RELAY: Static Race Detection on Millions of Lines of Code *

Jan Wen Voung
UC San Diego
jvoung@cs.ucsd.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

ABSTRACT

Data races occur when multiple threads are about to access the same piece of memory, and at least one of those accesses is a write. Such races can lead to hard-to-reproduce bugs that are time consuming to debug and fix. We present RELAY, a static and scalable race detection analysis in which unsoundness is modularized to a few sources. We describe the analysis and results from our experiments using RELAY to find data races in the Linux kernel, which includes about 4.5 million lines of code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Verification

1. INTRODUCTION

Data races occur when multiple threads are about to access the same piece of memory, and at least one of those accesses is a write. Such races can lead to hard-to-reproduce bugs that are time consuming to debug and fix, especially if the code base is large. Thus, to build large, reliable concurrent programs, we require a race detection algorithm that is (1) *static*, in that it runs before the program is executed, (2) *sound*, in that it should guarantee that it finds all races, and (3) *scalable*, in that it should be effective on programs comprising millions of lines of code.

The above three goals have previously never been achieved all at once. In particular, while sound and static race detection techniques have proven to be effective, the largest programs they have ever been applied to are on the order of tens of thousands of lines of C code [23] and little over a hundred thousand lines of Java code [20]. Furthermore, while some static race detection algorithms run on millions of lines of code [8], they are extremely unsound, and therefore miss many errors.

*This research was supported in part by NSF Grants CCF-0644306, CCF-0644361, and the UCSD FWGrid Project, NSF Research Infrastructure Grant Number EIA-0303622.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

In this paper, we take a step towards achieving all three goals by developing RELAY, a static and scalable algorithm that can perform race detection on programs as large and complicated as the Linux kernel (which comprises 4.5 million lines of C code). In RELAY unsoundness is *modularized* to the following sources: (1) RELAY ignores reads and writes that occur inside blocks of assembly code; (2) RELAY does not handle corner cases of pointer arithmetic correctly; (3) RELAY uses a per-file alias analysis to optimistically resolve function pointers; and (4) RELAY uses a set of simple but unsound syntactic filters to categorize warnings into likely races (these filters are unsound in that they can remove real races). Because unsoundness in RELAY is modularized, we can easily revisit the above sources of unsoundness as we devise more precise analyses, with the hope that one day we will make RELAY entirely sound.

The standard mechanism for preventing data races is to ensure that for each shared lvalue, there exists a unique *lock* that is held whenever a thread accesses the lvalue. By ensuring that only one thread can hold a lock at any given time, we can ensure the absence of races. One family of algorithms for inferring whether such a lock exists is that based on computing *locksets*. These algorithms determine either statically [7, 8, 27] or dynamically [25, 5, 30] the set of locks held by the program at every program point. If the intersection of the locksets at each point the lvalue is accessed is non-empty then there are no races on the lvalue. If the intersection is empty, the analysis conservatively reports that there may be a race on the lvalue.

While static lockset based techniques have proven to be effective for race analysis, there are significant hurdles that must be crossed to scale them to millions of lines of systems code. First, it is difficult to tell which piece of memory a given operation will actually affect. The lvalue being accessed need not be in global scope — it may have been passed into the function as a parameter, and thus, the actual memory accessed can only be determined by a careful, calling-context-sensitive analysis. Second, for similar reasons, it is difficult to tell which locks are held at each point, as it is hard to tell exactly what locks are acquired and released by various operations as the locks may be derived from structures passed in as parameters. As a result, it becomes hard to tell if the set of locks held at two different accesses are the same, as the locks may have very different syntactic names. Third, in low-level systems code, the acquisition and release of locks is not syntactically nested (as is the case in Java). A lock may be acquired in one function, the access may happen in a second function, and the lock may be released in a third function. As a result, many modular type-based [10, 3] and flow-insensitive [20, 19] approaches cannot be applied in this setting, and instead a precise flow-sensitive approach is required.

The technical contribution of this paper is a technique for addressing the above limitations. In particular, we introduce the concept of a *relative lockset*, which describes the changes in the locks being held relative to the function entry point. These relative locksets allow us to summarize the behavior of a function independent of the calling context. For example, the summary of a function whose formal is `x` may say that the field `x->f` is accessed inside the function while holding all locks that were held on entry, plus `x->lock1`, and minus `x->lock2`. The information about this guarded access is not absolute — it is relative to the locks held at the entry point, which allows RELAY to plug the summary in while analyzing any callers.

This switch to relative locksets, rather than absolute locksets, is the key to scalability: relative locksets allows us to aggressively exploit modularity. In particular, RELAY analyzes functions in isolation to compute summaries that capture the behavior of a function for any calling context, and then it composes these summaries to determine whether races exist. This leads to a bottom-up context-sensitive analysis over the call graph that scales to programs as large as the Linux kernel. The modularity also enables easy parallelization. Because functions can be analyzed independently, we can run our analysis of the Linux kernel using the FWGrid cluster of machines [1] in about 5 hours, as opposed to the 72 hours it takes without parallelization.

This paper presents our work in two parts. We first present the RELAY algorithm for scalable race detection. In particular, Section 2 describes an overview of our algorithm through a simple example, while Section 3 describes the algorithm in detail. We then describe in Section 4 experimental results of running RELAY on the Linux kernel. Currently, RELAY generates warnings for races that can happen between *explicitly* created kernel threads and not user threads that have entered the kernel via system calls. While the summary generation scales to the entire kernel, a consequence of the unsound treatment of function pointers is that only 5291 out of 46872 functions of the kernel are reached from explicit thread creation sites. For these sites, RELAY produces 5022 race warnings. By applying a set of heuristic filters on the warnings to prune out likely false positives, we reduced the number of warnings down to 161, of which we categorized 31, finding 25 actual races (an 80 % race detection rate).

2. OVERVIEW

This section presents an overview of the RELAY race detection algorithm using the example code from Figure 1. The code in Figure 1 is a simplified version of two functions from the Linux kernel. On the left of Figure 1, the function `airo_read_stats` takes as input a single parameter `ai` that is a reference to a complex structure. On the right, the function `airo_thread` takes a pointer to a device structure `d`. The latter function is a *thread entry point*, *i.e.,* it is the first function that a new thread begins executing from. In the sequel, suppose that multiple threads can begin executing concurrently from this entry point. The structure that `d` refers to is a shared structure — it may be accessed outside the thread running `airo_thread`, and thus, it can possibly be accessed by multiple concurrently running threads. Similarly, because the `vals` array used on line 5 is declared to be global, it can be accessed by different threads, and so it is shared.

