When Threads Meet Events: Efficient and Precise Static Race Detection with Origins

Bozhen Liu*
Texas A&M University, USA
april1989@tamu.edu

Peiming Liu*
Texas A&M University, USA
peiming@tamu.edu

Chia-Che Tsai
Texas A&M University, USA
chiache@tamu.edu

Dilma Da Silva
Texas A&M University, USA
dilma@cse.tamu.edu

Yanze Li
Texas A&M University, USA
li14865@tamu.edu

Jeff Huang
Texas A&M University, USA
jeff@cse.tamu.edu

Abstract

Data races are among the worst bugs in software in that they exhibit non-deterministic symptoms and are notoriously difficult to detect. The problem is exacerbated by interactions between threads and events in real-world applications. We present a novel static analysis technique, O2, to detect data races in large complex multithreaded and event-driven software. O2 is powered by “origins”, an abstraction that unifies threads and events by treating them as entry points of code paths attributed with data pointers. Origins in most cases are inferred automatically, but can also be specified by developers. More importantly, origins provide an efficient way to precisely reason about shared memory and pointer aliases.

Together with several important design choices for race detection, we have implemented O2 for both C/C++ and Java/Android applications and applied it to a wide range of open-source software. O2 has found new races in every single real-world code base we evaluated with, including Linux kernel, Redis, OVS, Memcached, Hadoop, Tomcat, ZooKeeper and Firefox Android. Moreover, O2 scales to millions of lines of code in a few minutes, on average 70x faster (up to 568x) compared to an existing static analysis tool from our prior work, and reduces false positives by 77%. We also compared O2 with the state-of-the-art static race detection tool, RacerD, showing highly promising results. At the time of writing, O2 has revealed more than 40 unique previously unknown races that have been confirmed or fixed by developers.

Keywords: Origins, Data Race Detection, Pointer Analysis, Static Analysis

ACM Reference Format:

1 Introduction

Threads and events are two predominant programming abstractions for modern software such as operating systems, databases, mobile apps, and so on. While the thread vs. event debate has never ended [44, 66], it is clear that both face a common problem: threads and events often lead to non-deterministic behaviors due to various types of race conditions, which are notoriously difficult to find, reproduce, and debug.

There has been intensive research on race detection of multithreaded code. Most successful techniques have been dominated by dynamic analysis [13, 23, 31, 42, 46, 72], notably Google’s ThreadSanitizer [56]. However, dynamic techniques face an inherent challenge of performance overhead and low code coverage. In contrast, static detection techniques have had only very limited success, notably Facebook’s RacerD [4, 8], despite decades of research [12, 37, 43, 47, 68]. A crucial reason is that reasoning about races typically requires sophisticated pointer alias analysis to attain accuracy, which is difficult to scale.

Races in event-driven programs have attracted much attention in recent years [14, 20–22, 26, 40, 49, 53]. Event-based races can be more challenging to detect than thread-based races because most events are asynchronous and the event handlers may be triggered in many different ways. Moreover, the difficulty in detecting event-based races is exacerbated by interactions between threads and events, which are common in real-world software such as distributed systems. The state-of-the-art race detectors [4, 8, 56] do not perform well

*Bozhen Liu and Peiming Liu contributed equally to this work.
in detecting event-based races, also due to the large space of
casual orders among event handlers and threads.

In this paper, we present O2, a new system for detecting
data races in complex multithreaded and event-driven
applications. We show that conventional thread-sensitive static
analysis (with some tuning and care) is highly effective for
finding races, even more effective than RacerD. A key con-
cept behind O2 are origins, an extended notion of threads and
events that unify them through two parts: 1) an entry point
that represents the beginning of a thread or an event handler,
and 2) a set of attributes that capture additional semantics,
such as thread ID, event type, or pointers to memory objects
that will be used in the thread or event handler. Figure 1
depicts an "origin" view for threads and events in C/C++ and
Java. The origin attributes can be specified or inferred auto-
matically at the origin’s entry point and the allocation site
of the receiver object. We elaborate the design in Section 3.1.

Rather than a straightforward unification, origins enables
origin-sensitive pointer analysis (OPA), in which the conven-
tional call-string-based or object-based context abstractions
are replaced by origins. This has several advantages:

- Functions within the same origin share the same context,
  therefore the computation complexity inside an origin
does not grow with the length of the call chain; and
- Computing \( k \)-most-recent calling contexts at every call
  site is redundant in many applications [64], e.g., when
determining which objects are local to or are shared by
which threads.
- The crucial origin entry point is preserved, not discarded
  as a trivial context in \( k \)-limiting [57] when the call stack’s
  depth exceeds the context depth \( k \).

Meanwhile, compared to conventional thread-based [9,
48, 50, 63] or event-based [6, 25, 27, 39, 70, 71] analyses,
the inclusion of data pointers in origins enables precisely identi-
fying shared- and local-memory accesses by different threads
and events. We develop origin-sharing analysis (OSA), which
uses an origin-sensitive heap abstraction to precisely com-
pute heap objects local to each origin, and objects shared
by each combination of multiple origins. OSA has several
advantages over classical thread-escape analysis. In partic-
ular, besides answering whether an object is shared, OSA
provides detailed information on how the object is shared
across origins, which is needed by race detection.

To illustrate these advantages, consider an example in
Figure 2. To correctly infer that threads T1 (line 5) and T2 (line
6) do not access the same data on line 23, typically, a \( k \)-call-
site analysis (denoted \( k \)-CFA) is performed, in which \( k \) is the
depth of the call chain [59]. Additionally, a call-site-sensitive
heap context is necessary to analyze the object allocation on
line 13. This complexity is shared by \( k \)-object-sensitivity [41]
(denoted \( k \)-oba), in which the sequence of subN() functions
are invoked on different receiver objects. With origins, it
suffices to mark the function run() in each thread as an
origin’s entry point. In this way, the allocation on line 13
can be distinguished by an origin unique to its thread. At the
same time, the virtual function act() invoked by each thread
on line 24 can be distinguished by the origin’s data pointers:
\( s \) and \( op1 \) for T1, and \( s \) and \( op2 \) for T2. Thus, it can be inferred
that the two threads invoke different member functions (from
\( util() \) to \( act() \)) in classes \( Op1 \) and \( Op2 \) respectively, which
manage the object that \( y \) points to differently. Figure 2(d)
shows a sample OSA output for the example code.

