Dataflow Analysis for Concurrent Programs using Datarace Detection *

Ravi Chugh Jan W. Voung Ranjit Jhala Sorin Lerner

University of California, San Diego

{rchugh,jvoung,jhala,lerner}@cs.ucsd.edu

Abstract

Dataflow analyses for concurrent programs differ from their single-threaded counterparts in that they must account for shared memory locations being overwritten by concurrent threads. Existing dataflow analysis techniques for concurrent programs typically fall at either end of a spectrum: at one end, the analysis conservatively kills facts about all data that might possibly be shared by multiple threads; at the other end, a precise thread-interleaving analysis determines which data may be shared, and thus which dataflow facts must be invalidated. The former approach can suffer from imprecision, whereas the latter does not scale.

We present RADAR, a framework that automatically converts a dataflow analysis for sequential programs into one that is correct for concurrent programs. RADAR uses a race detection engine to kill the dataflow facts, generated and propagated by the sequential analysis, that become invalid due to concurrent writes. Our approach of factoring all reasoning about concurrency into a race detection engine yields two benefits. First, to obtain analyses for code using new concurrency constructs, one need only design a suitable race detection engine for the constructs. Second, it gives analysis designers an easy way to tune the scalability and precision of the overall analysis by only modifying the race detection engine. We describe the RADAR framework and its implementation using a pre-existing race detection engine. We show how RADAR was used to generate a concurrent version of a null-pointer dereference analysis, and we analyze the result of running the generated concurrent analysis on several benchmarks.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Validation; F.3.2 [*Semantics of Programming Languages*]: Semantics of Programming Languages – Program analysis

General Terms Languages, Reliability, Verification

Keywords Interprocedural Analysis, Locksets, Multithreaded Programs, Summaries

* This work was supported by NSF CAREER grants CCF-0644306, CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603, and the UCSD FWGrid Project, NSF Research Infrastructure Grant Number EIA-0303622.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

1. Introduction

Advances in static algorithms for program optimization and error detection have shown that compiler technology can dramatically improve the reliability and performance of computer systems. Most of these algorithmic advances are limited to *sequential* programs and ignore the challenges introduced by *concurrency*, where the need for static checking and potential for optimization are even greater. The main difficulty with concurrent code is that for an analysis to be sound, it must account for the concurrent *interleaving* of multiple executions.

Consider the following scenario found in an old version of the Linux kernel. A thread acquires the lock on a list to find an array within the list. Once found, it releases the lock. It then reads the array length once and iterates through the array. A sequential analysis would unsoundly report the system safe – as the thread first checks the array size before indexing the array. Unfortunately, another thread can change the array size in the time *between* the check and the index, thereby causing a memory error.

Currently, the problem of analyzing concurrent programs is addressed in one of the following ways. First, the programmer can provide annotations (*e.g.* `volatile`) that alert the compiler to the pieces of data that can be modified by concurrent threads. The compiler can then safely ignore these pieces of data – *i.e.,* not perform any analysis that depends on them. Unfortunately, this solution is error-prone as the programmer can mistakenly forget annotations. Second, the compiler could sidestep the need for annotations using an *escape analysis* that determines if a piece of data is modified by multiple threads [23]. However, the precision of these analyses is inherently limited – even if the escape information is perfect, there are some *critical pieces* of shared data about which the analysis can infer nothing, making it impossible, for example, to statically prove the safety of dereferences of shared pointers or the access of shared arrays. Third, to overcome this imprecision, one can use custom concurrent analyses – tailored to specific problems and models of concurrency – to infer specific kinds of information [9, 13, 22]. These analyses can be precise, but one must painstakingly retool a new analysis for each concurrent setting. Finally, one could use model checking to infer facts by exhaustively exploring all thread interleavings [2, 7, 11]. While this is an extremely precise and generic approach, such analyses are unlikely to scale due to the combinatorial explosion in the number of interleavings.

We present a solution to the problem of precisely analyzing concurrent programs in a scalable way. Our solution is based on two insights. First, the most common way for a programmer to ensure a fact about a piece of shared data at any given point in a thread is to ensure that no other thread can *modify* the data while the first thread is still at that point. Our second insight consists of a way of using *race detection* to determine when dataflow facts may be killed by the actions of other threads. A *data race* occurs when multiple threads are about to access the same piece of memory and at least one of those accesses is a write. Since data races are a common source of tricky bugs, several static analyses have been developed to find races, or show their absence. Our insight is that to determine whether the actions of other threads can invalidate, or *kill*, a fact inferred about some data at some point, it suffices to determine whether an *imaginary read* of the data at the point can race with a write to that data by another thread.

We combine these insights in a framework called RADAR that takes as input a *sequential* dataflow analysis and a *race detection* engine, and returns as output a version of the sequential analysis that is sound for multiple threads. RADAR combines our insights as follows. It first runs the sequential analysis. At each program point, after the transfer function for the sequential dataflow analysis has propagated facts to the point, RADAR queries the race detector to determine which facts must be killed due to concurrency. More precisely, for each propagated fact, RADAR asks the detector if an imaginary read, *at that program point*, of the memory locations that the fact depends on can race with writes performed by other threads. If the answer is yes (that is, if another thread may be concurrently writing to one of the locations), then the dataflow fact is killed. If the answer is no (that is, if no other threads can possibly be writing to these locations), then the dataflow fact remains valid in the concurrent setting.

RADAR's approach of factoring all reasoning about concurrency into a race detection engine is less precise than a custom analysis that may also generate facts from concurrent writes. However, RADAR yields two concrete benefits. First, to obtain analyses for code using new concurrency constructs, one need only design a suitable race detection engine for the constructs. Second, it gives analysis designers an easy way to tune the scalability and precision of the overall analysis by only modifying the race detection engine.

To sum up, the main contributions of our paper are as follows.

- We have designed a framework called RADAR that automatically converts a sequential dataflow analysis into a concurrent one using a race detection engine (Section 3).
- We have instantiated the RADAR framework with an existing race detection engine called RELAY [29]. We describe this implementation, which we call RADAR(RELAY) (Section 4).
- We have used RADAR(RELAY) to transform a sequential null-pointer analysis into a concurrent null-pointer analysis. We describe the null-pointer analysis and evaluate the precision of the resulting concurrent null-pointer analysis. We demonstrate that RADAR(RELAY) easily scales to hundreds of thousands of lines of code, and achieves good precision relative to some appropriate upper and lower bounds (Section 5).

2. Overview

We begin with an overview of our technique using some simple examples. First, consider the multithreaded program shown in Figure 1, which executes a single copy of the `Producer` thread and a single copy of the `Consumer` thread. There is a shared, acyclic list of structures named `bufs`, and a shared performance counter `perf_ctr`. To enable mutually exclusive list access, there is a lock `buf_lock` which is initially "unlocked", *i.e.,* not held by any thread. The `Producer` (respectively `Consumer`) thread has a local reference `px` (respectively `cx`) used to iterate over the list `bufs`.

The `Producer` thread iterates over the cells in the list `bufs`. In each iteration, it acquires the lock `buf_lock` protecting the cell, and resets the `px→data` to a new buffer initialized with the value `0` that will hold the data that will be produced. Next, it obtains the value, via a call to `produce` and once it is ready, the producer writes the value into `*px→data` and moves onto the next cell.

The `Consumer` thread iterates over the cells in the list `bufs`. In each iteration, it acquires the lock `buf_lock` and if the pointer `cx→data` is non-null, it consumes the data, resets the pointer to free the buffer, and moves to the next cell in the list. Finally, the consumer releases the lock.

We assume that the shared list contains no cycles and that it starts off with all the data fields set to `NULL`. Thus, the net effect of having the `Producer` and `Consumer` running in parallel is that the producer walks through the list setting the data field of each individual cell, and the consumer trails behind the producer, using the data field in each cell and resetting it to `NULL`. Finally, notice the `Consumer` thread initializes `perf_ctr` without holding any locks, and so the initialization races with the increment operation at `P5` in the `Producer` thread. However, we shall assume that the programmer has deemed that the race is benign as it is on an irrelevant performance counter.

2.1 Sequential Non-Null Analysis

Suppose we wish to statically determine whether any null-pointer dereference occurs during the execution of the program in Figure 1. To this end, we could perform a standard sequential dataflow analysis, using flow facts of the form $NonNull(l)$ stating that the lvalue l is non-null, and flow functions that appropriately generate, propagate, and kill such facts using guards and assignments.

Let us assume that `new()` returns a non-null pointer to a cell initialized with `0`. The set of facts in the fixpoint solution computed by this analysis is shown on the left in Figure 1 (for the moment, ignore the line crossing out a fact on the last line). At points `P0` and `P1`, every lvalue may be null. At `P2` and `P3` `px` is non-null, and everywhere else, the sequential analysis determines that the lvalues `px` and `px→data` are non-null. Thus, the analysis determines that at each program point where a pointer is dereferenced, the pointer is non-null, and so all the dereferences are safe.