Locks are acquired and released by calling appropriate functions on the lock arguments. Thus, in the code from Figure 1, we can deduce (assuming that `airo_read_stats` is called only from within `airo_thread`), that the lock `d->priv->lock` is held whenever the lvalue `d->priv->pwr.ev` is accessed. Since line 1 is the only place the lvalue is accessed, we can therefore conclude there are no races on the lvalue. On the other hand, when `d->priv->stats.rx_p` is written to on line 5, there is no lock that is held, and so the write may cause a race, since two or more threads could simultaneously perform the write.

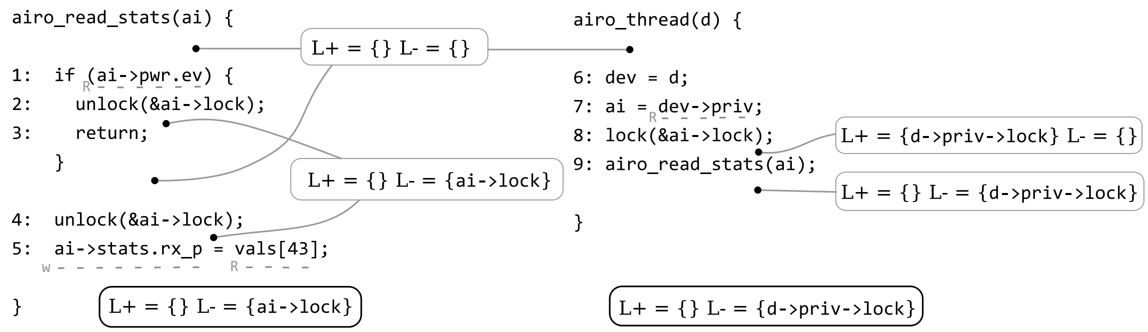
We devised a precise analysis that scales to millions of lines of code by aggressively exploiting modularity *i.e.,* by analyzing functions in isolation to compute summaries that capture the behavior of the function independent of the calling context, and then composing the summaries to determine whether races exist. In particular, our algorithm for race analysis is built using the following ingredients.

1. Relative Locksets: A *relative lockset* at a location is a disjoint pair of locksets (L_+, L_-) (resp. called the positive and negative locksets), which encodes the *difference* between the locks held at the given location and the locks held at the function entry location. Intuitively, the set L_+ is the set of additional locks that are definitely acquired on *all* executions from the entry to the location. The set L_- is the set of all locks that may have been released on *some* execution from the entry to the location. It is important to remember that L_+ is a *must* set and that L_- is a *may* set.

2. Guarded Accesses: 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 works by computing an overapproximation of the set of guarded accesses of each thread entry point. Once this set is computed, RELAY compares pairs of guarded accesses whose lvalues may be aliased. For each such pair, RELAY determines if the intersection of the positive locksets is empty, and if so, reports a race warning.

3. Function Summaries: To compute the guarded accesses for each thread entry point, RELAY builds the call graph and traverses it in a bottom up manner, computing the guarded accesses for each function along the way. To this end RELAY computes *two* summaries for each function. The first is a *relative lockset summary*, which is the relative lockset of the exit location of the function. This summary soundly approximates the effect the function has on the set of locks held by the thread just *before* calling the function. The second is a *guarded access summary*, which is a set of guarded accesses that includes the guarded accesses that may occur during the execution of the function. RELAY computes the summaries in a bottom-up manner, plugging in the summaries of the *callees* at callsites to compute the guarded access summaries and relative lockset of the *callers*.

4. Symbolic Execution: In order for a summary to capture the behavior of a function regardless of the calling context, the summary must be expressed in terms of the formals of the function, as well as globals. In this way, the summary can be instantiated at a call site by replacing formals in the summary with the actuals passed in at the call site, thus producing information in the caller's context. As an example, for the `airo_thread` function, we want to compute a summary stating that `d->priv->lock` is held when `d->priv->pwr.ev` is accessed, not that `ai->lock` is held when `ai->pwr.ev` is accessed. To build such summaries, one must re-express accesses inside a function in terms of the globals and the formals. To this end, RELAY performs an intra-procedural symbolic execution that maps each local lvalue to a value expressed in terms of the incoming values of formals, and the incoming values of globals. With appropriate join operators to handle merge nodes, we can handle loops while preserving termination. In the course of the intra-procedural analysis, whenever a function call is encountered, the guarded access summary of the callee is appended to the guarded accesses of the caller (after replacing the formals in



ai->pwr.ev	{}, {}	read
ai->stats.rx_p	{}, {ai->lock}	write
vals [0]	{},{ai->lock}	read
Lval	Relative Lockset	Kind

d->priv	{}, {}	read
d->priv->pwr.ev	{d->priv->lock}, {}	read
d->priv->stats.rx_p	{}, {d->priv->lock}	write
vals [0]	{}, {d->priv->lock}	read
Lval	Relative Lockset	Kind

Figure 1: Simplified example from Linux kernel. Below each function, in the curved box we show the relative lockset summary for the function, and below it, the guarded access summary for the function.

the summary with the symbolic values of the actual parameters at the callsite). Similarly, the set of locks that must be held and may have been released is updated, using the lockset summary of the callee.

We combine the above in the RELAY modular race analysis tool as follows. RELAY processes functions bottom-up in the call graph, starting with leaves, and working its way up the call graph. RELAY repeatedly picks a function to analyze amongst all the functions whose callees have been analyzed.

For each function being analyzed, RELAY performs three analyses: first a symbolic execution, second a relative lockset analysis, and third a guarded access analysis. The symbolic execution is used to express the values contained in memory locations in terms of the incoming values of the formals and the globals. This symbolic information is required by the two subsequent analyses. The relative lockset analysis is an iterative dataflow analysis that maintains a relative lockset at each program point. For call sites, the analysis uses the summaries of callees to compute the lockset after the call. Once a fixed point is reached, the relative lockset computed for the function's exit point becomes the relative lockset summary of the function. After performing the relative lockset analysis on a function, RELAY runs a guarded access analysis on that same function. The guarded access analysis maintains a monotonically increasing set A of guarded accesses. The analysis iterates through all the statements in the function (in a flow-insensitive way), accumulating in A the locations being accessed, along with the locks being held during those accesses. The information about which locks are held at the access points is provided by the results of the relative lockset analysis.