In addition to OPA and OSA, there are a few important
design choices we made in O2 that together make static race
detection highly effective. First, O2’s race detection engine
is highly optimized to achieve scalability and precision. We
construct a static happens-before graph (SHB) and use static
"happens-before" instead of static "may-happen-in-parallel"
as the foundational concept of the analysis. This allows
pruning many infeasible race pairs by checking only graph reach-
bility. Second, we develop several sound optimizations that
scale race detection to large code bases, including:

- An efficient representation of origin-local happens-before
  relations, which further enables efficient checking and
  caching the happens-before relation between memory ac-
  cesses;
- A compact representation of locksets, which enables a fast
  check of common locks and an efficient cache policy of
  the intermediate results;
- A lock-region-based race detection that allows effectively
  merging many memory accesses into a representative one,
  which reduces the number of race checks significantly.

We implemented O2 for both C/C++ and JVM applications
based on LLVM [36] and WALA [69], and applied it to a large
collection of widely-used mature open-source software. The
results show that O2 is both efficient and precise: it scales to
large programs, being able to analyze millions of lines of code
in a few minutes, up to 568x faster and reduces false positives
by 77% on average compared to existing static analyses from
our prior work (D4 [35]).

We compared O2 with RacerD (v1.0.0), the most recent
state-of-the-art static data race detector. For the programs
that can be compiled and analyzed by RacerD, O2 achieves
comparable performance while detecting many new races
and 4.33x fewer warnings on average. In most of the evalu-
ated programs in which O2 detects new races, RacerD either
fails to find the races or cannot run due to compiler errors.
Figure 2. (a) The example code. (b) The origin-sensitive call graph, where each origin consists of a sequence of calls of arbitrary length. The origin attributes precisely determine the call chain executed in each origin. (c) The context-sensitive call graph without origin. (d) A sample origin-sharing analysis (OSA) output.

Surprisingly, O2 found real and previously unknown races in every single real-world code base we evaluated with. At the time of writing, O2 has revealed more than 40 unique race bugs that have been confirmed or fixed by developers, including Linux kernel, Redis/RedisGraph, Open vSwitch OVS, Memcached, Hadoop, ZooKeeper, and the Firefox Android apps. O2 has been integrated into a commercial static analyzer. A free version of the tool is available at coderrect.com.

2 Related Work

Thread vs. Event. In their seminal work [28] in the late seventies, Lauer and Needham compared event-driven systems with thread-based systems and regarded threads and events as intrinsically dual to each other. In the mid-1990s, however, Ousterhout [44] argued against using threads due to the difficulty of developing correct threaded code. Lee [29] also noted the lack of understandability and predictability of multi-threaded code due to nondeterminism and preemptive scheduling. On the other hand, Von Behren et al. [66] remarked on the “stack ripping” problem of events and advocated for using threads for their simple and powerful abstraction. In Capriccio [67], they used static analysis and compiler techniques to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior. Adya et al. [5] also backed Von Behren by noting the question of threads or events as orthogonal to the question of cooperative or preemptive scheduling.

A key insight behind origins is that there exists an inherent connection between threads and events: they are both ways to implement a unit of the application’s functionality, corresponding to a unique origin. By reasoning at a higher level of abstraction with origins, we can systematically reason about both threads and events, and leverage such application-level semantics to develop a scalable and precise static analysis.

Unifying Thread and Event. It is highly desirable to unify threads and events so that these two models can be combined to achieve optimal performance on a real environment. In fact, for application domains with both heavy concurrency and intensive I/O such as web and database servers, a hybrid model of threads and events is often used. For example, in web servers and mobile applications, I/O and lifecycle events are used to model network connections and user interactions, and thread pools are used to handle concurrent user requests. Researchers have exploited this direction at the programming language level, including Scala with Actors [18] and Haskell [32], to produce a unified concurrency model. To fill the missing gap in program analysis, we propose a novel concept of origins and a static race detection approach that works uniformly across different components, regardless of their deployment of threads and events.

Origins and Pointer Analysis Contexts. Origins unify threads and events as the context to achieve significant improvements in the pointer analysis scalability and precision. Using threads as the context in pointer analysis is not new [9, 48, 50, 63], however, combining them through the data pointers is new and powerful. Meanwhile, the union of threads and events is essential to detecting races triggered by interactions between threads and events. In fact, all the new
races we report in Section 5.4 are caused by a combination of threads and events, which owe to origins. If considering events only or threads only, or considering them separately, these races will be missed. In real-world code bases, races often happen in a complex environment that leverages both concepts for concurrency. It is not obvious how to capture them without unifying threads with events. Section 4.2 provides details of our approach for handling events in Android apps.

Many dataflow analysis techniques [6, 25, 27, 39, 70, 71] have been proposed for event-driven programs to model event lifecycles and event handlers, but they only scale to hundreds of lines of code. These techniques either compromise precision due to unsound treatment of thread interactions or lose scalability due to expensive value-flow analysis. Other algorithms for multithreaded programs are not general as they target specific analyses (e.g., escape analysis and region-based allocation [55], synchronization elimination [19, 51]), or only work for structured multithreading (e.g., Cilk [15, 52]). In addition to using threads or events, prior research has proposed a variety of ways to represent contexts such as call-site [57, 58], receiver object [41] and type [60]. Recently, selective context-sensitive techniques [16, 17, 24, 33, 34, 38, 61, 64] have also been proposed. Although much progress has been made, context-sensitive pointer analysis remains difficult to scale.

**Static Race Detection.** RacerD, developed at Facebook, is by far the most successful static race detector [4]. It is regularly applied to Android apps in Facebook and has flagged over 2500 issues that have been fixed by developers before reaching production [8]. RacerD’s design favors reducing false positives over false negatives through a clever syntactic reasoning, but it does not reason about pointers and thus can miss races due to pointer aliases. In contrast, O2 deals with both Java pointers and low-level pointers in C/C++ such as indirect function targets and virtual tables. Other classic static race detection tools (e.g., RacerX [12], RELAY [68]) have various difficulties when applied to modern software. RacerX contains many heuristics and engineering decisions, which are difficult to duplicate. RELAY depends on the CIL compiler front-end, which supports only a subset of C and has not been actively developed [11]. Technically, RELAY uses a context- and field-insensitive pointer analysis, a major source of false positives. String-pattern-based heuristics are used in RELAY to filter out false aliasing. These heuristics are effective in reducing false positives, but are only specific to the code conventions in the target program and are unsound.