Sequential analysis is unsound, even without data races. In this case, the above conclusion is sound: there are indeed no null-pointer dereferences in the program. In general, however, a sequential non-null dataflow analysis may actually miss some null-pointer dereferences. As an example, consider a scenario where the programmer who wrote Figure 2 mistakenly uses the intuition that the system is safe as long as there are no *data races* on any of the shared cells. Thus, to improve the performance of the program from Figure 1, the programmer writes the modified version of the `Producer` thread shown in Figure 2, where `buf_lock` is temporarily released while the data is being produced to allow the `Consumer` thread to concurrently make progress. The resulting system has no races, and so the programmer may think that the system is safe.

However, this intuition turns out to be incorrect, and in fact the programmer has actually introduced a null-pointer bug, even though there are no races. This is because *after* the producer thread has initialized `px→data` and released the lock, the consumer can acquire the lock and reset the pointer. When the producer thread re-acquires the lock after storing the data temporarily in `t`, the dereference at `P8` can cause a crash because the pointer is null.

Unfortunately, even though the new program has a null-pointer bug, the sequential analysis returns exactly the same solution as for the original program (shown on the left in Figure 2). That is, at each point the same lvalues are deemed to be non-null, and thus the sequential analysis would not discover the null-pointer bug.

2.2 The Problem: Adjusting for Multiple Threads

To address the above problem, we want to to *automatically* convert a sequential dataflow analysis into one that is sound in the presence of multiple threads. Doing this in a way that is also precise is not trivial.

Adjusted Analysis	buffer_list *bufs; lock buf_lock; int perf_ctr;
	`thread producer1(){`
$\emptyset$	`P0: px = bufs;`
$\emptyset$	`P1: while (px != NULL){`
px	`P2:   lock(buf_lock);`
px	`P3:   px->data = new();`
px->data,px	`P5:   perf_ctr++;`
px->data,px	`P6:   t=produce();`
px->data,px	`P8:   *px->data = t;`
px->data,px	`P9:   unlock(buf_lock);`
~~px->data~~,px	`PA:   px = px->next;`
	`}`
	`thread consumer1(){`
$\emptyset$	`perf_ctr = 0;`
$\emptyset$	`C0: cx = bufs;`
$\emptyset$	`C1: while(cx != NULL){`
cx	`C2:   lock(buf_lock);`
cx	`C3:   if(cx->data != NULL){`
cx->data,cx	`C4:     consume(*cx->data);`
cx->data,cx	`C5:     cx->data = NULL;`
cx	`C6:     cx = cx->next;`
	`}`
$\emptyset$	`C7:   unlock(buf_lock);`
	`}`

Figure 1. `Producer-Consumer` program

Adjusted Analysis	buffer_list *bufs; lock buf_lock; int perf_ctr;
	`thread producer1(){`
$\emptyset$	`P0: px = bufs;`
$\emptyset$	`P1: while (px != NULL){`
px	`P2:   lock(buf_lock);`
px	`P3:   px->data = new();`
px->data,px	`P4:   unlock(buf_lock);`
~~px->data~~,px	`P5:   perf_ctr++;`
~~px->data~~,px	`P6:   t=produce();`
~~px->data~~,px	`P7:   lock(buf_lock);`
~~px->data~~,px	`P8:   *px->data = t;`
~~px->data~~,px	`P9:   unlock(buf_lock);`
~~px->data~~,px	`PA:   px = px->next;`
	`}`
	`thread consumer1(){`
$\emptyset$	`perf_ctr = 0;`
$\emptyset$	`C0: cx = bufs;`
$\emptyset$	`C1: while(cx != NULL){`
cx	`C2:   lock(buf_lock);`
cx	`C3:   if(cx->data != NULL){`
cx->data,cx	`C4:     consume(*cx->data);`
cx->data,cx	`C5:     cx->data = NULL;`
cx	`C6:     cx = cx->next;`
	`}`
$\emptyset$	`C7:   unlock(buf_lock);`
	`}`

Figure 2. Buggy version

Adjusted Analysis	buffer_list *bufs; int flag; int perf_ctr;
	`thread producer2(){`
$\emptyset$	`P0: px = bufs;`
$\emptyset$	`P1: while (px != NULL){`
px	`P2:   while(px->flag !=0){};`
px	`P3:   px->data = new();`
px->data,px	`P5:   perf_ctr++;`
px->data,px	`P6:   t=produce();`
px->data,px	`P8:   *px->data = t;`
px->data,px	`P9:   px->flag = 1;`
~~px->data~~,px	`PA:   px = px->next;`
	`}`
	`thread consumer2(){`
$\emptyset$	`perf_ctr = 0;`
$\emptyset$	`C0: cx = bufs;`
$\emptyset$	`C1: while(cx != NULL){`
cx	`C2:   while(cx->flag==0){};`
cx	`C3:   if(cx->data != NULL){`
cx->data,cx	`C4:     consume(*cx->data);`
cx->data,cx	`C5:     cx->data = NULL;`
cx	`C6:     cx = cx->next;`
	`}`
$\emptyset$	`C7:   cx->flag = 0;`
	`}`

Figure 3. Flag-based version

Consider for example the simple solution of running an escape analysis, and keeping only *NonNull* facts for those lvalues that do not escape the current thread. The `px`→`data` field escapes the current thread, so the analysis would never infer *NonNull*(`px`→`data`). Although this solution makes the analysis sound in the face of concurrency (and in particular, it would find the bug in Figure 2), it also makes the analysis imprecise: the resulting concurrent analysis would not even be able to show that the original program from Figure 1 is free of null-pointer bugs.

Alternatively, one may be tempted to run independent sequential analyses over blocks that are *atomic* in the sense of [16, 7] and conservatively *kill* facts over shared variables at atomic block exit points [3]. Intuitively, a block is atomic if for each execution of the operations of the block where the operations are interleaved with those of other threads, there exists a semantically equivalent execution where the operations of the block are run without interleaving. Unfortunately, the body of the `Producer` loop from Figure 1 is not atomic because of the benign race on the performance counter `perf_ctr`. This race splits the body of `Producer` into multiple atomic blocks – the statements before, at, and after the racy increment. Thus, such an analysis would kill the *NonNull*(`px`→`data`) fact at `P5`, and would be too imprecise to prove that the program from Figure 1 is free of null-pointer bugs.

2.3 Our Solution: Pseudo-Race Detection

We now describe our solution to this problem, which allows us to leverage existing race detection engines to build a sound concurrent analysis that is strictly more precise than the previously mentioned simple approaches. As discussed in Section 6, race detection is a well-studied problem. For programs using lock-based synchronization, there are scalable race detectors that infer the sets of locks that protect each shared memory location and produce a race warning if two threads access a shared cell without holding a common lock.

Adjust. Our first insight is that the facts that can soundly be inferred to hold in the presence of multiple threads are the *subset* of facts established via sequential analysis which are not killed by operations of other threads. Thus, the multithreaded dataflow analysis can be reduced to determining which facts inferred at a particular point by the sequential analysis are killed by other threads. To determine if a fact can be killed by a concurrently executing operation of another thread, it suffices to check if another thread can *concurrently write* any lvalue appearing in the flow fact.

Pseudo-Races. Our second insight is that to perform this check we can insert *pseudo-reads* corresponding to the lvalues in the flow fact at the program point, and query a *race detection* engine to determine if any of the pseudo-reads can race with a write to the same memory location. If such a *pseudo-race* occurs, then the fact is killed; otherwise, the analysis deduces that the fact continues to hold at the point even in the presence of other threads.

We have designed a framework called RADAR that combines these two insights to convert an arbitrary dataflow analysis for sequential programs into one that is sound for multithreaded programs. During analysis, RADAR uses the sequential flow function, but at each program point, it kills the facts over lvalues that have pseudo-races at that point. This mechanism captures the following informal idiomatic manner in which the programmer reasons about multiple threads. Each thread performs some operation that establishes a certain fact in the programmer's head, *e.g.* a null check or initialization or an array bounds check. The programmer can only expect that the fact continues to hold as long as other threads cannot modify the memory locations. As a result, the programmer uses synchronization mechanisms to "protect" the memory locations from writes by other threads as long as the information is needed. Our technique of adjusting preserves only those facts that the race detection engine deems to be protected from modification.

2.4 Multithreaded Non-Null Analysis

Let us consider the result of running the non-null analysis adjusted using RADAR to account for multiple threads. The lines in Figures 1 and 2 show the facts generated by the sequential analysis that get killed during the adjusting because of pseudo-races.

For both the correct and the buggy programs, in the `Consumer` thread the adjusting has no effect because `cx` is thread-local, and due to the held lock `buf_lock`, there are no races on the pseudo-reads of `cx`→`data` at program points `C4` and `C5`.