Consider the program comprising the two functions shown in Figure 1. RELAY begins with the leaf function `airo_read_stats`. Initially, the relative lockset at the entry point is the pair $(\{\}, \{\})$. The calls to `unlock` result in the addition of `ai->lock` to the negative lockset of program points `3:` and `5:`. Because negative locksets are *may* sets, the negative lockset of the exit point is the *union* of those of its predecessor program points `3:` and `5:`, namely `ai->lock`. Because positive locksets are *must* sets, the positive lockset is the *intersection* of those of the predecessors, which is the empty set. Thus, the relative lockset summary of the function is the pair: $(\{\}, \{$`ai->lock`$\})$. This summary states that the `airo_read_stats` function does not acquire any locks, and it may release (in fact in this case, it definitely does release) the `ai->lock` lock.

After having computed the relative lockset information for `airo_read_stats`, RELAY iterates through the statements of `airo_read_stats` to find the guarded access set of the function. There are three accesses in this function: the read of `ai->pwr.ev` on line 1, with a relative lockset of $(\{\}, \{\})$; the read of the `vals` array on line 5, with a relative lockset of $(\{\}, \{$`ai->lock`$\})$; and the write to `ai->stats.rx_p` with the same lockset. This information is collected in the guarded access summary of the function, which is shown in the left table of Figure 1 (the index 0 in `vals`[0] represents all array indices).

Next, RELAY picks the function `airo_thread`. The relative lockset for the entry location is the same as for `airo_read_stats`, namely $(\{\}, \{\})$. The symbolic execution tracks that at `7:` the lvalue `dev` refers to the formal `d`, and that at `8:` the lvalue `ai->lock` refers to `d->priv->lock`. As a result, the relative lockset at `8:` (just before the call to `airo_read_stats`) is $(\{$`d->priv->lock`$\}, \{\})$, indicating that `d->priv->lock` was added since the entry point of the function. Now the call to `airo_read_stats` is analyzed. The relative lockset summary for `airo_read_stats` is $(\{\}, \{$`ai->lock`$\})$, and since the symbolic execution tells us that `ai` is in fact `d->priv`, the instantiated summary in the caller's context is $(\{\}, \{$`d->priv->lock`$\})$. Updating the information from before the call $(\{$`d->priv->lock`$\}, \{\})$, with the effect of the call $(\{\}, \{$`d->priv->lock`$\})$ gives us the information $(\{\}, \{$`d->priv->lock`$\})$ after the call. In particular, after the call, we have lost the information we had previously about `d->priv->lock` being held, because `airo_read_stats` releases that lock. Furthermore, we gain the information that since the beginning of execution of `airo_thread`, the lock `d->priv->lock` may end up being released because of the call to `airo_read_stats`. Since the call is the last statement in `airo_thread`, the information after the call becomes the relative lockset summary for `airo_thread`.

Next RELAY processes all the statements in `airo_thread` to compute its guarded access set. There is only one access in `airo_thread` itself, namely the read of `dev->priv` at line 8 (the write to `ai` is not recorded because `ai` is a local stack variable). Using the symbolic information, this read access is recorded in the guarded access summary (shown in the right table of Figure 1) as an access to `d->priv` with relative lockset $(\{\},\{\})$. The other accesses in the guarded access summary of `airo_thread` are added when the call to `airo_read_stats` is processed. Since the relative lockset of the `ai->pwr.ev` entry in the summary is $(\{\},\{\})$ (*i.e.,* no locks were added or removed), the relative lockset for this access in the caller is just the relative lockset at the callsite. For the other two access in the summary of `airo_read_stats`, the negative lockset contains `ai->lock`, which, after plugging in the actuals corresponds to `d->priv->lock` as shown in the last two rows of the guarded access set shown in the figure.

Race Warnings. Once RELAY has computed the guarded access summaries for all functions that are thread entry points, it reports warnings for all pairs of accesses, where the lvalues may be aliases, and whose positive locksets have an empty intersection, and where at least one of the accesses is a write. Suppose that a sound alias analysis shows that the lvalues corresponding to the accesses shown on the right table of Figure 1 have no other aliases. In this case, RELAY reports that:

1. There are no races due to concurrent accesses to `vals` or `d->priv` as both accesses would be reads.
2. There are no races due to concurrent accesses to `d->priv->pwr.ev` as (the intersection of) the positive lockset(s) is non-empty.
3. There may be a race involving concurrent accesses to `d->priv->stats.rx_p` in different threads, as (the intersection of) the positive lockset(s) is empty. The accesses involved in this race are the access on line 5, but from two different threads.

3. ALGORITHM

This section describes our race detection algorithm in detail. As outlined in Section 2, our algorithm performs a bottom-up analysis that has three interacting components: a symbolic execution (Section 3.1), an analysis that computes lockset changes (Section 3.2), and an analysis that computes guarded accesses (Section 3.3). After the bottom-up analysis has finished running, the results are used to generate warnings (Section 3.4).

3.1 Symbolic Execution

Before starting the symbolic execution, we perform Steensgard's flow-insensitive points-to analysis [26], computing conservative representative nodes for all lvalues. These representative nodes are used in our symbolic execution to ensure termination. Our symbolic execution analysis keeps track of the values contained in memory locations in terms of the incoming values of the formals and the globals. Our analysis is fairly standard, and the details of the symbolic execution are orthogonal to the contribution of our work, so we only present an overview of our analysis here. The domains of the symbolic execution are shown Figure 2. We use metavariable $x \in \mathbb{X}$ to denote formals and globals, and metavariable $p \in \mathbb{P}$ to denote representative nodes from the Steensgard flow-insensitive points-to-analysis. The set $\mathbb{O}$ of symbolic lvalues denotes the locations that our symbolic execution analysis keeps track of, and these include formals, globals and field/pointer accesses through these. We use $os \in 2^{\mathbb{O}}$ to represent a set of lvalues. The set $\mathbb{V}$ of symbolic values denotes the values that our symbolic analysis computes, and these include: $\perp$, which means "not assigned yet"; $\top$, which means "any possible value"; i, which represents a constant integer; $init(o)$, which denotes the incoming value of lvalue o; $must(o)$, which represents a value that must point to lvalue o; and $may(os)$, which represents a value that may point to any of the lvalues in os. Finally, a symbolic execution map $\sigma \in \Sigma$ is a function from symbolic lvalues to symbolic values.