OSA can benefit any analysis that requires analyzing pointers or ownership of memory accesses, e.g., deadlock, over-synchronization, and memory isolation. We present O2’s race detection engine in the next section.

### 3.1 Automatically Identifying Origins

In general, a program can be divided into many different origins, each represents a unit of the program’s functionality. At the code level, an origin is a set of code paths all with the same starting point (i.e., the entry point) and data pointers (i.e., the origin attributes). In this way, origins divide a program into different sets of code paths according to their semantics where each origin represents a separate semantic domain. While origins can be specified by code annotations, we aim to extract them automatically from common code patterns in multithreaded and event-driven programs. Our system identifies two kinds of origins automatically by default: threads and event handlers. Finding static threads is not difficult in practice because threads are almost always explicitly defined, either at the language level or through common APIs such as POSIX Threads (Pthreads) and Runnable and Callable interfaces in Java. Finding event handlers relies on code patterns such as Linux system call interfaces (all with prefix __x86_sys_), Android callbacks (onReceive and onEvent), and popular event-driven frameworks (Node.js and REST APIs). In cases where threads or events are implicit, such as customized user-level threads, developers may be willing to provide annotations to mark origins, since customized threads are likely to be an important aspect of the target application.

For Java and Pthread-based C/C++ programs, we automatically identify the methods in Table 1 as the origin entry points, which are frequently used to run code in parallel or handle an event. We then reason about the origin attributes in order to distinguish different origins with the same entry point but different data. The origin attributes can be inferred at two places:

- **Origin Allocation** is the allocation site of a receiver object of an origin entry point. The attributes include the arguments passed to the allocation site. For example, o4 (line 5) is an origin allocation in Figure 2, which is the receiver object of the entry point start() of Origin T1. As its arguments, s and op1 are the origin attributes of T1.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Event handlers</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Thread.start()</td>
<td>actionPerformed(...)</td>
</tr>
<tr>
<td>java.lang.Runnable.run()</td>
<td>onMessageEvent(...)</td>
</tr>
<tr>
<td>java.util.concurrent.Callable.call()</td>
<td>handleEvent(...)</td>
</tr>
<tr>
<td>pthread_create(...)</td>
<td>onReceive(...)</td>
</tr>
</tbody>
</table>

* More details for Android events are in Section 4.2.

| Table 1. The origin entry points identified by O2. |

3 Origin-Sensitive Analyses

In this section, we first present origins, OPA and OSA. The use of OPA and OSA enables a more precise pointer analysis and identification of shared- and local-memory accesses by threads and events. Beyond race detection, OPA and
Table 2. The OPA rules for Java. Consider the following statements are in method \( m \) with Origin \( O_i \), denoted \( \langle m, O_i \rangle \). The edges \( \rightarrow \) are in the PAG and \( \leadsto \) in the call graph.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Points-to Edge &amp; Call Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>❶ ( x = \text{new} \ C() )</td>
<td>( \langle (c, O_j) \rangle \rightarrow (x, O_i) )</td>
</tr>
<tr>
<td>❷ ( x = y )</td>
<td>( \langle (y, O_i) \rangle \rightarrow (x, O_i) )</td>
</tr>
<tr>
<td>❸ ( x.f = y )</td>
<td>( \forall (o, O_k) \in \text{pts}(x, O_i)) \langle (y, O_i), f \rangle \rightarrow (o, O_k) )</td>
</tr>
<tr>
<td>❹ ( x = y.f )</td>
<td>( \forall (o, O_k) \in \text{pts}(\langle y, O_i \rangle) \langle (o, O_k), f \rangle \rightarrow (x, O_i) )</td>
</tr>
<tr>
<td>❸ ( x[idx] = y )</td>
<td>( \forall (o, O_k) \in \text{pts}(\langle x, O_i \rangle) \langle (y, O_i), \ast \rangle \rightarrow (o, O_k) )</td>
</tr>
<tr>
<td>❷ ( x = y[idx] )</td>
<td>( \forall (o, O_k) \in \text{pts}(\langle y, O_i \rangle) \langle (o, O_k), \ast \rangle \rightarrow (x, O_i) )</td>
</tr>
<tr>
<td>⁹ ( x.f(a_1, .., a_n) )</td>
<td>( \forall (o, O_k) \in \text{pts}(\langle y, O_i \rangle) \langle (f', O_i) \rangle \rightarrow (f, O_i) )</td>
</tr>
<tr>
<td>( \text{//non-origin entry} )</td>
<td>( \langle a_h, O_i \rangle \rightarrow \langle p_h, O_i \rangle, \text{where } 1 \leq h \leq n )</td>
</tr>
<tr>
<td></td>
<td>( \langle f'_\text{hist}, O_i \rangle \rightarrow (x, O_i) )</td>
</tr>
<tr>
<td></td>
<td>( \text{add call edge } (m, O_i) \leadsto (f', O_i) )</td>
</tr>
<tr>
<td>( x = \text{new} \ O(b_1, .., b_n) )</td>
<td>( \langle (o, O_j) \rangle \rightarrow (\text{init}_\text{hist}, O_j) )</td>
</tr>
<tr>
<td>( \text{//origin allocation} )</td>
<td>( \langle (o, O_j) \rangle \rightarrow (\text{init}, O_j) )</td>
</tr>
<tr>
<td></td>
<td>( \langle b_h, O_i \rangle \rightarrow \langle p_h, O_i \rangle, \text{where } 1 \leq h \leq n )</td>
</tr>
<tr>
<td></td>
<td>( \text{add call edge } (m, O_i) \leadsto (\text{init}, O_j) )</td>
</tr>
<tr>
<td>( x.\text{entry}(c_1, .., c_n) )</td>
<td>( \forall (o, O_j) \in \text{pts}(\langle x, O_i \rangle) \langle (\text{entry}', O_j) \rangle \rightarrow \langle \text{entry}', O_j \rangle )</td>
</tr>
<tr>
<td>( \text{//origin entry point} )</td>
<td>( \langle c_h, O_i \rangle \rightarrow \langle p_h, O_i \rangle, \text{where } 1 \leq h \leq n )</td>
</tr>
<tr>
<td></td>
<td>( \text{add call edge } (m, O_i) \leadsto (\text{entry}', O_j) )</td>
</tr>
</tbody>
</table>

- **Origin Entry Point** may be invoked with parameters, of which pointers are also included in the attributes. For example, \( \text{onReceive(context, intent)} \) is an entry point of \( \text{BroadcastReceiver} \) in Android apps, where \( \text{intent} \) contains the incoming message and \( \text{context} \) represents the environment the message is sent from.