Safety without Atomicity. In the correct `Producer` thread of Figure 1, the adjusting process has no effect on facts over (only) the thread-local, and hence, race-free lvalue `px`. The initialization at `P3` causes the fact $\mathit{NonNull}(\texttt{px}{\rightarrow}\texttt{data})$ to get generated at program point `P5`. The adjusting does not kill this fact because the lock `buf_lock` held at `P5` ensures there is no pseudo-race on $\texttt{px}{\rightarrow}\texttt{data}$. Similarly, the fact $\mathit{NonNull}(\texttt{px}{\rightarrow}\texttt{data})$ generated at `P3` is not killed by the adjusting at `P6`−`P9`, as the held lock `buf_lock` ensures there are no races with the pseudo-read on $\texttt{px}{\rightarrow}\texttt{data}$ at any of these points. As the lock is released at `P9`, the fact $\mathit{NonNull}(\texttt{px}{\rightarrow}\texttt{data})$ is killed by the adjusting at `PA`, as the pseudo-read can race with the write in the `Consumer` thread. The adjusted analysis shows that the dereferences in the program are safe, as $\texttt{px}{\rightarrow}\texttt{data}$ is soundly inferred to be non-null at `P8`, where the dereference takes place. Notice the adjusted analysis can soundly show that the program does not cause any null-pointer dereferences, even though the producer thread, even the loop-body, is not atomic, due to `perf_ctr` which may be accessed without any synchronization. By preserving the facts that are over protected lvalues, our adjusting technique can ignore atomicity "breaks" caused by *benign races* on irrelevant entities like `perf_ctr`. Thus, our RADAR adjusting technique is strictly more precise than running independent sequential analyses over semantically atomic blocks.

Concurrency Errors without Races. In the buggy `Producer` thread of Figure 2, the facts over the thread-local lvalues are not killed by adjusting. However, notice that although the sequential analysis would propagate the fact $\mathit{NonNull}(\texttt{px}{\rightarrow}\texttt{data})$ to `P5`, the adjusted version kills the fact since once the protecting lock is released at `P4`, the pseudo-read of $\texttt{px}{\rightarrow}\texttt{data}$ at `P5` can race with the write at `C5` in a `Consumer` thread. As this fact is killed at `P5`, it does not propagate in the adjusted analysis to `P6` − `P9`, as happens in the sequential analysis. Thus, as a result of the adjusting, the dereference at `P8` is no longer inferred to be safe as $\texttt{px}{\rightarrow}\texttt{data}$ may be null at this point! Thus, our technique finds an error caused by multi-threading that is absent from the sequential version of the program, *even though there are no data races* in the program except on the irrelevant `perf_ctr`.

Beyond lock-based synchronization. Although we have used lock-based synchronization to show how RADAR works, our adjusting technique is applicable to any synchronization regime for which race detection techniques exist, not just those based on locks. Consider a version of the `Producer-Consumer` example, shown in Figure 3, which has finer-grained synchronization done with a flag field in each of the structures in the cyclic list. Now, instead of acquiring the lock, the `Producer` thread spins in a loop while $\texttt{px}{\rightarrow}\texttt{flag}$ is non-zero, which indicates that the data in the structure has not yet been consumed. Once the flag is zero, the producer, initializes the $\texttt{px}{\rightarrow}\texttt{data}$ field, writes the new data into it, and sets the $\texttt{px}{\rightarrow}\texttt{flag}$ field to `1` indicating the data is ready. Dually, the `Consumer` thread spins while the $\texttt{cx}{\rightarrow}\texttt{flag}$ field is zero, at which point it consumes the data and resets the $\texttt{cx}{\rightarrow}\texttt{data}$ field. The result of the adjusted analysis for this program is identical to the result for the fixed program of Figure 2, as a more general race detection engine (*e.g.* one based on model checking [11]) would deduce that there were no pseudo-races on $\texttt{px}{\rightarrow}\texttt{data}$ in the locations `P5` − `P9`. Once the flag is set at `P9`, the pseudo-read of $\texttt{px}{\rightarrow}\texttt{data}$ at `PA` can race with the write at `C5` in a `Consumer` thread, and so the adjust kills the fact $\mathit{NonNull}(\texttt{px}{\rightarrow}\texttt{data})$ at `PA`.

In the rest of the paper, we formalize the RADAR framework and show how it converts sequential analyses into concurrent ones.

3. The RADAR Framework

This section presents the RADAR framework for concurrent dataflow analysis in several steps. We start by presenting (in Section 3.1) a basic version of RADAR. Although this basic version lacks certain important features and optimizations (such as support for function calls), it illustrates the foundation of our approach. We then gradually refine the basic framework (in Sections 3.2 and 3.3) by adding various optimizations and features.

Assumptions. We make two standard assumptions about the program being analyzed and the system it is to be run on. First, we assume that for each procedure, either we have its code or we have a summary that soundly approximates its behaviors. As a result, our framework can analyze incomplete programs (*e.g.* programs that use libraries), since we can model the missing procedures using summaries. Second, we assume that the shared memory system is sequentially consistent [14] in that memory operations are executed in the order in which they appear in the program.

3.1 Intraprocedural Framework

We start by presenting a basic framework for generating an intraprocedural concurrent dataflow analysis from an intraprocedural sequential dataflow analysis.

Sequential Dataflow Analysis. We assume a Control Flow Graph (CFG) representation of programs, where each node represents a statement, and edges between nodes are program points where dataflow information is computed. We use Node to represent the set of all CFG nodes and PPoint the set of all program points. In the program shown in Figure 1, the program point `P3` is the point just before executing the statement at `P3`.

We assume that the sequential dataflow analysis computes a set of dataflow facts at each program point, where the set of all possible dataflow facts is $\mathit{DataflowFact}$. For the purposes of exposition, we assume that dataflow facts in $\mathit{DataflowFact}$ are must facts, which means that may information, if needed, has to be encoded by the absence of must information. Although we make this assumption in our exposition, our implementation explicitly supports may facts. Thus, the domain of the sequential dataflow analysis is $D = \mathcal{P}(\mathit{DataflowFact})$, ordered as a lattice $(D, \sqsubseteq, \top, \bot, \sqcap, \sqcup)$, where $\sqsubseteq$ is $\supseteq$, $\top$ is $\emptyset$, $\bot$ is $\mathit{DataflowFact}$, $\sqcap$ is $\cup$, and $\sqcup$ is $\cap$.

We also assume that the flow function is given as:

$$F : \mathit{Node} \times D \times \mathit{PPoint} \rightarrow D$$

Given a node n, some incoming dataflow information d, and an outgoing program point p for node n, $F(n, d, p)$ returns the outgoing dataflow information. We assume that if a node n has more than one incoming program point, the dataflow information is merged using $\sqcup$ before being passed to the flow function.

Examples of dataflow facts that can be propagated include: $\mathit{HasConstantValue}(\texttt{x}, 5)$, which states that `x` has the value of 5, $\mathit{MustPointTo}(\texttt{x}, \texttt{y})$, which states that `x` points to `y`, and $\mathit{NonNull}(\texttt{p})$, which states that `p` is safe to dereference.

Requirement on Dataflow Information. As our framework does not depend on the exact details of the $\mathit{DataflowFact}$ set, analysis writers have the freedom to choose the way in which they encode dataflow information. However, we do place a requirement on the $\mathit{DataflowFact}$ set: we assume the existence of a function lvals that returns the set of lvalues that a fact depends on. Intuitively, given a dataflow fact $f \in \mathit{DataflowFact}$, $\mathit{lvals}(f)$ returns the set of lvalues that, if written to with arbitrary values, would invalidate the fact f. We denote the set of all lvalues by LVal, and so the function lvals has type $\mathit{DataflowFact} \rightarrow \mathcal{P}(\mathit{LVal})$. As an example, for the dataflow facts mentioned above, we would have:

$$\begin{aligned} \mathit{lvals}(\mathit{HasConstantValue}(\texttt{x}, 5)) &= \{\texttt{x}\} \\ \mathit{lvals}(\mathit{MustPointTo}(\texttt{x}, \texttt{y})) &= \{\texttt{x}\} \\ \mathit{lvals}(\mathit{NonNull}(\texttt{p})) &= \{\texttt{p}\} \end{aligned}$$

Although we assume that the lvals function is given, it can easily be computed from the sequential flow function F if F handles "havoc"

CFG nodes of the form "$l := \bot$". In particular:

$$lvals(f) = \{l \mid l \in LVal \wedge f \notin F(\text{"}l := \bot\text{"}, \{f\}, \_)\}$$

Concurrent Dataflow Analysis. We capture the way in which concurrency affects the sequential dataflow analysis through a function $ThreadKill : PPoint \times LVal \rightarrow Bool$. Given a program point p and an lvalue l, $ThreadKill(p, l)$ returns whether or not l may be written to by concurrent threads when the current thread is at program point p. The $ThreadKill$ function, which we will define later in terms of a race detection engine, is at the core of our technique: it allows RADAR to kill dataflow facts that are made invalid by concurrent writes.