formals, globals	$x \in$	$\mathbb{X}$
PTA reps	$p \in$	$\mathbb{P}$
symbolic lvalues	$o \in \mathbb{O} ::=$	$x \mid x.f \mid p.f \mid (*o).f$
symbolic values	$v \in \mathbb{V} ::=$	$\top \mid \perp \mid i \mid init(o) \mid must(o) \mid may(os)$
symbolic map	$\sigma \in$	$\Sigma = \mathbb{O} \to \mathbb{V}$

Figure 2: Symbolic analysis domain

The symbolic execution keeps track of a symbolic map at each program point, and this symbolic map is updated using flow functions. The flow function for a simple assignment $x := e$ evaluates e in the current map to a symbolic value, and then updates x in the map. For assignments through pointers, namely $*x := e$, the flow function evaluates x to a symbolic value v_1 and e to a symbolic value v_2. Which lvalues are updated in the store depends on the value v_1. For example, if v_1 is $must(o)$, then only o is updated to the value v_2. As another example, if v_1 is $may(os)$, then all the lvalues in os are updated to the value v_2. The remaining cases, not shown here, are very similar in nature.

When performing symbolic execution on a given function, we must account for the effect that other threads may have on the state of variables. To do this, we use our Steensgard's points-to analysis to compute the set of locations that may escape the current thread, which means that they could be accessed by another thread. We then map these locations to $\top$ (meaning "any possible value") after each invocation of the symbolic flow function. This approach to handling thread interaction is very conservative. However, it retains full precision on non-escaping local variables, which are the most important locations to keep track of when re-expressing accesses in a function in terms of its formal parameters. For example, on line 8 of Figure 1, our symbolic execution is able to conclude that the lock being acquired is `d->priv->lock` because the variables involved, namely `dev` on line 6 and `ai` on line 7, are non-escaping local variables.

3.2 Lockset Analysis

After the symbolic execution has finished, RELAY runs a relative lockset analysis. A relative lockset L is a pair (L_+, L_-), where the set $L_+ \subseteq \mathbb{O}$ represents the locks that have definitely been acquired since the beginning of the function, and the set $L_- \subseteq \mathbb{O}$ represents the locks that may have been released since the beginning of the function. We denote by $\mathbb{L} = 2^{\mathbb{O}} \times 2^{\mathbb{O}}$ the set of all relative locksets.

The lockset analysis is a dataflow analysis whose domain is the lattice $(\mathbb{L}, \perp, \top, \sqsubseteq, \sqcup, \sqcap)$, where the ordering is defined as:

- $\perp = (\mathbb{O}, \emptyset)$, $\top = (\emptyset, \mathbb{O})$
- $(L_+, L_-) \sqsubseteq (L'_+, L'_-)$ iff $L'_+ \subseteq L_+ \wedge L_- \subseteq L'_-$
- $(L_+, L_-) \sqcup (L'_+, L'_-) = (L_+ \cap L'_+, L_- \cup L'_-)$.
- $(L_+, L_-) \sqcap (L'_+, L'_-) = (L_+ \cup L'_+, L_- \cap L'_-)$

The analysis runs bottom-up on the call graph. After a function f has been analyzed, its effect on locksets is stored as a summary $LockSummary(f) \in \mathbb{L}$ that represents the relative lockset at the end of the function. For simplicity of exposition, we assume that functions take only one parameter.

The flow function for the lockset analysis is shown in Figure 4. Because we model lock and unlock operations as function calls, the

$$lockUpdate : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$$

$$\begin{aligned} &lockUpdate\,((L_+, L_-), (L'_+, L'_-)) = \\ &\quad ((L_+ \cup L'_+) - L'_-, (L_- \cup L'_-) - L'_+) \end{aligned}$$

Figure 3: Relative lockset update

$$F : Stmt \times \mathbb{L} \to \mathbb{L}$$

$$F(call(e,a), L) = \bigsqcup_{f \in targets(e)} \begin{array}{l} \textbf{let } L_f = LockSummary(f) \textbf{ in} \\ \quad lockUpdate(L, rebind(L_f, f, a)) \end{array}$$

$$F(s, L) = L$$

Figure 4: Lockset flow function

only statements that modify locksets are function calls $e(a)$. In particular, the `lock(l)` function is modeled as having a relative lockset summary of $(\{\texttt{l}\}, \{\})$ and the `unlock(l)` function is modeled as having a relative lockset summary $(\{\}, \{\texttt{l}\})$. Given a function call $e(a)$, for each possible function f that e may represent, the flow function first retrieves the summary $LockSummary(f)$, and then, using the $rebind$ function shown in Figure 5, it replaces all occurrences of f's formal in the summary with the actual being passed in. The resulting rebound summary represents the changes in the lockset that occur from the moment f starts executing until it reaches a return. To find the relative lockset after the call to f (relative to the caller's entry point), we apply the changes indicated by the summary to the incoming relative lockset. This is done using the $lockUpdate$ function shown in Figure 3. In particular, the positive differences are added together and so are the negative differences, with the following post-processing: the locks that may have been released in f are removed from the final must-have-acquired lockset, and the locks that must have been acquired in f are removed from the final may-have-been-released lockset.

3.3 Guarded Access Analysis

Once the lockset analysis from Section 3.2 has finished computing the relative locksets for all program points of a given function, the guarded access analysis uses this information to compute the guarded accesses performed by the function.

A guarded access is a triple $a = (o, L, k)$, where $o \in \mathbb{O}$ is the lvalue being accessed, $L \in \mathbb{L}$ is the relative lockset at the point where the access is made, and $k \in \mathbb{K} = \{Read, Write\}$ is the kind of access being made (either a read or a write). The set of all guarded accesses is denoted by $\mathbb{A} = \mathbb{O} \times \mathbb{L} \times \mathbb{K}$.

For each function, our guarded access analysis maintains a guarded access set $A \subseteq \mathbb{A}$ for the entire function. After the lockset analysis has reached a fixed point for a given function, the guarded access analysis starts out by initializing the function's guarded set to the empty set. Then, for each statement s in the function, the access set is updated by calling $UpdateAccessSet(s, L)$, where L is the relative lockset computed by the lockset analysis at the program point right before s. As statements are being processed, the guarded access set increases monotonically, and when all statements in the function have been processed, the final guarded access set becomes the access summary of the function. For a function f, we denote the access summary of f by $AccessSummary(f)$.