3.2 Origin-Sensitive Pointer Analysis

Interestingly, reasoning about pointers and heap objects can be done simultaneously with origin-sensitive pointer analysis (OPA). Pointer analysis typically uses the **pointer assignment graph** (PAG) [30] to represent points-to relations between pointers and objects. To achieve good precision, the PAG constructed by OPA is built together with the call graph (a.k.a. on-the-fly pointer analysis [30]). The key difference is that the context of pointers in OPA is represented by origins. The rules of OPA for Java are summarized in Table 2. A set of similar rules can be inferred for other programming languages.

---

**Intra-Origin Constraints.** Statements ❶-❹ are in method \( \langle m, O_i \rangle \), and all the program elements created by them share the same origin \( O_i \) to indicate where they are originated from. For example, the allocated object by statement ❶ is represented as \( (o, O_i) \) and assigned to pointer \( (x, O_i) \), and their relation is represented by a points-to edge \( (o, O_i) \rightarrow (x, O_i) \) in the PAG.

An object field pointer is distinguished by the origin of its receiver object. For statement ❺, each receiver object \( (o, O_k) \) corresponds to an object field pointer \( (o, O_k).f \) that points to \( (x, O_i) \). Note that a pointer and its points-to objects may have different origins, which shows how data flows across origins.

Although there exists a large body of work that can infer the content of arrays, analyzing array index \( idx \) in statements ❻❼ is statically undecidable and expensive. Hence, we do not distinguish different array indexes: array objects are modeled as having a single field that may point to any value stored in the array, e.g., \( x[idx] = y \) is modeled as \( x.* = y \). This model simply captures objects allocated by different origins that flow to an array without any complex index analysis. Besides, our algorithm can be easily integrated with existing array index analysis algorithms with no conflict.

A non-origin entry method call ❹ invokes a target method \( f' \) within the same origin \( O_j \) as its caller, even though its receiver object \( (o, O_k) \) might be allocated from a different origin \( O_k \). To determine a virtual call target and its context (e.g., the call on line 13 in Figure 3), we use the type of its receiver object \( o \) and the origin \( O_j \) of which thread/event-handler executes the target. The target’s origin must be consistent with its caller’s, regardless of whether it is an entry point or not.

**Inter-Origin Constraints.** We switch contexts from current origin \( O_i \) to a new origin \( O_j \) for an origin allocation ❺ and an origin entry point ❹. Note that, to avoid false aliasing introduced by thread creations, we analyze every origin allocation in its new origin instead of its parent origin where it should be executed. Figure 3 shows two origins (Ta and Tb) allocated in Origin
Tmain. The two origin allocations share the same super constructor T(). If we analyze them in their parent origin Tmain, only one object $o_f$ will be allocated for field $f$ on line 14. This will cause $pts(o_f, f) = pts(o_b, f) = \{ o_f, Tmain \}$, which introduces false aliasing. To eliminate such imprecision, OPA creates two objects, $\langle o_f, Ta \rangle$ and $\langle o_f, Tb \rangle$, for each $f$ under each origin by forcing the context switch at origin allocations on lines 2 and 3.

To identify origin allocations on-the-fly, we check the type of the allocated object against the classes in Table 1, i.e., if it implements interface Runnable or event handler handleEvent(). Context switch on \( \textcircled{O} \) can efficiently separate data flows to the same origin constructor but from different allocation sites, e.g., both op1 and op2 flow to the constructor of T in Figure 2. Specifically, in this example a new and unique origin $O_j$ is created for this new allocation $(o, O_j)$.

Both $\textcircled{O}$ and $\textcircled{S}$ designate the attributes for the new origin $O_j$, including constructor arguments $(b_1, ..., b_n)$ and method parameters $(c_1, ..., c_n)$, which reveal significant information of the accessed data and the origin behavior. To reflect the own-ship of the actual parameters use $O_i$ as their contexts and the formal ones use $O_j$. Meanwhile, call edges are added in the call graph, e.g., $(m, O_j) \leadsto \langle \text{init}, O_j \rangle$ for $\textcircled{S}$ and $(m, O_i) \leadsto \langle \text{entry'}, O_j \rangle$ for $\textcircled{O}$.

**Wrapper Functions and Loops.** In practice, both $\textcircled{O}$ and $\textcircled{S}$ may be hidden in a wrapper function (e.g., cross-platform thread wrappers) invoked by multiple call sites. To efficiently separate such origins, we can extend the entry point of an origin to also include its k-call-site. In our tools, we set $k=1$. Meanwhile, for an origin allocated in a loop, we always create two origins with identical attributes but different origin IDs.

**K-Origin-Sensitivity.** In the same spirit as k-CFA and k-obj, a sequence of origins can be concatenated, denoted as $k$-origin. For example, a method $m()$ can be denoted as follows:

$$\langle m, [O_1, O_2, ..., O_{k-1}, O_k] \rangle$$

where $m()$ is invoked within Origin $O_k$ that has a parent origin $O_{k-1}$, etc. k-origin can further improve the precision when a pointer propagates across nested origins, and we observed such cases in many of our evaluated programs (e.g., Redis) where thread creations are nested.

### Table 3. The time complexity of different pointer analyses.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Worst-Case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-context</td>
<td>$O(p \times h^2)$</td>
</tr>
<tr>
<td>heap</td>
<td>$O(p^1 \times h^2)$</td>
</tr>
<tr>
<td>2-CFA + heap</td>
<td>$O(p^2 \times h^2)$</td>
</tr>
<tr>
<td>2-obj + heap</td>
<td>$O(p^2 \times h^2)$</td>
</tr>
<tr>
<td>1-origin + heap</td>
<td>$O(p^3 \times h^2)$</td>
</tr>
</tbody>
</table>

**Time Complexity.** Table 3 summarizes the worst-case time complexity of different pointer analysis algorithms according to [65], where $p$ and $h$ are the number of statements and heap allocations, respectively. The complexity of k-CFA and k-obj varies according to the context depth $k$. However, their worst-case complexity can be doubly exponential [54]. The selective context-sensitive techniques [16, 17, 33, 34, 38, 61] are also bounded by the context depth and have the same worst-case complexity as their corresponding full k-CFA and k-obj algorithms.