Given the sequential flow function F and the $ThreadKill$ function, we define F_{Adj}, the flow function for the concurrent analysis:

$$\begin{aligned} F_{Adj}(n, d, p) = \{f \mid \quad & f \in F(n, d, p) \wedge \\ & \forall l \in lvals(f) \,.\, \neg ThreadKill(p, l)\} \end{aligned}$$

This adjusted flow function essentially propagates dataflow facts that are produced by the original flow function F and that are not killed by any concurrent writes.

Adjusting via Race Detection. The key contribution of our work lies in the way in which we use a race detection engine to compute $ThreadKill$. As a result, we need a way to abstract the race detection engine. We achieve this through a function $RacyRead : PPoint \times LVal \rightarrow Bool$, which behaves as follows: given a program point p and an lvalue l, $RacyRead(p, l)$ returns $true$ if a read of l at program point p would cause a race.

Soundness. For our framework to be sound, the race detection engine must be sound, in the sense that if there really is a race, then $RacyRead$ must return $true$ (but $RacyRead$ can also return $true$ in cases where there is no race). To formalize this soundness property, we assume a perfect race detection oracle $RealRace : PPoint \times LVal \rightarrow Bool$, such that $RealRace(p, l)$ returns $true$ exactly when there is an execution in which a read of l at p would cause a race. The following requirement states that the race detection engine $RacyRead$ must approximate the oracle $RealRace$:

$$\begin{aligned} & \forall p \in PPoint, l \in LVal \,. \\ & \quad RealRace(p, l) \Rightarrow RacyRead(p, l) \end{aligned} \tag{1}$$

Having a sound race detection engine, the $ThreadKill$ function can then be defined as:

$$ThreadKill(p, l) = RacyRead(p, l)$$

This basic definition of $ThreadKill$ expresses the key insight behind the RADAR framework, which is that pseudo-races can be used as a way of determining when concurrent writes could happen.

Instantiation Requirements: To instantiate the basic RADAR framework, one needs to provide a race detection engine $RacyRead : PPoint \times LVal \rightarrow Bool$ that satisfies the soundness property (1).

3.2 Optimized Framework: Race Equivalence Regions

The basic RADAR framework from Section 3.1 performs a race check at each program point for each lval that the dataflow facts depend on. This can lead to a large number of race checks, in the worst case $n \times m$, where n is the number of program points and m is the number of lvalues. To reduce this large number of race checks, we partition program points into race equivalence regions.

Intuitively, a race equivalence region is a set of program points that have the same raciness behavior: for each lvalue, either the lvalue is racy throughout the entire region, or it is not racy throughout the entire region. It is not possible for an lvalue to be racy in one part of the region and not racy in another part. Race equivalence regions reduce the number of race checks because by checking the raciness of an lvalue at (any) one program point in a region, RADAR can know the raciness of the lvalue throughout the entire region.

Race Equivalence Regions. Formally, we define a partitioning of program points into race equivalence regions as a pair (R, Reg), where R is a set of regions, and $Reg : PPoint \rightarrow R$ is a function mapping each program point to a region. Two program points p and p' are *race equivalent* if $Reg(p) = Reg(p')$. We say a program point p *belongs to* a region r if $Reg(p) = r$. Since all program points belonging to a region are equivalent in terms of race detection, we change the interface to the race detection engine to take a race equivalence region rather than a program point:

$$RacyRead : R \times LVal \rightarrow Bool$$

One possible implementation for this new $RacyRead$ is to choose a unique representative program point for each region, and when queried with a particular region r and lvalue l, to return the result of the old $RacyRead$ on r's representative point and l.

Soundness. Instead of imposing a particular way of implementing $RacyRead$, we define a soundness requirement for the new $RacyRead$ engine and the Reg function:

$$\begin{aligned} & \forall p \in PPoint, l \in LVal \,. \\ & \quad RealRace(p, l) \Rightarrow RacyRead(Reg(p), l) \end{aligned} \tag{2}$$

With this new abstraction for the race detection engine, the $ThreadKill$ function becomes:

$$ThreadKill(p, l) = RacyRead(Reg(p), l)$$

This new definition of $ThreadKill$ reduces the number of race checks for each lvalue from at most once per program point to at most once per region.

Instantiation Requirements: To instantiate the region-based RADAR framework, one needs to provide:

1. A **race detection engine** $RacyRead : R \times LVal \rightarrow Bool$
2. A **region map** $Reg : PPoint \rightarrow R$

such that $RacyRead$ and Reg satisfy property (2). We now give two examples to illustrate the idea of race equivalence regions.

Example: Global Locksets. One possible instantiation of regions uses locksets. If we assume that there is a global set of locks L, we can define $R = \mathcal{P}(L)$, which means that a race equivalence region is simply a set of locks, and the program points in the region are those program points at which the region's locks are held.

Consider the buggy example from Figure 2. The Reg map is:

$$Reg(p) = \begin{cases} \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{P3}, \texttt{P4}, \texttt{P8}, \texttt{P9}\} \\ \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ \emptyset & \text{otherwise} \end{cases}$$

which captures the set of program points where `buf_lock` is held. The $RacyRead$ function is defined as:

$$\begin{aligned} RacyRead(r, l) \quad = \quad & (l = \texttt{px}\rightarrow\texttt{data} \wedge \texttt{buf\_lock} \notin r) \vee \\ & (l = \texttt{cx}\rightarrow\texttt{data} \wedge \texttt{buf\_lock} \notin r) \end{aligned}$$

which captures the fact that an access of `px→data` in `Producer` or `cx→data` in `Consumer` is racy at any point where the lock `buf_lock` is not held, and that all other accesses are non-racy. It is easy to check that the Reg and $RacyRead$ functions soundly approximate the possible races and pseudo-races. The adjusting at program point `P5` kills the fact $NonNull(\texttt{px}\rightarrow\texttt{data})$ as

$$RacyRead(Reg(\texttt{P5}), \texttt{px}\rightarrow\texttt{data}) = true$$

In the correct version from Figure 1, the absence of the unsafe lock operations changes the Reg map to:

$$Reg(p) = \begin{cases} \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{P3}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ \emptyset & \text{otherwise} \end{cases}$$

reflecting the fact that in this program, `buf_lock` is held throughout from `P3` to `P9`. The $RacyRead$ function remains the same as before, as the synchronization discipline is unchanged: as in Figure 1, `buf_lock` is held at all points where the buffer cells are written, namely `P3`, `P8`, and `C5`. In the fixed program, the adjusting at $p \in \{\texttt{P5}, \texttt{P6}, \texttt{P8}\}$ does not kill the fact $NonNull(\texttt{px}{\rightarrow}\texttt{data})$, as for each of these p

$$RacyRead(Reg(p), \texttt{px}{\rightarrow}\texttt{data}) = false$$

Example: Predicates. Another possible instantiation of regions consists of having R be a set of predicates $Pred$. A race equivalence region is then a predicate from $Pred$, and the set of program points in the region are those program points at which the predicate holds. The predicate instantiation is more general than the lockset instantiation because we can encode a set of locks as a predicate stating that all the locks in the set are held.

Recall the version of the `Producer-Consumer` program shown in Figure 3, where the synchronization is performed via a `flag` field and not explicitly declared locks. As shown in [11], one can generalize race regions to an *access predicate* describing the thread's state. For the example in Figure 3, the Reg map is:

$$Reg(p) = \begin{cases} \texttt{px}{\rightarrow}\texttt{flag} = 0 & \text{if } p \in \{\texttt{P3}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ \texttt{cx}{\rightarrow}\texttt{flag} \neq 0 & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ true & \text{otherwise} \end{cases}$$

The $RacyRead$ function is defined as:

$$\begin{aligned} RacyRead(\varphi, l) = \quad & (l = \texttt{px}{\rightarrow}\texttt{data} \land \varphi \not\Rightarrow \texttt{px}{\rightarrow}\texttt{flag} = 0) \lor \\ & (l = \texttt{cx}{\rightarrow}\texttt{data} \land \varphi \not\Rightarrow \texttt{cx}{\rightarrow}\texttt{flag} \neq 0) \end{aligned}$$

Together, Reg and $RacyRead$ capture the intuition that a read of $\texttt{px}{\rightarrow}\texttt{data}$ in `Producer` (respectively $\texttt{cx}{\rightarrow}\texttt{data}$ in `Consumer`) is racy at any point where the $\texttt{px}{\rightarrow}\texttt{flag}$ is zero (respectively $\texttt{cx}{\rightarrow}\texttt{flag}$ is zero).

3.3 Interprocedural Framework

The RADAR framework presented so far does not take function calls into account. To understand how function calls affect our basic framework, consider the example from Figure 4, which is a version of the `Producer-Consumer` example where there is a call to a function `foo` right before the increment of `perf_ctr`.