The most important cases of the $UpdateAccessSet$ function are shown in Figure 6. For a function call $e(a)$, $UpdateAccessSet$ copies all guarded accesses from the callee, re-expressing them in the caller's context. In particular, for each possible function f that e may represent, we look up the access summary of f, and for each guarded access (o, L_f, k) in the summary, we use $rebind$ to re-express o and L in terms of the caller's actuals. We also use $lockUpdate$ to plug the rebound L into the caller's context.

$$rebind : T \times Function \times Expr \to T$$

$$rebind(q, f, e) = q[formal(f) \mapsto eval(e)]$$

Figure 5: Rebinding formals to actuals. The function $eval(e)$ evaluates e to a symbolic value using the store computed by the symbolic execution at the program right before e is used.

$UpdateAccessSet : Stmt \times \mathbb{L} \to$ `void`

$UpdateAccessSet(x := e, L) =$
    $A := A \cup \{(eval(e), L, Read)\}$
    $A := A \cup \{(x, L, Write)\}$

$UpdateAccessSet(call(e,a), L) =$
    $A := A \cup \{(e, L, Read), (a, L, Read)\}$
    foreach f **in** $targets(e)$ **do**
        foreach (o, L_f, k) **in** $AccessSummary(f)$ **do**
            let $L' = lockUpdate(L, rebind(L_f, f, a))$ **in**
            let $o' = rebind(o, f, a)$ **in**
            if $isAccessible(o')$ **then**
                $A := A \cup \{(o', L', k)\}$

Figure 6: Guarded access update. We only show the case for assignment to a global, and a function call.

The resulting lvalue o' and lockset L' are added to the guarded access set A only if o' is accessible from globals or from the formals of the function being analyzed. The $isAccessible(o')$ call performs this pruning by running a reachability query in the flow-insensitive points-to graph from the globals and formals to the node representing o'.

3.4 Warning Generation

Once the bottom-up guarded access analysis from Section 3.3 has finished running on all functions, the $GenerateWarning$ function from Figure 7 uses the resulting guarded access summaries to generate warnings. The $GenerateWarning$ function takes as a parameter the thread entry points, which are all the functions passed to thread creation sites, in addition to the original thread that starts executing when the kernel boots up.

For each pair of thread entry points, $GenerateWarning$ re-

$GenerateWarnings : 2^{Func} \to$ `void`

$GenerateWarnings(ThreadEntryPoints)$
    foreach (f, f') **in** $ThreadEntryPoints^2$ **do**
        foreach (o, L, k) **in** $AccessSummary(f)$ **do**
            foreach (o', L', k') **in** $AccessSummary(f')$ **do**
                let $(L_+, L_-) = L$ **in**
                let $(L'_+, L'_-) = L'$ **in**
                if $mayEqual(o, o') \wedge (L_+ \cap L'_+ = \emptyset) \wedge$
                    $(k = Write \vee k' = Write)$ **then**
                      $GenerateWarning(o, o')$

Figure 7: Producing warnings

trieves the guarded access sets for the two entry points, and then it searches for two guarded accesses such that the lvalues may be equal, the must-hold locksets do not overlap, and one of the accesses is a write. If two such accesses are found, a warning is generated.

The $mayEqual$ function determines if two lvalues could be the same (that is to say, could alias). Nominally, $mayEqual$ looks up the representative node of the two lvalues in the flow-insensitive points-to graph, and returns true if the two representative nodes are the same. To improve precision, some sound syntactic checks are added to avoid going to the points-to graph when it is not needed. For example, if the two lvalues are the exact same variable, then $mayEqual$ immediately returns true without consulting the points-to graph.

4. EXPERIMENTS

We now describe our experiences running RELAY on a large software base, the Linux Kernel v. 2.6.15, which is about 4.5 million lines of code, spanning 46872 functions, scattered across 18042 files. The kernel was first pre-processed using the `makeallyes` option and with loadable module support turned off so as to maximize the code included in the build. This choice serves to demonstrate the scalability of our techniques as well as to obtain a close understanding of the variety of idioms used in systems code for synchronizing and avoiding data races.

We begin in Section 4.1 with some details about the implementation of RELAY. We then describe the results of running RELAY on the Linux kernel. In particular, RELAY's sound, context- and flow-sensitivity resulted in the generation of 5022 warnings (over a 4.5 million line code base). We performed a close analysis of a randomly chosen subset of the warnings, and found that most of these warnings were in fact false positives. We categorized the false positives based on the coding idioms used to prevent races, and present the result in Section 4.2. Our categorization reveals that to soundly remove the false positives would require sophisticated analyses that are concurrency- , path- and shape- sensitive, and also scale to millions of lines, a challenging task that we leave to future work.

Instead, we used our categorization of the sample warnings to devise post-processing *warning filters* capable of automatically placing every warning into one of the categories (Section 4.3). After applying the filters we were left with 161 warnings, 31 of which we again carefully categorized. 25 of this subset (80%) were real data races.

4.1 Implementation

RELAY is implemented in OCAML and uses CIL[22] as a front-end. To build the call graph, RELAY processes the kernel one file at a time. It traverses each function's body, adding call edges to each function called within the body. The bottom-up analysis can process a function as soon as summaries of the callees have been computed. Thus, it is possible to analyze multiple functions concurrently, as long as the summaries for their callees has been computed. RELAY exploits this by distributing the summary computations across a grid of 32 nodes each equipped with 2.8Ghz Xeons and 4Gb of RAM. Each SCC of the call graph is analyzed by a fresh process. This process, which starts at any free node in the grid, downloads the summaries of the callees to the local file system, computes the new summaries for the SCC functions and then informs a server of the whereabouts of the new summaries. RELAY took 72 hours to perform the whole analysis on a single machine. By distributing the computation, we were able reduce the analysis time to 5 hours.

4.2 Warnings

RELAY uses the guarded access sets to generate 5022 warnings using the method described in Section 3.4. Rather than undertake the herculean task of sifting through all these warnings, we chose to randomly sample and classify 90 of the warnings. This sample contained some races, but the vast majority of the warnings were false positives. However, it turns out that most of the false positives in the sample fell into one of a handful of categories described below. Each of these patterns appears to require a somewhat specialized analysis as they require careful reasoning about path-sensitivity, concurrency and the shape of the heap neither of which is easy to scale.