The 1-origin has the same complexity as 1-call-site-sensitive heap analysis (denoted heap). But the number of operations is increased linearly by a factor ($#O \times O\%$), where $#O$ is the
number of origins and O% is the ratio between the average number of statements within an origin and the total number of program statements. The ratio is small (<10%) for most applications, according to our experiments in Section 5.

3.3 Origin-Sharing Analysis

Based on OPA, our origin-sharing analysis (OSA) uses an origin-sensitive heap abstraction and automatically identifies memory objects shared by different origins. A sample output is shown in Figure 2(d). A key in OSA is to track the objects accessed in the code path of each origin by leveraging OPA. Consistently with OPA, OSA is sound, interprocedural, and field-sensitive. More importantly, OSA is more scalable than conventional thread-escape analysis techniques [10, 19, 55]—it only requires a linear scan of the program statements.

As depicted in Algorithm 1, we traverse the program statements starting from the main entry method. There are three kinds of statements relevant to OSA:

- For each object field access (statement ❶ in Table 2), we query OPA to find all the possible allocated objects that the base reference may point to. Each object has an origin which is represented by its allocation site together with an origin. For each such origin O, we call the procedure ComputeOriginSharing(s, f, O, isWrite) to compute if the field access is shared by multiple origins or not. In ComputeOriginSharing, we maintain for each access a set of write origins and a set of read origins, retrieved by GetWriteOrigins and GetReadOrigins, respectively. If a field access in a statement is accessed by more than one origin, and with at least one of them is a write, we mark the access as origin-shared. For static field accesses, the procedure is similar except that each static field is directly encoded into a unique signature including the class name and the field index.

- For array accesses (statement ❷ in Table 2), we handle array accesses similar to that of object field accesses, but query about its field * representing all array elements.

- For method invocation statements (statement ❸ in Table 2) (the receiver object is also included in the arguments args), we use OPA again to determine the possible callees methods and traverse their statements.

Compared to thread-escape analysis, OSA has the following key advantages:

- OSA is more general than thread-escape analysis since an origin can represent a thread or an event;

- OSA is more precise than thread-escape analysis. For example, static variables (and any object that is reachable from static variables) are often considered as thread-escaped. However, certain static variables may only be used by a single thread. OSA can distinguish such cases.

- While standard thread-escape analysis algorithms do not directly work for array accesses (because they have no information about array aliases), OSA can distinguish if an access a[i] is an origin-shared array object or not through reasoning about the points-to set of a.*

- OSA also identifies origin-shared array object or not through reasoning about the points-to set of a.*

access a[i] is an origin-shared array object or not through reasoning about the points-to set of a.*

- OSA also identifies origin-shared array object or not through reasoning about the points-to set of a.*

4 Static Data Race Detection

In O2, we model both threads and events statically as functional units, each represented by a static trace of memory accesses and synchronization operations. Our race detection engine uses hybrid happens-before and lockset analyses similar to most prior work on dynamic race detection [43] (although ours is static). More specifically, our detection represents happens-before relations by a static happens-before graph (SHB) [35, 73], which is designed to efficiently compute incremental changes from source code.

We modify the graph with origins as shown in Table 4. We record the field/array read and write accesses for statements ❷-❾ by creating read and write nodes. For statement ❶, we create a method call node (call) with two happens-before (HB) edges (denoted ⇒): one points from the call node to the first node (fstart) of its target method f within the same origin O, the other points from the last node (fend) of (f, O) to the next node after the call (callnext). Intra-origin HB edges are created by pointing from one intra-origin node to another in their statement order.

For lock operation ❾, we create lock and unlock nodes to maintain the current lockset. For Java programs, we consider synchronized blocks and methods. For C/C++ programs, O2 currently only considers monitor-style locks (including both standard pthread mutexes and customized locks through configurations). And we aim to support atomic

<table>
<thead>
<tr>
<th>Statement</th>
<th>Intra-Origin Happen-before Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.ENTRY</td>
<td>(x, ENTRY) ∈ pts((x, O), ENTRY)</td>
</tr>
<tr>
<td>x.WRITE</td>
<td>(x, WRITE) ∈ pts((x, O), WRITE)</td>
</tr>
<tr>
<td>x.READ</td>
<td>(x, READ) ∈ pts((x, O), READ)</td>
</tr>
<tr>
<td>x.HB</td>
<td>(x, HB) ∈ pts((x, O), HB)</td>
</tr>
<tr>
<td>x.call</td>
<td>(x, call) ∈ dispatch((x, O), call)</td>
</tr>
<tr>
<td>x.unlock</td>
<td>(x, unlock) ∈ dispatch((x, O), unlock)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statement</th>
<th>Inter-Origin Happen-before Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.entry</td>
<td>(x, entry) ∈ dispatch((x, O), entry)</td>
</tr>
<tr>
<td>x.join</td>
<td>(x, join) ∈ dispatch((x, O), join)</td>
</tr>
<tr>
<td>x.HB</td>
<td>(x, HB) ∈ pts((x, O), HB)</td>
</tr>
</tbody>
</table>

Table 4. SHB Graph with Origins: the following statements are in method m() with Origin O_i.
For calls to an origin entry point $O$, we create an origin entry node (entry) to represent the start of a new origin $O_j$ from its parent origin $O_i$. And we add an inter-origin HB edge pointing to the first node (origin$_{first}$) of $O_j$. For thread join statement $O$, we create a join node (join) to indicate the end of $O_j$ that finally joins to $O_i$. An inter-origin HB edge is created from the last node (origin$_{last}$) of current origin ($O_j$) to the join node.

Existing static race detection (such as [73]) typically checks each pair of two conflict accesses from different threads: run a depth-first search (or breadth-first search) starting from one access and vice versa to check their happens-before relation on the SHB graph, and compute the locksets for both accesses to check whether they have common lock guards.

However, the efficiency is limited by the redundant work in graph traversals and lockset retrievals for all pairs of memory accesses. The straw man approach cannot scale to real-world programs which can generate large SHB graphs with millions of memory accesses.

### 4.1 Three Sound Optimizations

To address the performance challenges, we develop the following sound optimizations:

**Check Happens-Before Relation.** We only create inter-origin HB edges in the SHB graph. Instead of creating intra-origin HB edges, we assign a unique integer ID to each node, which is monotonically increased during the SHB construction. Therefore, we convert the traversal of visiting all intra-origin nodes along HB edges to a constant time integer comparison.