Let us assume that `foo` itself does not modify $\texttt{px}{\rightarrow}\texttt{data}$, and that the sequential dataflow analysis uses a simple "modifies" analysis to determine this. As a result, the sequential dataflow analysis is able to propagate $NonNull(\texttt{px}{\rightarrow}\texttt{data})$ from `P4` to `P5` (that is to say, through the call to `foo`). However, in the face of concurrency, even if `foo` itself does not modify $\texttt{px}{\rightarrow}\texttt{data}$, a call to `foo` while other threads are running can in fact lead to $\texttt{px}{\rightarrow}\texttt{data}$ being modified. In particular, calling `foo` has the effect of unlocking `buf_lock` and then re-locking it, which gives concurrent threads an opportunity to modify $\texttt{px}{\rightarrow}\texttt{data}$. As a result, the adjusted flow function needs to kill $NonNull(\texttt{px}{\rightarrow}\texttt{data})$ at `P5`, as well as any dataflow information about $\texttt{px}{\rightarrow}\texttt{data}$.

Unfortunately, with the definition of $ThreadKill$ given so far, the adjusted flow function would not do this. In particular, the adjusted flow function would ask $ThreadKill$ if $\texttt{px}{\rightarrow}\texttt{data}$ could be written concurrently at the program point right after `foo` returns (which is `P5`). $ThreadKill$ in turn would ask the race detection engine if a read of $\texttt{px}{\rightarrow}\texttt{data}$ would cause a race at `P5`. The race detection engine would answer back saying "no race" since by the time `foo` returns, the lock protecting $\texttt{px}{\rightarrow}\texttt{data}$ would already have been re-acquired. As a result, the information about $\texttt{px}{\rightarrow}\texttt{data}$ being non-null would incorrectly survive the adjusting process.

Adjusted Analysis	`buffer_list *bufs;` `lock buf_lock;` `int perf_ctr;`
	`thread producer1(){`
ϕ	`P0: px = bufs;`
ϕ	`P1: while (px != NULL){`
px	`P2:   lock(buf_lock);`
px	`P3:   px->data = new();`
px->data,px	`P4:   foo();`
~~px->data~~,px	`P5:   perf_ctr++;`
~~px->data~~,px	`P6:   t=produce();`
~~px->data~~,px	`P8:   *px->data = t;`
~~px->data~~,px	`P9:   unlock(buf_lock);`
~~px->data~~,px	`PA:   px = px->next;`
	`}`
	`foo(){`
ϕ	`G0: unlock(buf_lock);`
ϕ	`//do work`
ϕ	`G1: lock(buf_lock);`
	`thread consumer1(){`
ϕ	`perf_ctr = 0;`
ϕ	`C0: cx = bufs;`
ϕ	`C1: while(cx != NULL){`
cx	`C2:   lock(buf_lock);`
cx	`C3:   if(cx->data != NULL){`
cx->data,cx	`C4:     consume(*cx->data);`
cx->data,cx	`C5:     cx->data = NULL;`
cx	`C6:     cx = cx->next;`
	`}`
ϕ	`C7:   unlock(buf_lock);`
	`}`

Figure 4. `Producer-Consumer` with function calls

The problem in the example above is that the execution of `foo` passes through a region that does not hold `buf_lock`, which allows concurrent threads to modify $\texttt{px}{\rightarrow}\texttt{data}$, and callers of `foo` must take this into account. More broadly, the problem is that the execution of a function can pass through a *variety* of race equivalence regions, and callers need a way to summarize the effect of having passed through all of the callee's regions.

Summary Regions. To address this problem, we add to RADAR a new function called $SumReg : CS \rightarrow R$, which returns for each call-site an *interprocedural summary region*. This call-site-specific summary region is meant to approximate the possible regions that the callee can go through when invoked at the given call-site. We denote by CS the set of all call-sites, and we define the call-site of a call node to be the CFG edge that immediately follows the call node (so that $CS \subseteq PPoint$).

Soundness. Intuitively, for soundness we require that for every lvalue l, if a read of l is racy at *some* program point transitively reachable during the call made at cs, then the summary region for the call-site cs must be a region that is racy for l. Thus, to formalize soundness in a context-sensitive manner, we extend the perfect race detection oracle to $RealRace : CS \times PPoint \times LVal \rightarrow Bool$, such that $RealRace(cs, p, l)$ returns $true$ exactly when there is an execution in which a read of l at p while cs is on the callstack would cause a race. Let cs^* be the set of program points in the function being called at call-site cs and any of its transitive callees. The soundness requirement for $SumReg$ can be formally stated as:

$$\begin{aligned} & \forall cs \in CS, l \in LVal \; . \\ & \quad [\exists p \in cs^* \; . \; RealRace(cs, p, l)] \Rightarrow \\ & \qquad RacyRead(SumReg(cs), l) \end{aligned} \tag{3}$$

Having defined $SumReg$ and its soundness property, we can now define the $ThreadKill$ function as follows:

$$ThreadKill(p,l) = \quad RacyRead(Reg(p),l) \vee (RacyRead(SumReg(p),l) \wedge p \in CS)$$

Instantiation Requirements: To instantiate the interprocedural RADAR framework, one needs to provide:

1. A **race detection engine** $RacyRead : R \times LVal \rightarrow Bool$
2. A **region map** $Reg : PPoint \rightarrow R$
3. A **summary map** $SumReg : CS \rightarrow R$

such that $RacyRead$ and Reg satisfy property (2), and $SumReg$ satisfies property (3). We now go through the same two instantiation from Section 3.2, and show how $SumReg$ can be defined.

Example: Global Locksets. If we instantiate regions as locksets over a global set of locks L, the summary of a function is the intersection of all the locksets that the function goes through. As a result, in the example from Figure 4, the $SumReg$ function is defined as:

$$SumReg(\texttt{P5}) = \emptyset$$

since the unlock in `foo` causes the lockset to be empty at `G1`, and thus the intersection of all locksets in `foo` is empty. The Reg and $RacyRead$ functions are the same as in Section 3.2.

With these definitions, the adjusting process now correctly kills the fact $NonNull(\texttt{px}{\rightarrow}\texttt{data})$ at `P5` , since, even though

$$RacyRead(Reg(\texttt{P5}), \texttt{px}{\rightarrow}\texttt{data}) = false$$

we have

$$RacyRead(SumReg(\texttt{P5}), \texttt{px}{\rightarrow}\texttt{data}) = true$$

which, combined with $\texttt{P5} \in CS$, means

$$ThreadKill(\texttt{P5}, \texttt{px}{\rightarrow}\texttt{data}) = true$$

Example: Predicates. If we use a set $Pred$ of predicates for the race equivalence regions, the summary of a function is the disjunction of all the predicates in the function.

4. An implementation of RADAR using RELAY

We have implemented an instantiation of the RADAR framework using an existing scalable race detection engine called RELAY [29]. We call this instantiation RADAR(RELAY). We first give a brief overview of RELAY, and then we describe RADAR(RELAY).

4.1 Overview of RELAY

RELAY is a lockset-based static race detection tool that scales to code bases as large as the Linux kernel (5 million lines of C code). RELAY works by traversing the call graph in a bottom-up manner, analyzing each function in isolation.

Assumptions. We assume that programmers can create threads, acquire locks and release locks by calling functions from specific thread libraries (*e.g.* pthreads). Our techniques work even when matching lock and unlock calls appear in different procedures, as is often the case in many of our benchmarks. RELAY is also summary-based, so the programmer can provide summaries that describe the effect of missing procedures on the lockset. Finally, when analyzing a library in isolation, we conservatively assume that each API function can be called by separate threads created by clients.

Relative Locksets. For each function, RELAY first performs a lockset analysis on the function using a *relative lockset* representation. A relative lockset at a program point is a disjoint pair of locksets (L_+, L_-) (respectively called the positive and negative locksets), which encodes the *difference* between the locks held at the given program point and the locks held at the function entry point. In particular, the set L_+ represents the locks that have definitely been acquired since the beginning of the function, and the set L_- represents the locks that may have been released since the beginning of the function. Notice that L_+ is a must set and that L_- is a may set.

Locks can be represented as lvalues, and so $L_+ \subseteq LVal$ and $L_- \subseteq LVal$. We denote by $\mathbb{L} = \mathcal{P}(LVal) \times \mathcal{P}(LVal)$ the set of all relative locksets. The set of relative locksets form a lattice $(\mathbb{L}, \sqsubseteq, \top, \bot, \sqcap, \sqcup)$ defined as:

- $\bot = (LVal, \emptyset)$, $\top = (\emptyset, LVal)$
- $(L_+, L_-) \sqsubseteq (L'_+, L'_-)$ iff $L'_+ \subseteq L_+ \wedge L_- \subseteq L'_-$
- $(L_+, L_-) \sqcup (L'_+, L'_-) = (L_+ \cap L'_+, L_- \cup L'_-)$.
- $(L_+, L_-) \sqcap (L'_+, L'_-) = (L_+ \cup L'_+, L_- \cap L'_-)$

Relative Lockset Summaries. The relative lockset computed for the exit point of the function is stored in a context-sensitive *relative lockset summary*. This summary soundly approximates the effect of the function on the thread's lockset just *before* calling the function and is used to update locksets at call-sites.