1. Initialization: A common idiom is to allocate a structure within a thread, and perform some initialization *without* any synchronization while the structure is still local to the thread, and then to make the structure accessible to other threads, by adding it to a global data structure. Even though subsequent accesses happen while holding a lock, RELAY will report a warning due to the first unprotected access. Figure 8 shows a simple code fragment from the kernel that illustrates this pattern. The lower function calls a helper to allocate a structure. The structure `conn` is allocated on line `1:` and passed back to the caller. At this point, the structure is not shared and so on line `4:` some fields of the structure get initialized, and then on line `5:` the structure gets added to a global queue after which it can be accessed by multiple threads. RELAY would warn about subsequent accesses being a race with the access on line `4:`.

2. Unlikely aliasing: Many of the warnings reported are false positives because of the flow-, field- and arithmetic- insensitivity of the alias analysis. For example, our alias analysis reports that there is a single "blob" representative node that represents over 10000 objects, and race conditions reported on objects within this blob are most likely false positives.

3. Unsharing: RELAY reported many warnings on objects that are indeed shared, but which are not shared *during* the time they were accessed. A common situation where this happens is that the object belongs in a shared list, and therefore can be accessed by multiple threads. However, just *before* a thread performs the access, it *removes* the object from the shared list, and then safely accesses the object without any lock. Figure 9 illustrates this pattern. `pam` is a reference to the first element of the `page_addr_pool`, and this element is removed from the list in line `1:` (after acquiring the appropriate locks for the list). Then, the list lock is released and on line `2:` the previously shared object referred to by `pam` is written to without any synchronization.

4. Re-entrant locks: A significant fraction of the false warnings we analyzed were because some data structures were protected with the kernel semaphore, which is a re-entrant lock. For such locks, acquires and releases can be nested, and after k nested acquires, the lock is actually released only after k successive releases. RELAY conservatively models these locks, by treating them as released after the very first release call, and thus, finds several unsynchronized shared accesses, even though they are protected by previous acquires.

5. Non-parallel threads: Many false warnings arose due to unsynchronized accesses that take place at instances when the kernel has ensured, using one of several mechanisms, that there is only a *single* active thread that can access the shared object. The most common case is when an object is accessed from multiple threads, but the threads use program logic, including signals and other mechanisms, to order operations in such a way that the threads in essence never run in parallel. One such example is shown in Figure 10. On line `1:` the function `start_sync_thread` checks the shared variable `state` to see if the thread already exists. If not, on line `2:` it attempts to create the thread by looping until the thread gets created. After the creation succeeds, the parent thread waits for the child to set the `state` variable on line `4:` and then signal completion `6:`, at which point, on line `3:` the parent returns. This code essentially ensures that only one copy of the `sync_thread` ever runs, and so the access on line `4:` is safe, even through RELAY will warn that two copies of `sync_thread` may write to `state` at the same time. There are other mechanisms that, like the above, require a very precise thread interleaving analysis, such as the use of blocking primitives like `wait_for_completion` (illustrated in the example).

6. Conditional Locking: Several false warnings generated by RELAY were because the program checks some condition to determine whether to acquire locks, and later, checks a correlated condition to determine whether the access should occur. Unfortunately, the acquisition of the lock and the actual access occur in different blocks or functions thereby introducing a path-sensitivity problem. The example in Figure 11 exhibits this pattern. The upper function either returns `NULL` without holding the lock if the condition on line `1:` holds, or acquires the lock on line `2:` and returns a non-null value. This return value is checked on line `4:` before performing the access on line `5:`.

4.3 Filters

We have devised simple syntactic filters based on the above categorization to automatically categorize the warnings thereby yielding a subset of the warnings that are very likely genuine races. The design of these filters was guided by finding common patterns among the warnings in a given category.

These filters are very aggressive, and they are unsound, in the sense that they can remove real races too. However, since this source of unsoundness is confined to a post-processing pass, it can easily be removed as our analysis becomes more precise.

We now describe the filters — in each case, in parentheses we list the categories the filter corresponds to.

1. Thread-local Allocation (Initialization): To handle the initialization false-positives, we filter out warnings on objects that are allocated inside the thread within which the conflicting access occurs.

2. Large Points-To Reps. (Unlikely aliasing, Unsharing): For unlikely aliasing we can filter warnings where the flow-insensitive alias analysis is asked to compare lvalues whose representative nodes represent more than k lvalues for a parameter k ($k = 1$ for our results). Typically, these are nodes where different data-structures are mixed at a common function (*e.g.* , two different lists merging at a node removal function). As this filter captures warnings involving data-structures, it also applies to the "unsharing" pattern.

3. Bootup thread (Re-entrant locks): The most heavily used re-entrant lock is `kernel_sem`. This lock is mainly used by the boot-up thread which holds it for most of its execution. Thus, to filter warnings about re-entrant locks, it sufficed to filter warnings where one of the accesses was in the boot-up thread.

4. Same entry (Non-parallel threads): We noticed that many false positives involving threads that cannot execute concurrently were warnings where the two accesses originated from the same thread. We therefore designed a filter that removes such warnings.

```
__rxrpc_create_connection(**_conn){
1: conn = kmalloc(sizeof(...), ...);
2: timer_init(&conn->timeout, ...);
3: *_conn = conn;
}

rxrpc_create_connection(*trans){
   __rxrpc_create_connection(&conn);
   /* fill in the specific bits */
4: conn->addr.sin_family = AF_INET;
   write_lock(&peer->conn_idlock);
5: list_add(&conn->id_link, _p);
   //...
}
```

Figure 8: Initialization

```
set_page_address(*virtual){
   spin_lock_irq(&pool_lock);
   pam = list_entry(page_addr_pool);
1: list_del(&pam->list);
   spin_unlock_irq(&pool_lock);
2: pam->virtual = virtual;
   spin_lock_irq(&pas->lock);
3: list_add(&pam->list, &pas->lh);
   spin_unlock_irq(&pas->lock);
}
```