**Check Lockset.** Intuitively, a list of locks is associated with each memory access node in the SHB graph in order to represent the mutex protection. We observe that the number of different combinations among mutexes is much smaller than the number of conflict memory accesses we need to check. Therefore, we assign each combination of mutexes (including the empty lockset) a canonical ID and associate each access node with such an ID. This not only reduces the memory for storing the SHB graph, but also speeds up the lockset checking process. All memory accesses with an identical lockset ID, or different IDs corresponding to overlapping locksets, are protected by the same lock(s), and the intersection of the IDs between two locksets can be cached for later checks.

**Lock-Region-based Race Detection.** We observe that a synchronization block or method often guards a large sequence of memory accesses on the same origin-shared object(s) ($o_r$), which incurs redundant race checking. Instead, we treat all the memory accesses on $o_r$ within the same lock region as a single memory access on $o_r$, and check races on that single access once. This is sound because their happens-before relations and locksets are exactly the same. This optimization significantly boosts O2’s performance by reducing the number of memory access pairs for detecting data races.

### 4.2 Unify Threads with Android Events

Mobile applications are a representative class of modern software that contains complex interactions between threads and events. For instance, in Android apps, there are hundreds of different types of events that can be created from the Activity lifecycles, callbacks, UI, or the system services [62]. Meanwhile, the app logic may create any number of normal Java threads and AsyncTask to improve performance.

Keen readers may wonder that O2 may not work well for mobile apps, since such event-driven applications will generate a large number of origins. However, as we will show in our experiments, O2 scales well on Android apps, because Android apps often have short-duration events that explore only a small fraction starting from the entry points.

O2 detects data races in Android apps through the following treatments. In Android apps, there is no explicit main method as in other Java programs that can be used as the analysis entry of O2. Instead, we automatically generate an analysis harness from the main Activity of every Android app (i.e., the home screen). The main activity can be identified by parsing the file AndroidManifest.xml within each Android apk.

Our tool treats each event handler as an origin entry. Once we hit a startActivity() or startActivityForResult(), we create a harness for the activity being started and analyze the new harness. All lifecycle event handlers are treated as method calls, while the normal event handlers are viewed as origin entries in OPA and SHB graph construction. Since all events are handled by the main thread [2], we protect the memory accesses within all the event handlers by one global lock, so that no false positive among event handlers will be reported by O2.

### 4.3 Other Implementation Details

**Sequential and Relaxed Memory Models.** Different from sequential consistency, a relaxed memory model may reorder certain reads and writes in the same thread and different threads may see different orders. O2 works for both sequential and relaxed memory models. The reason is that the SHB graph captures inter-origin happens-before relations at synchronization sites, and it does not assume a global ordering of reads and writes. Hence, our happens-before relations already relax the ordering constraints for reads and writes from the same origin.

**Cross-Module and External Pointers.** For C/C++, O2 always links the IR files into a single LLVM module and performs the analysis based on the whole module. Meanwhile,
**Table 5.** Performance comparison on JVM programs (in sec.). The left part compares OPA with other pointer analyses. The right part compares O2 with other race detection algorithms. The slowdown (in red) is normalized with 0-ctx as the baseline.

<table>
<thead>
<tr>
<th>App</th>
<th>#-ctx</th>
<th>#O</th>
<th>OPA 1-CFA</th>
<th>2-CFA</th>
<th>1-obj</th>
<th>2-obj</th>
<th>0-ctx</th>
<th>O2</th>
<th>1-CFA</th>
<th>2-CFA</th>
<th>1-obj</th>
<th>2-obj</th>
<th>RacerD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avorla</td>
<td>13.42</td>
<td>4</td>
<td>15.56</td>
<td>17.81</td>
<td>56.06</td>
<td>56.30</td>
<td>20.70</td>
<td>17.85</td>
<td>-14</td>
<td>30.63</td>
<td>0.48x</td>
<td>615.42</td>
<td>29x</td>
</tr>
<tr>
<td>Batik</td>
<td>7.22</td>
<td>4</td>
<td>9.83</td>
<td>246.60</td>
<td>4h</td>
<td>-</td>
<td>14.47</td>
<td>14.93</td>
<td>3%</td>
<td>84.01</td>
<td>4.81x</td>
<td>2648</td>
<td>182x</td>
</tr>
<tr>
<td>Eclipse</td>
<td>5.03</td>
<td>4</td>
<td>6.52</td>
<td>8.43</td>
<td>8.71</td>
<td>11.84</td>
<td>7.21</td>
<td>8.03</td>
<td>11%</td>
<td>20.35</td>
<td>1.82x</td>
<td>40.36</td>
<td>460x</td>
</tr>
<tr>
<td>H2</td>
<td>49.95</td>
<td>3</td>
<td>111.37</td>
<td>192.71</td>
<td>1397</td>
<td>4h</td>
<td>58.13</td>
<td>169.63</td>
<td>192%</td>
<td>263.00</td>
<td>3.52x</td>
<td>3208</td>
<td>54x</td>
</tr>
<tr>
<td>Jython</td>
<td>25.77</td>
<td>4</td>
<td>66.34</td>
<td>16.09</td>
<td>58.85</td>
<td>4h</td>
<td>163.49</td>
<td>537.63</td>
<td>229%</td>
<td>100.35</td>
<td>0.39x</td>
<td>172.46</td>
<td>6.05x</td>
</tr>
<tr>
<td>Lutindex</td>
<td>10.23</td>
<td>3</td>
<td>15.73</td>
<td>16.74</td>
<td>26.38</td>
<td>4h</td>
<td>14.43</td>
<td>20.19</td>
<td>40%</td>
<td>31.62</td>
<td>1.19x</td>
<td>1634</td>
<td>112x</td>
</tr>
<tr>
<td>Lusarch</td>
<td>10.06</td>
<td>3</td>
<td>5.66</td>
<td>31.60</td>
<td>2384</td>
<td>4h</td>
<td>7.09</td>
<td>7.99</td>
<td>13%</td>
<td>33.79</td>
<td>3.77x</td>
<td>2401</td>
<td>338x</td>
</tr>
<tr>
<td>Pmd</td>
<td>5.50</td>
<td>3</td>
<td>5.75</td>
<td>6.04</td>
<td>8.00</td>
<td>4h</td>
<td>25.07</td>
<td>13.32</td>
<td>47%</td>
<td>57.21</td>
<td>1.28x</td>
<td>122.93</td>
<td>190x</td>
</tr>
<tr>
<td>Sunflow</td>
<td>3.13</td>
<td>9</td>
<td>5.68</td>
<td>5.08</td>
<td>5.44</td>
<td>5.12</td>
<td>20.67</td>
<td>26.01</td>
<td>25%</td>
<td>297.05</td>
<td>23x</td>
<td>2408</td>
<td>116x</td>
</tr>
<tr>
<td>Tomcat</td>
<td>3.77</td>
<td>7</td>
<td>10.18</td>
<td>30.47</td>
<td>2312</td>
<td>5.89</td>
<td>7.58</td>
<td>16.87</td>
<td>123%</td>
<td>66.48</td>
<td>7.77x</td>
<td>2829</td>
<td>372x</td>
</tr>
<tr>
<td>Tradebeans</td>
<td>4.96</td>
<td>3</td>
<td>6.05</td>
<td>6.76</td>
<td>7.54</td>
<td>4h</td>
<td>8.19</td>
<td>12.05</td>
<td>47%</td>
<td>16.49</td>
<td>1.01x</td>
<td>111.80</td>
<td>13x</td>
</tr>
<tr>
<td>Tradesoap</td>
<td>6.25</td>
<td>3</td>
<td>7.44</td>
<td>8.33</td>
<td>9.31</td>
<td>4h</td>
<td>10.22</td>
<td>10.77</td>
<td>5%</td>
<td>24.32</td>
<td>1.38x</td>
<td>149.53</td>
<td>14x</td>
</tr>
<tr>
<td>Xalan</td>
<td>31.30</td>
<td>3</td>
<td>35.73</td>
<td>65.51</td>
<td>3922</td>
<td>213.99</td>
<td>34.71</td>
<td>42.87</td>
<td>24%</td>
<td>79.33</td>
<td>1.29x</td>
<td>5722</td>
<td>164x</td>
</tr>
</tbody>
</table>