Guarded Access Summaries. For each function, RELAY also computes the set of *guarded accesses* in the function. A *guarded access* is a triple of an lvalue, the relative lockset at the program location where the access takes place, and the *kind* of access, either a read or a write. The set of guarded accesses of a function is the set of triples corresponding to accesses that may occur during the execution of the function. RELAY stores this set as a *guarded access summary* for the function.

RELAY computes these two summaries (the *relative lockset summary* and the *guarded access summary*) in a bottom-up manner, plugging in the summaries of the *callees* at call-sites to compute the relative lockset and guarded access summaries of the *callers*. Once summaries have been computed for all functions, RELAY looks at the guarded accesses at thread entry points. For each pair of guarded accesses whose lvalues may be aliased, if the intersection of the positive locksets is empty, RELAY reports a race.

4.2 Putting RELAY into RADAR

We now show how to instantiate the three functions Reg, $RacyRead$, and $SumReg$ using RELAY. The result is an instantiation of RADAR using RELAY, namely RADAR(RELAY).

Region map. We define $R = \mathbb{F} \times \mathbb{L}$, where $\mathbb{F}$ is the set of function identifiers. Given a program point p, $Reg(p)$ returns $(g, (L_+, L_-))$, where g is the function to which p belongs, and (L_+, L_-) is the relative lockset computed at p by the RELAY lockset analysis.

Summary map. To define the $SumReg$ function, we first define a helper function $AllUnlocks$. Intuitively, the set $AllUnlocks(cs)$ represents an overapproximation of the set of locks that could possibly be released by performing the call at cs. In particular, given a call-site cs, $AllUnlocks(cs)$ computes the union of all the L_- sets in the function being called at cs, and then converts this set into the caller context. The conversion consists of replacing the callee's formals that occur in the locksets with the parameters passed in at the call-site.

Given a call-site $cs \in CS$ where function h calls g, and given that (L_+, L_-) is the relative lockset computed by RELAY for the program point right before the call to g at cs, then $SumReg(cs)$ is defined as follows:

$$SumReg(cs) = (h, (L_+ - AllUnlocks(cs), L_- \cup AllUnlocks(cs)))$$

Essentially, $SumReg$ subtracts the $AllUnlocks$ set from the locks that were held before the call is made. The result of this subtraction is a conservative approximation of the set of locks that are guaranteed to remain locked *at all points* during the call.

Race detection engine. Given a region $r = (g, (L_+, L_-))$ and an lvalue l, $\mathit{RacyRead}(r, l)$ conceptually runs a full RELAY analysis bottom-up, except that when it analyzes function g, it inserts an additional guarded access to the guarded access set. The additional guarded access is the triple $(l, (L_+, L_-), \mathit{read})$, indicating that we are simulating a read of l with a lockset of (L_+, L_-). The $\mathit{RacyRead}(r, l)$ function returns true if and only if, after being propagated up to the thread roots, this pseudo-read leads to a RELAY race warning.

This conceptual description of $\mathit{RacyRead}(r, l)$ is not how we implement it, since each call to $\mathit{RacyRead}$ would lead to an entire RELAY bottom-up analysis of the program. Instead, we structure the execution of RADAR(RELAY) into the following four passes, two of which are the RELAY bottom-up analysis.

- **First pass.** RADAR(RELAY) runs the bottom-up RELAY analysis to compute the relative locksets at each program point, and hence, the race equivalence regions.
- **Second pass.** RADAR(RELAY) runs the sequential analysis on the entire program with a $\mathit{RacyRead}$ function that returns false all the time. This has the effect of running the sequential analysis without any adjusting, but it allows RADAR(RELAY) to collect the parameters of all the $\mathit{RacyRead}$ queries into a set S of (r, l) pairs. Since the sequential analysis computes a superset of the facts computed by the adjusted analysis, the set S is a superset of the queries that the adjusted analysis will make.
- **Third pass.** RADAR(RELAY) then runs the bottom-up RELAY analysis again to insert pseudo-accesses. In particular, when analyzing a function g, for each pair $(r, l) \in S$ where $r = (g, (L_+, L_-))$, RADAR(RELAY) adds the guarded access $(l, (L_+, L_-), \mathit{read})$ to the guarded access summary of g. RADAR(RELAY) uses the results of the second RELAY run to build a map $\mathit{RelayResults} : S \to \mathit{Bool}$. Given $(r, l) \in S$, $\mathit{RelayResults}(r, l)$ returns whether or not the pseudo-read inserted for (r, l) caused a RELAY race warning.
- **Fourth pass.** Finally, RADAR(RELAY) runs the sequential analysis again, but this time performs the adjusting process. In particular, RADAR(RELAY) uses the $\mathit{RelayResults}$ map computed in the second pass to answer the $\mathit{RacyRead}$ queries.

Example: Relative Locksets. We now illustrate how RADAR(RELAY) would analyze the program in Figure 4.

- **First pass.** RELAY computes the Reg map where $\mathit{Reg}(p)$ is:

$$\begin{cases} (\texttt{Producer}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{P0}\} \\ (\texttt{Producer}, (\{\texttt{buf\_lock}\}, \emptyset)) & \text{if } p \in \{\texttt{P3}, \texttt{P4}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ (\texttt{Producer}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{P1}, \texttt{P2}, \texttt{PA}\} \\ (\texttt{Consumer}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{C0}\} \\ (\texttt{Consumer}, (\{\texttt{buf\_lock}\}, \emptyset)) & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ (\texttt{Consumer}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{C1}, \texttt{C2}\} \\ (\texttt{foo}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{G0}\} \\ (\texttt{foo}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{G1}\} \end{cases}$$

  Notice that both locksets are empty at the first point in each function meaning the lockset is trivially the same as at the entry point. Using the above Reg map, RADAR(RELAY) determines:

$$\mathit{AllUnlocks}(\texttt{P5}) = \{\texttt{buf\_lock}\}$$

  as the `buf_lock` is released inside `foo`. Thus,

$$\mathit{SumReg}(\texttt{P5}) = (\texttt{Producer}, (\emptyset, \{\texttt{buf\_lock}\}))$$

- **Second pass.** In the second pass, RADAR(RELAY) runs a sequential non-null analysis which generates the flow facts shown on the left in Figure 4, *including* the facts crossed out by a line. Using these facts, RADAR(RELAY) computes the superset S as the set of tuples $\{(\mathit{Reg}(p), l) \mid \mathit{NonNull}(l) \text{ at } p\}$.
- **Third pass.** RADAR(RELAY) then inserts pseudo-reads corresponding to the queries S generated above, and builds the map $\mathit{RelayResults}$ which yields the following $\mathit{RacyRead}$ map:

$$\begin{aligned} &\mathit{RacyRead}((g, (L_+, L_-)), l) = \\ &\qquad (l = \texttt{px}{\to}\texttt{data} \wedge \texttt{buf\_lock} \notin L_+) \\ &\qquad \vee (l = \texttt{cx}{\to}\texttt{data} \wedge \texttt{buf\_lock} \notin L_+) \end{aligned}$$

- **Fourth pass.** When the adjusted sequential analysis is performed, the fact $\mathit{NonNull}(\texttt{px}{\to}\texttt{data})$ gets killed at `P5` since the summary region at that call-site does not include `buf_lock` in L_+. The result is shown on the left in Figure 4 – the dataflow solution includes only the facts that are *not* crossed out.

5. Evaluation

To evaluate the RADAR framework, we have designed a *sequential* null-dereference analysis, which we describe first. We then present three instantiations of the adjusting framework to convert the analysis into *multithreaded* versions of the analysis, and we compare the results of each on three benchmarks: the Apache web server (approximately 130,000 lines of code), the OpenSSL library (210,000 lines of code) and part of the Linux kernel (830,000 lines of code).

5.1 Sequential Null-Dereference Analysis

We describe the sequential null-dereference analysis in two steps: we start with an intraprocedural analysis, and then we extend it to an interprocedural one.

Intraprocedural Analysis. To describe the analysis, we define the set of dataflow facts $\mathit{DataflowFact}$ (whose powerset is the domain D of the analysis) and the sequential flow function F.

