX implementation when we modified it to flag empty critical sections. There were six minor errors, one error where we mishandled arrays (and so ignored all array uses indexed by pointer variables) and a particularly nasty silent lockset caching error that caused us to miss over 20% of all code paths. We plan to extend this approach more globally and ensure that every shared variable is associated with some lock and that every lock is associated with some shared state.

8.7 Race results

Table 5 summarizes the results for System X and Linux. System X had seven hard bugs, four unconfirmed bugs, and thirteen harmless errors. Six of the false positives were due to signal-wait semaphores; we obtained these results before developing the analysis described in Section 5.1. The other eight were unfortunate interactions with function pointers.

We had fewer inspected messages for Linux than for System X because it was harder to get confirmation of them. There were three valid bugs (all fixed), two unconfirmed errors, two errors involving statistic variables and six false positives.

9. CONCLUSION

RacerX is a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. It uses novel strategies to infer checking information such as which locks protect which operations, which code contexts are multithreaded, and which shared accesses are dangerous. We applied it to FreeBSD, Linux and a large commercial code base and found serious errors in all of them.

Acknowledgements

This research was supported by DARPA contract MDA904-98-C-A933 and by an NSF Career Award. We are especially grateful for the last-minute rewriting help of Godmar Back and Ted Kremenek. We thank Stefan Savage, Madanlal Musuvathi, Rushabh Doshi, and the anonymous reviewers for their feedback.

10. REFERENCES

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[2] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[3] M. Burrows and K. Leino. Finding stale-value errors in concurrent programs. Technical Report SRC-TN-2002-004, Compaq Systems Research Center, May 2002.

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[5] S. Chandra, B. Richards, and J. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–33, May-June 1999.

[6] Cheng, Feng, Leiserson, Randall, and Stark. Detecting data races in cilk programs that use locks. In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[7] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[10] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[11] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[12] D. Detlefs, K. R. M. Leino, G. Nelson, and J. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, Dec. 1998.

[13] A. Dinning and E. Schonberg. An empirical comparision of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990.

[14] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[15] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[16] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods Europe*, pages 500–517, Mar. 2001.

[17] D. Freedman, R. Pisani, and R. Purves. *Statistics*. W.W. Norton, third edition edition, 1998.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (2nd Edition)*. Addison-Wesley, 2000.

[19] D. Grossman. Type-safe multithreading in cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, Jan. 2003.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[21] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, 1994.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[23] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–116, Feb. 1980.

[24] K. M. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, Oct. 2001.

[25] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 Supercomputer Debugging Workshop*, 1991.

[26] A. Morton. Personal communication. Semantics and deadlock implications of the Linux BKL, Feb. 2003.

[27] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, 1996.

[28] T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th Annual Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[30] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the 1993 USENIX Winter Technical Conference*, pages 97–106, 1993.