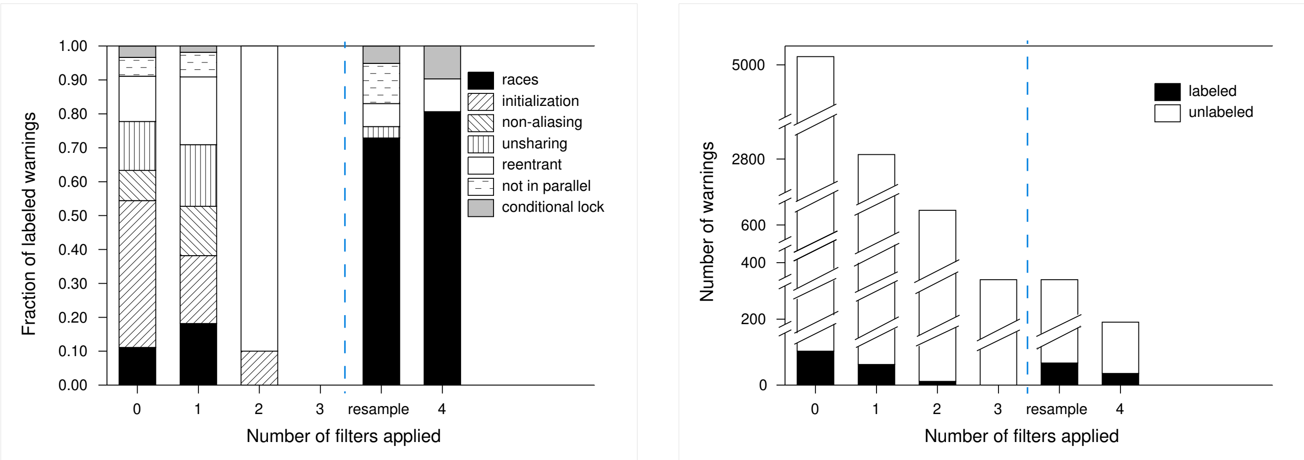
Figure 9: Unsharing

```
static sync_thread(*startup){
4: state = IP_VS_STATE_MASTER;
5: set_sync_mesg_maxlen(state);
6: complete(startup);
   //...
}

start_sync_thread(state, ...){
1: if (state == IP_VS_STATE_MASTER)
     return -EEXIST;
repeat:
2: if (kernel_thread(sync_thread,&startup) < 0)
      goto repeat;
3: wait_for_completion(&startup);
   return 0;
}
```

Figure 10: Non-parallel threads

```
static * swap_info_get(entry){
1: if (!entry.val)
      goto 3;
   p = &swap_info[type];
2: spin_lock(&swap_lock);
   return p;
3: return NULL;
}

swap_free(entry){
   p = swap_info_get(entry);
4: if (p) {
5:    swap_entry_free(p, ...);
      spin_unlock(&swap_lock);
   }
}
```

Figure 11: Conditional locking



(a) Relative proportions of categories in the manually labeled warnings after each filter is applied.

(b) Number of manually labeled warnings vs. unlabeled warnings after each filter is applied.

Figure 12: The effect of filters on warnings.

4.4 Results

We evaluate the result of applying the filters using two criteria. First, they should remove false positives, *i.e.,* after applying the filters, the *fraction* of real races left in the warning should increase (Figure 12(a)). Second, they should not remove too many races, *i.e.,* after applying the filters we should still be left with a pool of warnings large enough to contain many real races (Figure 12(b)).

We applied the filters to the warnings as follows. First, we drew a random sample of 90 warnings from the 5022 warnings generated by RELAY. We manually placed each of the warnings into one of the six categories described in Section 4.2. When a warning fell in multiple categories, as was often the case, we placed the warning into the first (according to the order shown) category.

Next, we applied the filters in the order described in Section 4.3. Figure 12(b) shows how after each filter is applied, the total number of warnings as well as the number of warnings in the manually categorized sample set decreases. Figure 12(a) shows how the distribution of the categories changes in the remaining set of samples, as we apply more filters. The important thing to note here is that the dark solid bar, which represents the percentage of real races in the sample set increases monotonically as we apply filters, and reaches 80 % after having applied 4 filters.

We now describe the four steps in applying each one of the four filters:

1. After applying the first filter (which is meant to remove initialization false positives), the total number of warnings drops to 2812 and the manually categorized sample set drops to 55 (bar 1). Moreover, the fraction of remaining sampled warnings that are initialization false positives drops from about 43% to 20%, indicating that the filter did in fact remove a larger proportion of initialization false positives than other warnings.

2. After applying the representative node filter (which was meant to remove false positives due to unlikely aliasing or unsharing), the total number of warnings drops to 639 and the sample set drops to 10 (bar 2). Of the remaining samples, there are no more unlikely aliasing or unsharing false positives, indicating that the filter was a good heuristic for removing these false positives. This filter also had the unintended effect of removing all the non-parallel-threads false positives. Unfortunately, it also removed all the races that we had identified in our first sample set.

3. After applying the third filter, the manually categorized sample set went down to zero (bar 3), and so we *re-sampled* the set of 355 remaining warnings to obtain a new sample set of 59 warnings which we again manually categorized. After resampling, we applied the third filter (bar labeled "resample"), namely the bootup-thread filter, which was meant to remove the re-entrant locks false positives. At this point, the percentage of false positives categorized as re-entrant locks decreases significantly, indicating that the filter is effective at removing these false positives.

4. After applying the same-entry filter (which is meant to remove non-parallel threads false positives), all the non-parallel threads false positives have been removed (bar 4). At this point, the number of remaining warnings is 161, and the size of the manually categorized sample set is 31, of which 25 (80 %) are real races. Note that we have not been able to devise a filter targeted at conditional locks, and therefore the majority of remaining false positives fall in this category.

We conclude from the above that the filters effectively refine the set of warnings and increase the fraction of races from 11% to 80 %, without eliminating an unacceptably large number of races. Counted another way, we manually analyzed 149 warnings in all, and found 53 races.

Races. After the application of the filters, the vast majority of warnings are real races. Figure 13 shows one such race that survives all filters. By the time we obtained our results, this race had already been reported and fixed.

The race involves the read on line `2:` of `p->size` and the write, on line `5:` of `t->size`, since `t` and `p` can point to the same object and there are no common locks held. This race is serious because the function `change_page_attr` uses the `p->size` parameter that is passed in as the bound for a loop iterating over an array. Due to the race the read of `p->size` can return a stale bound causing the loop inside `change_page_attr` to access the array out of bounds.

```
iounmap(volatile *addr){
   read_lock(&vmlist_lock);
   for (p = vmlist; p; p = p->next) {
      if (p->addr == addr) break;
   }
1: read_unlock(&vmlist_lock);
   change_page_attr(virt_to_p(p->phys_addr),
2:                    p->size >> PAGE_SHIFT);
}

/* called with write_lock(vmlist_lock) */
__remove_vm_area(*addr){
3: for (t = vmlist; t != NULL; t = t->next) {
        if (t->addr == addr) break;
   }
4: unmap_vm_area(t);
5: t->size -= PAGE_SIZE;
   return t;
}
```

Figure 13: A real race found after applying filters.