- *Dataflow Facts:* The dataflow facts used by the intraprocedural sequential analysis are $\mathit{DataflowFact} \equiv \{\mathit{NonNull}(l) \mid l \in \mathit{LVal}\}$. Intuitively, if the analysis computes $\mathit{NonNull}(l)$ at a program point, we conclude that the lvalue l is *always* non-null at that program point. Thus, the dereference of an lvalue l is safe at a node n if $\mathit{NonNull}(l)$ is in the set of facts computed for the program point right before n.
- *Flow Function:* Recall that the sequential flow function F takes as input a CFG node n, a set of input facts d, and an outgoing program point p for which to compute a set of output facts.

$$\begin{aligned} &\mathit{NonNull}(l') \in F(\text{“}\texttt{if } e\text{”}, d, p) \textbf{ if} \\ &\qquad \mathit{NonNull}(l') \in d \\ &\qquad \textbf{or } [p = \mathit{true\_out} \textbf{ and } e = (l' \,!\!= \texttt{NULL})] \\ &\mathit{NonNull}(l') \in F(\text{“}l := e\text{”}, d, \_) \textbf{ if} \\ &\qquad [\mathit{NonNull}(l') \in d \textbf{ and } \neg \mathit{Aliased}(l, l')] \\ &\qquad \textbf{or } [l = l' \textbf{ and } \mathit{NonNull}(e) \in d] \\ &\qquad \textbf{or } [l = l' \textbf{ and } e = \texttt{malloc}(\ldots)] \\ &\qquad \textbf{or } [l = l' \textbf{ and } e = \&\ldots] \end{aligned}$$

Interprocedural Analysis. In the presence of multiple procedures, the set of facts and the flow function are updated to summarize the effects of procedure calls, but the safety check remains the same.

- *Dataflow Facts:* The set of dataflow facts includes the facts used for the intraprocedural analysis, as well as facts $\mathit{NotMod}(l)$ that state whether a given lvalue l has *not been modified* during the course of executing the function call. The $NotMod()$ facts are a *must* version of the *may* information captured via the usual *may-modify* analysis.

- *Flow Function:* The flow function must be updated to handle the $NotMod()$ facts and function calls. We define the *summary* of a function the flow facts that hold at its exit point. We use a function $Norm : CS \rightarrow D$, which takes a call-site cs as input, looks up the current summary for the function being called, and normalizes the flow facts by renaming formals to be in terms of the parameters passed in at the call-site cs. Using the summaries, we get a new flow function, where in addition to the rules described above for the intraprocedural case, we have rules to propagate the non-modified facts, and rules that propagate the non-null facts at procedure call-sites:

$$
\begin{aligned}
&NotMod(l') \in F(\text{“}l := e\text{”}, d, \_) \textbf{ if} \\
&\qquad\qquad NotMod(l') \in d \textbf{ and } \neg Aliased(l, l') \\
&NotMod(l') \in F(\text{“}\texttt{if } e\text{”}, d, \_) \textbf{ if} \\
&\qquad\qquad NotMod(l') \in d \\
&NotMod(l') \in F(\text{“}\texttt{call...}\text{”}, d, cs) \textbf{ if} \\
&\qquad\qquad NotMod(l') \in d \textbf{ and } NotMod(l') \in Norm(cs) \\
&NonNull(l') \in F(\text{“}\texttt{call...}\text{”}, d, cs) \textbf{ if} \\
&\qquad\qquad [NonNull(l') \in d \textbf{ and } NotMod(l') \in Norm(cs)] \\
&\qquad\quad \textbf{or } NonNull(l') \in Norm(cs)
\end{aligned}
$$

5.2 Instantiations

In addition to the non-null sequential analysis, RADAR requires a black box to answer $RacyRead$ queries. We present four implementations of this component.

Steensgaard-based instantiation. The simplest and least precise instantiation we consider is based on Steensgaard's pointer analysis [26], and we call this instantation RADAR(STEENS). For this instantiation, $RacyRead(r, l)$ ignores the region r it is passed and returns true if l is reachable from a global in the Steensgaard's points-to graph. This matches our intuition that lvalues that cannot be reached from globals cannot be shared, and thus cannot be racy.

RELAY-based instantiations. We have already described the RELAY-based instantiation of RADAR in Section 4. However, for the purpose of better understanding where the precision of RADAR(RELAY) is coming from, we separate RADAR(RELAY) into two instantiations based on the observation that RELAY can prove the absence of a race in two different ways: (1) by showing that the two lvalues do not alias; and (2) if they can alias, by showing that the intersection of the locksets is non-empty.

To better understand how these two different ways of showing the absence of a race contribute to RADAR(RELAY), we separate the instantiation into two parts, RADAR(RELAY$_{\neg L}$) and RADAR(RELAY). We have already seen the latter; it is just as described in Section 4. The former is a version of RADAR(RELAY) where we change the Reg map to always return $\top$, which represents the empty set of locks. This modification simulates a version of RELAY that only answers race queries based on possible aliasing relationships but not on locksets.

Optimistic instantiation. The last instantation we consider is the most optimistic possible: the one where $RacyRead$ always returns $false$. We call this version SEQ because it is equivalent to the sequential analysis without any adjusting. Although this instantiation of RADAR is unsound, it establishes an upper bound on how well any adjusted analysis can do.

Each one of these four instantiations – RADAR(STEENS), RADAR(RELAY$_{\neg L}$), RADAR(RELAY), and SEQ – is strictly more precise than the previous, with the last one being unsound.

5.3 Results

Figure 5 shows the number of dereferences proven safe, as a fraction of all dereferences, by each of the four RADAR-adjusted analyses. As expected, the size of each bar grows from left to right, indicating that each analysis is more precise.

The first thing to notice is that for each benchmark the sequential analysis (the fourth bar in each cluster) can prove only a small percentage of dereferences safe, between 42% and 68%. Thus, no matter how precise the adjusting process, the resulting multi-threaded analysis will not be able to prove a majority of dereferences.

The imprecision in the sequential non-null analysis is mostly due to imprecision in analyzing the heap. The alias analysis we use merges many of the lvalues on the heap into "blob" nodes, thus losing precision for heap-allocated variables. Previous null-pointer analyses [4] have also found that heap structures are hard to analyze precisely and lead to many false-positives when performing null-dereference checks. To factor this degree of imprecision out of our experiments, we plot in Figure 6 the percentages of safe dereferences to pointers *not* including those in blob nodes. The percentage of all non-blobby dereferences is approximately 52% for Apache, 76% for SSL, and 71% for Linux. Considering these remaining dereferences, the sequential analysis is able to prove the safety of the majority of dereferences on each benchmark (again the fourth bar in each cluster). This demonstrates that, aside from the issue of precise heap analysis, this sequential analysis is a reasonable and practical one on which to test RADAR.

Recall that the sequential analysis is unsound in the concurrent setting because it assumes lvalues cannot be modified by concurrent threads, so it may miss actual null-pointer dereferences. Nevertheless, because we cannot know what an oracle would provide as the "correct" answer for adjusting, we use SEQ as an upper bound; the other three analyses, as well as the oracle, cannot do any better. At the other end of the spectrum is RADAR(STEENS), the least precise of the analyses. We included this Steensgaard-based approach in our evaluation because it can easily be implemented in a compiler or program analyzer that needs to be sound but is not concerned with being extremely precise. We therefore use RADAR(STEENS) as a baseline for comparison.

We now evaluate how the RADAR(RELAY$_{\neg L}$) and RADAR(RELAY) instantiations compare to the RADAR(STEENS) lower bound and the SEQ upper bound. We consider what percentage of this *gap* – the difference between the results of SEQ and RADAR(STEENS) – is bridged by the other two analyses. Keep in mind that because SEQ is an unsound overapproximation, the real gap – the difference between a perfect oracle and RADAR(STEENS) – may be smaller than the gap we consider. Thus, the percentages we report are in fact lower bounds on how much of the real gap we bridge. Figure 7 summarizes these results. On average, RADAR(RELAY$_{\neg L}$) bridges 55.0% of the gap on non-blobby dereferences while RADAR(RELAY) bridges 58.4%.

The results on Linux are what we would expect: each analysis is incrementally better than the previous one. From left to right, each analysis incorporates, in the following order, a simple alias analysis, a more precise thread sharing analysis, and a lockset analysis. As a result, each analysis better captures concurrency interactions in the program. This leads to more precise race detection, which is ultimately the factor that determines RADAR's effectiveness.

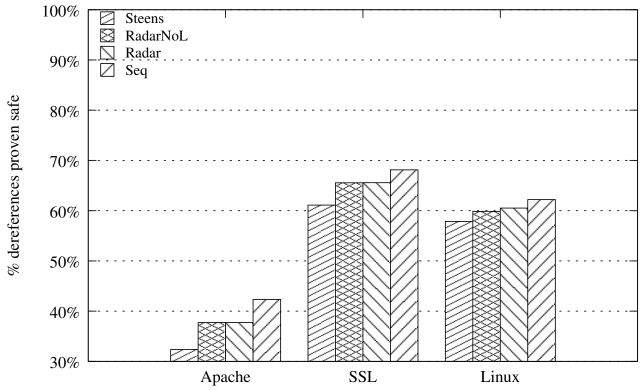

Figure 5. Percentage of all dereferences proven safe by each instantiation.