*O*: The number of origins detected during the analysis. - : Time out. *: RacerD could not run successfully due to compiler errors.

**Table 6.** Performance comparison on C/C++ benchmarks (in sec.). The slowdown (SD) is normalized with 0-ctx as the baseline.

<table>
<thead>
<tr>
<th>App</th>
<th>#KLOC</th>
<th>Metrics</th>
<th>0-ctx</th>
<th>O2</th>
<th>2-CFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>20.4</td>
<td>Time/SD</td>
<td>5.3</td>
<td>5.8</td>
<td>9/7</td>
</tr>
<tr>
<td>(O = 12)</td>
<td></td>
<td>#Pointer</td>
<td>8.40</td>
<td>12</td>
<td>883</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Object</td>
<td>2.42</td>
<td>2.46</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Edge</td>
<td>5.35</td>
<td>10</td>
<td>415</td>
</tr>
<tr>
<td>Redis</td>
<td>116</td>
<td>Time/SD</td>
<td>9.3</td>
<td>15</td>
<td>0.61</td>
</tr>
<tr>
<td>(O = 15)</td>
<td></td>
<td>#Pointer</td>
<td>44.55</td>
<td>54</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Object</td>
<td>14.4589</td>
<td>14</td>
<td>913</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Edge</td>
<td>598.981</td>
<td>963</td>
<td>654</td>
</tr>
<tr>
<td>Sqlite3</td>
<td>245</td>
<td>Time/SD</td>
<td>213</td>
<td>273</td>
<td>28%</td>
</tr>
<tr>
<td>(O = 3)</td>
<td></td>
<td>#Pointer</td>
<td>57.65</td>
<td>61</td>
<td>796</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Object</td>
<td>10.093</td>
<td>10</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Edge</td>
<td>7.909626</td>
<td>819</td>
<td>879</td>
</tr>
</tbody>
</table>

**Avg.** 126 Time/SD 75.8 97.9/30%

In Table 5, there is always a default origin (starting from the main entry point), so we do not have to deal with cross-module pointers. For JVM applications, O2 extends WALA’s ZeroOneCFA to analyze all bytecode-level pointers loaded by the application classloader. When a pointer is passed from an external function call for which the IR file does not exist, we will create an anonymous object for that pointer.

## 5 Experiments

We evaluated O2 on a large collection of real-world, widely-used distributed systems (e.g., ZooKeeper and HBase), Android apps (e.g., Firefox and Telegram), key-value stores (e.g., Redis/RedisGraph, Memcached and TDengine), network controllers (Open vSwitch, OVS), lock-free algorithms (e.g., cpqueue and mrlock), as well as the Linux kernel.

### 5.1 Performance for Java, Android and C/C++

#### 5.1.1 OPA vs Other Pointer Analyses

The left part of Table 5 summarizes the performance of different pointer analysis algorithms on the JVM benchmarks, including Dacapo [7], a collection of popular Android apps and distributed systems. Overall, OPA significantly outperforms 1-CFA, 2-CFA, 1-obj and 2-obj by 1x, 152x, 390x and 465x speedup (on average) respectively, and OPA has only a small performance slowdown compared to the context-insensitive baseline (denoted 0-ctx, 1.76x on average). In particular, the majority of benchmarks running 1-obj and 2-obj cannot terminate within 4 hours.
Table 7. Performance and #Shared memory accesses (#S-access) of OSA.

<table>
<thead>
<tr>
<th>App</th>
<th>#S-access</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>16</td>
<td>16.72s</td>
</tr>
<tr>
<td>Batik</td>
<td>293</td>
<td>7.79s</td>
</tr>
<tr>
<td>Eclipse</td>
<td>343</td>
<td>9.22s</td>
</tr>
<tr>
<td>H2</td>
<td>2,207</td>
<td>2.3min</td>
</tr>
<tr>
<td>Jython</td>
<td>13,121</td>
<td>4.5min</td>
</tr>
<tr>
<td>Luindex</td>
<td>2,001</td>
<td>1.6min</td>
</tr>
<tr>
<td>Lusearch</td>
<td>252</td>
<td>7.01s</td>
</tr>
<tr>
<td>Pmd</td>
<td>300</td>
<td>8.57s</td>
</tr>
<tr>
<td>Sunflow</td>
<td>1,603</td>
<td>10.15s</td>
</tr>
<tr>
<td>Tomcat</td>
<td>700</td>
<td>15.39s</td>
</tr>
<tr>
<td>Tradebeans</td>
<td>45</td>
<td>7.48s</td>
</tr>
<tr>
<td>Tradesoap</td>
<td>37</td>
<td>9.12s</td>
</tr>
<tr>
<td>Xalan</td>
<td>14</td>
<td>1.2min</td>
</tr>
</tbody>
</table>

Table 8. #Races detected by O2 and D4 utilizing different pointer analyses. The percentages of reduced races (in red) are normalized with 0-ctx as the baseline. The comparison between O2 and RacerD (v1.0.0) is shown separately on the right, due to different scale.