5. RELATED WORK

We now present a brief overview of the vast body of work pertaining to techniques for finding data races.

Dynamic Techniques. Most currently used race detection techniques are dynamic. These detectors principally use two techniques. The first is Lamport's happen's-before relation [17], used in [7, 18]. The second is dynamically computed locksets, popularized by [25]. Much recent effort has gone into lowering the overhead imposed by dynamic analysis – for example by using statically precomputed locksets to prune redundant checks [5, 30]. Recent developments include the extension of these techniques to find *atomicity* [11] violations in Java code [29, 12], and the use of automated replay to determine whether a given dynamically detected race is benign or harmful [21]. The principal drawback with dynamic approaches is that they only work on closed programs which can be executed, they require tests that sufficiently exercise the code, and that ultimately, they cannot be used to classify all potential accesses. It is also unclear whether they can be scaled to multi-million line, low-level software.

Static Techniques for Java. Java's native support for multithreading coupled with its restricted use of syntactically scoped locks has given rise to a variety of static techniques for detecting and proving the absence of races in Java code. Early work includes the development of type systems that encode a static lockset analysis [9, 10]. These type based approaches were made more expressive by incorporating a notion of *ownership* [3]. Similar type systems were designed to ensure race-freedom in Cyclone [14]. While these type systems are eminently scalable, they require user annotation, though there has been some work on using SAT solvers [13] and dynamic locksets [2] to infer the lock annotations. Another line of work is that of [28] which finds races by computing an *Object Use Graph* that statically approximates the dynamic happens-before relation. A recent line of work [20] shows how to effectively use cloning-based context-sensitivity to drastically improve the precision of lockset computations. The approach was further refined in [19] by using a notion of must-not aliasing to prune the set of warnings. The above techniques exploit key properties of Java – namely the scoped use of locks, which mitigates the need for flow-sensitivity. Thus, while they are not directly applicable to our setting, we believe that it may be possible to apply ideas like ownership and must-not aliasing to lower the false positives that arise due to initialization and unlikely aliasing respectively.

Static Techniques for C. Analyses devised for finding races in C programs must cope with several additional problems. Principal among them is the use of unstructured locks, which force the analysis to be flow- and context- sensitive. The only approach we know of that has scaled to millions of lines is RACERX, which also finds deadlocks, and runs over large code bases in minutes. Unlike RELAY, RACERX uses a top-down approach to computing the locksets at each program point. The paper reports that in order to scale, several drastic compromises had to be made, such as truncating the summaries and representing all lvalues with their types. As a result, the analysis discards valuable information prematurely, discarding possible races well before the warning generation phase. Consequently, the tool was only able to unearth a small handful of warnings and, and an order of magnitude fewer races. A more precise approach is that of [23] which uses a constraint based technique to compute *correlations* that describe the locks that protect an lvalue. While this approach is as precise as ours, it has only been applied to programs two orders of magnitude smaller than the Linux kernel. We conjecture that the principle bottleneck is the difficult task of solving a monolithic set of constraints generated over millions of lines of code. This is in contrast to RELAY whose algorithm is modular and readily parallelizable. Finally, heavyweight techniques such as model checking [16, 24] have been applied to find and prove the absence of races. These techniques are essential in situations where the synchronization is not lock-based, but instead is via exotic mechanisms like state variables, interrupt disabling, or the idioms described in Section 4. It is unclear whether such heavyweight methods can be scaled to large code bases.

Finally, while others have designed bottom-up analyses using complete summaries [4, 6], our work, and the notion of parallelization is directly inspired by the approach taken by [15].

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a static race detection analysis that scales to millions of lines of C code. At the heart of our technique is the notion of a relative lockset which allows functions to be summarized independent of the calling context. This, in turn, allows us to perform a modular, bottom-up analysis that is easy to parallelize. We have analyzed 4.5 million lines of C code in 5 hours, and after applying some simple filters, found a total of 53 races.

One of our long-term goals is to soundly eliminate false positives to the point where a large fraction of the remaining warnings, say more than 70%, correspond to real races. To this end, we would like to replace the simple but unsound filters with sound analyses targeted at the coding patterns that we have found to be the leading causes of false positives. Examples of such analyses include a thread-escape analysis for the initialization pattern, a less conservative version of the *lockUpdate* function for the re-entrant locking pattern, and a light-weight shape analysis for the unsharing pattern.

Another long-term goal is to address the problem of determining "serious" races. Some of the races are clearly benign, as deduced from syntactic cues such as variable names like `oops_in_progress`, while others appear to be dangerous. The dangerous races are often those that cause higher-level semantic bugs, such as atomicity violations or unsafe memory accesses like the one shown in Figure 13. We hope to use RELAY as a foundation for finding such deeper semantic bugs.

7. REFERENCES

[1] FWGrid Project. http://fwgrid.ucsd.edu.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S.D. Stoller. Optimized run-time race detection and atomicity checking

using partial discovered types. In *ASE 05: Automated Software Engineering*, pages 233–242, 2005.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[4] Ben-Chung Cheng and Wen mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI 2002: Programming Languages Design and Implementation*, pages 258–269. ACM, 2002.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[7] Annette Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[8] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP 03: ACM Symposium on Operating System Principles*, pages 237–252. ACM Press, 2003.

[9] C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming*, LNCS 1576, pages 91–108. Springer, 1999.

[10] C. Flanagan and S.N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Languages Design and Implementation*, pages 219–232. ACM, 2000.

[11] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 03: Model Checking Software*, LNCS 2648, pages 213–224. Springer, 2003.

[12] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[13] Cormac Flanagan and Stephen N. Freund. Type inference against races. In *SAS*, pages 116–132, 2004.

[14] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI*, pages 13–25, 2003.

[15] Brian Hackett and Alex Aiken. How is aliasing used in systems software ? In *FSE 2006: Foundations of Software Engineering*, pages 69–80. ACM, 2006.

[16] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI 2004: Programming Languages Design and Implementation*, pages 1–12. ACM, 2004.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] John M. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 129–139, 1993.

[19] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[20] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[21] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, page (to appear), 2007.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

[23] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[24] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

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

[26] B. Steensgard. Points-to analysis in almost linear time. In *POPL 96: Principles of Programming Languages*, pages 32–41. ACM, 1996.

[27] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Winter 1993 Technical Conference*, pages 97–106, 1993.

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

[29] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[30] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.