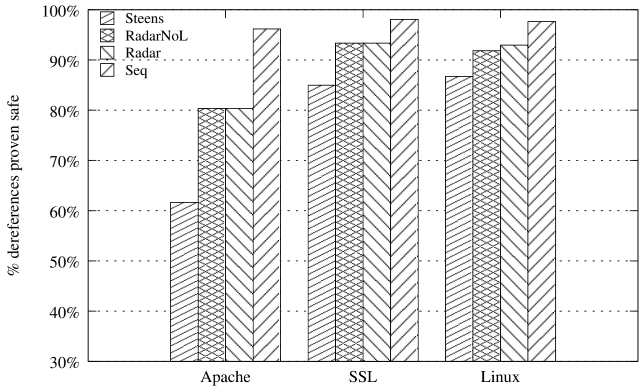

Figure 6. Percentage of non-blobby dereferences proven safe by each instantiation.

	w/ blobs		no blobs	
	no locks	w/ locks	no locks	w/ locks
Apache	53.9	53.9	54.2	54.2
SSL	63.3	63.4	63.9	64.0
Linux	46.4	61.2	46.9	57.0
Average	54.5	59.5	55.0	58.4

Figure 7. Percent of the gap bridged.

The results for Apache and SSL, however, are different from the Linux ones. In particular, RADAR(RELAY$_{\neg L}$) is just as effective as RADAR(RELAY) on Apache and almost as effective as RADAR(RELAY) on SSL.

To better understand the implications of these results, recall how RADAR(RELAY) and RADAR(RELAY$_{\neg L}$) differ: RADAR(RELAY) uses the full version of RELAY, whereas RADAR(RELAY$_{\neg L}$) uses a version of RELAY that answers race queries based on possible aliasing relationships, but not on locksets. The fact that RADAR(RELAY$_{\neg L}$) is nearly as precise as RADAR(RELAY) indicates that in many of the cases arising in our non-null analysis, the lvalue being adjusted is simply not shared. This in turn is an indication that a more precise alias analysis in the RELAY race detection engine could drastically improve the precision of RADAR(RELAY).

Overall, our experiments on RADAR(RELAY) demonstrate the precision and scalability of RADAR. Running RADAR(RELAY) on a single machine took 1 hour on SSL, 4 hours on Apache, and 12 hours on Linux. But because the callgraph of Linux is embarassingly parallel, the implementation of RADAR(RELAY) can easily be parallelized in the same way as RELAY to run much faster on a cluster of nodes. In each of the test cases, RADAR(RELAY) was able to bridge a sizable portion of the gap between optimistic and conservative conconcurrent non-null analyses, while still producing a sound result.

6. Related Work

Race Analyses. Java's native support for threads and its syntactically scoped locks enable many techniques for detecting and proving the absence of races. These include type systems encoding lockset information [6], and extended with ownership [1]. Another approach is to statically approximate the happens-before relation [28]. Escape analyses have been proposed as a simpler way of detecting shared objects and removing synchronization [23]. A recent line of work [17] shows how to combine nested locking with cloning-based context-sensitivity to improve the accuracy of lockset computations. The results of [17] show that even in Java benchmarks, the large majority potential races are eliminated by a precise sharing analysis. Analyses for C programs must cope with the unstructured use of locks and thread creation, which make flow- or context-insensitivity very imprecise. [5] works by computing summaries in a top-down manner, but prunes summaries to scale, [20] uses a constraint based technique to compute *correlations* that describe the locking protocol.

Dataflow Analysis. There are frameworks for intraprocedural dataflow analysis of concurrent programs that use par-begin and par-end constructs nested within functions. These frameworks work by building a parallel flow graph (PFG) (a control flow graph with concurrency nodes). The dataflow analysis is lifted by extending the flow equations to handle fork, join and the communication between threads. Examples include a reaching definitions analysis [9] and bit-vector analyses [13]. Intraprocedural representation-based approaches include [24] which uses cobegin/end and wait/notify constructs to build a Parallel Program Graph, analagous to the PDG, and [15] which describes a Concurrent SSA (CSSA) representation which enables subsequent optimizations. The pointer analysis of [22] is interprocedural but matching par-begin and par-end constructs must be in the same function. Recently, [3] proposed an analysis framework for optimizing embedded programs written in NESC [8]. This framework is tailored to NESC's interrupt-based concurrency and explicit `atomic` sections.

All the above frameworks are more precise than our framework in which facts can only be *killed* by concurrent interactions. In contrast, these frameworks exploit specific concurrency constructs to also allow new facts to be *generated* by concurrent interactions. However, RADAR is more general, as it is independent of the underlying concurrency constructs, requiring only that a race detector exists for the constructs. For example, none of these approaches could be applied to our thread-based benchmarks.

Model Checking. Model checkers explore all interleavings to verify abitrary safety properties, and so they can be used to encode dataflow analyses [25]. Flavers [2] is a finite-state property checker that employs conservative state and interleaving reductions, *e.g.* a may-happen-in-parallel dataflow analysis ([18]) that conservatively prunes interleavings. Even with techniques like these and others like partial-order and symmetry reduction that mitigate the effect of combinatorial explosion in interleavings, model checking has only been shown to scale to relatively small code bases. A technique related to RADAR is the *thread-modular* approach, proposed in [19, 12] which requires that users provide annotations describing when *other* threads can modify shared state. Model checking can be used to infer the annotations [7, 11], but these techniques do not scale. If the programs include recursive procedures, model checking (and hence, "exact" dataflow analysis in the sense of computing MOP solutions) is undecidable [21].

In contrast to the above, the principal benefit of our framework RADAR is that it is not tied to any particular concurrency constructs or structure, as all reasoning about concurrency is folded into the race detection engine. This allows RADAR to switch between race detection engines to explore the tradeoff between precision and scalability of the dataflow analysis. Moreover, RADAR enables a finer view of concurrency by preserving facts that are not killed by other threads, without exploring interleavings caused by irrelevant atomicity breaks as in Figure 1.

7. Conclusions and Future Work

We have presented a framework called RADAR for converting a sequential dataflow analysis into a concurrent one using a race detection engine as a black box. The main benefit of our approach is that it cleanly separates the part of the analysis that deals with concurrency, the race detection engine, from the rest of the analysis. With this separation in place, the race detection engine can be fine-tuned to improve its precision without changing any of the client analyses. As a result, RADAR provides a framework that allows the precision with which concurrency is analyzed to be easily tuned. Our experiments show that the framework scales, and for a particular analysis, achieves good precision with respect to some upper and lower bounds. Our experience also shows that the precision of the overall concurrent analysis depends on the precision of the underlying alias, escape, sequential and race analyses. We have identified two lines of future work that will, in combination, lead to the understanding and addressing of these issues.

The first direction is to apply the adjusting approach to lift a variety of previously developed sequential analyses to the concurrent setting. Examples include array bounds checking analyses [27, 10], other kinds of null-dereference analyses [4], and analyses that guide compiler optimizations, such as reaching definitions, available expressions, and live variables.

The second direction is to explore precise and scalable race detection techniques for other concurrency constructs. Improvements will likely involve a combination of better alias and escape analysis, better inter-thread flow analysis to handle system calls like `wait` and `notify`, path-sensitive analysis to handle flag-based synchronization, and possibly programmer-supplied annotations to help with cases that are too difficult to analyze automatically.

Adjusting a wide variety of sequential analyses will allow us to tune the precision of the race detector using actual facts deduced while analyzing real systems for a variety of properties. We believe that the generated concurrent analyses can lead to an empirical understanding of the concurrency idioms used in real programs. These patterns can then be used to iteratively tune the precision of the race detector, leading to a variety of scalable and precise concurrent program analyses and optimizations.

References

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[2] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[3] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *LCTES*, pages 44–53, 2006.

[4] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, 2007.

[5] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252. ACM Press, 2003.

[6] C. Flanagan and S.N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232. ACM, 2000.

[7] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, LNCS 2648, pages 213–224. Springer, 2003.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003*, pages 1–11. ACM, 2003.

[9] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPoPP*, San Diego, CA, 1993.

[10] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 129–144. ACM, 2006.

[11] T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–12. ACM, 2004.

[12] C.B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[13] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.

[14] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[15] J. Lee, D.A. Padua, and S.P. Midkiff. Basic compiler algorithms for parallel programs. In *PPOPP*, pages 1–12, 1999.

[16] R.J. Lipton. Reduction: A new method of proving properties of systems of processes. In *POPL*, pages 78–86, 1975.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[18] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing *mhp* information for concurrent java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.

[19] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[20] P. Pratikakis, J.S. Foster, and M.W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[21] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.

[22] R. Rugina and M.C. Rinard. Pointer analysis for multithreaded programs. In *PLDI*, pages 77–90, 1999.

[23] A. Salcianu and M.C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, pages 12–23, 2001.

[24] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, pages 94–113, 1997.

[25] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL*, pages 38–48. ACM, 1998.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.

[27] A. Venet and G.P. Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, pages 231–242, 2004.

[28] C. von Praun and T.R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[29] J. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *ESEC/FSE*. ACM, 2007.