<table>
<thead>
<tr>
<th>App</th>
<th>0-ctx</th>
<th>O2</th>
<th>1-CFA</th>
<th>2-CFA</th>
<th>1-obj</th>
<th>2-obj</th>
<th>O2 RacerD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>12,633</td>
<td>3899.7%</td>
<td>45/99.6%</td>
<td>45/99.6%</td>
<td>47/99.6%</td>
<td>-</td>
<td>38</td>
</tr>
<tr>
<td>Batic</td>
<td>4,369</td>
<td>186/95.7%</td>
<td>4,229/3.2%</td>
<td>640/85.4%</td>
<td>-</td>
<td>-</td>
<td>186</td>
</tr>
<tr>
<td>Eclipse</td>
<td>958</td>
<td>79/99.3%</td>
<td>94/1.5%</td>
<td>822/14.2%</td>
<td>945/1.4%</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>H2</td>
<td>9,698</td>
<td>2,817/71.0%</td>
<td>7,832/19.2%</td>
<td>6,322/34.8%</td>
<td>-</td>
<td>-</td>
<td>2,817</td>
</tr>
<tr>
<td>Jython</td>
<td>7,997</td>
<td>3,651/53.4%</td>
<td>2,402/70.0%</td>
<td>2,358/70.5%</td>
<td>-</td>
<td>-</td>
<td>3,651</td>
</tr>
<tr>
<td>Luindex</td>
<td>3,218</td>
<td>1,792/44.3%</td>
<td>2,821/12.3%</td>
<td>2,271/29.4%</td>
<td>-</td>
<td>-</td>
<td>1,792</td>
</tr>
<tr>
<td>Lusearch</td>
<td>367</td>
<td>341/39.9%</td>
<td>538/5.1%</td>
<td>494/12.9%</td>
<td>529/6.7%</td>
<td>526/7.2%</td>
<td>341</td>
</tr>
<tr>
<td>Pmd</td>
<td>307</td>
<td>256/16.6%</td>
<td>296/3.6%</td>
<td>293/4.6%</td>
<td>-</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td>Sunflow</td>
<td>9,238</td>
<td>1,925/72.9%</td>
<td>6,868/25.7%</td>
<td>5,899/36.1%</td>
<td>2,288/75.2%</td>
<td>-</td>
<td>1,925</td>
</tr>
<tr>
<td>Tomcat</td>
<td>751</td>
<td>307/59.1%</td>
<td>701/6.7%</td>
<td>693/7.7%</td>
<td>585/22.1%</td>
<td>575/23.4%</td>
<td>307</td>
</tr>
<tr>
<td>Tradebeans</td>
<td>193</td>
<td>75/61.1%</td>
<td>171/11.4%</td>
<td>168/13.0%</td>
<td>-</td>
<td>-</td>
<td>75</td>
</tr>
<tr>
<td>Tradesoap</td>
<td>264</td>
<td>64/75.8%</td>
<td>179/32.2%</td>
<td>177/33.0%</td>
<td>-</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>Xalan</td>
<td>6</td>
<td>1/83.3%</td>
<td>6/0.0%</td>
<td>6/0.0%</td>
<td>6/0.0%</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that the number of origins (denoted #O) in the evaluated Android apps is significantly larger than that in the other JVM applications, up to over a hundred origins in Telegram. However, OPA is still highly efficient, finishing in a few minutes in the worst case. Compared to the other algorithms, the performance of origin-sensitivity is comparable to 1-CFA (but much more precise by identifying thread-/event-local points-to constraints) and several orders of magnitude faster than 2-CFA, 1-obj and 2-obj.

The scalability of k-obj [41] and k-CFA [58] varies depending on the code. For most benchmarks, more objects are allocated when running k-obj than k-CFA, e.g., 2-CFA allocates 3357 objects for Tomcat, while 2-obj allocates 20679. Meanwhile, opposite cases exist, e.g., Avrora has 7369 objects for 1-CFA and 5848 for 1-obj.

Table 6 reports the performance for three C/C++ applications (Memcached, Redis and Sqlite3). OPA achieves up to 17x speedup over 2-CFA on Redis while only incurring 30% slowdown compared with 0-ctx. Moreover, 2-CFA got killed when running on Sqlite3 due to out of memory (OOM, 32GB) while OPA only imposes 28% slowdown. We note that O2 detected numerous real races in all three applications. We will elaborate the case of Memcached in Section 5.4.

5.1.2 OSA vs Escape Analysis. We compared OSA with an open-source escape analysis TLOA [19], which is integrated in the state-of-the-art static analysis framework Soot [1]. TLOA uses context-sensitive information flow analysis to decide whether a field can be accessed by multiple threads. Table 7 reports the number of thread-shared accesses for each benchmark computed by OSA, which has the same setting with the evaluation of OPA. OSA completes in 51s on average, while TLOA could not finish within the time limit for all the benchmarks. We further excluded JDK libraries and the benchmark-specific dependencies (e.g., antlr and asm for Jython). However, TLOA only finishes the analysis for Avrora in 90s, which generates an imprecise report with no thread-escape accesses. For the other benchmarks, TLOA still cannot finish within one hour, which is over 70x (on average) slower than OSA.

5.1.3 Race Detection Performance. The right part of Table 5 reports the performance for race detection including the time of running the corresponding pointer analysis. In summary, O2 achieves 70x speedup on average over the other context-sensitive detections. Among them, the most speedup (1461x on detection and 568x in total) is on Tomcat when comparing with 2-obj. Compared to 0-ctx, O2 is only 2.81x slower on average, and it is even faster for some applications (Avrora and Pmd), due to the much improved precision of origin-shared memory accesses.

O2 vs RacerD. We also compared O2 with RacerD (from the latest release Infer v1.0.0) in Table 5. RacerD did not complete the detection for 9 out of the 27 benchmarks, due to dependency limitation of the benchmark or compilation errors. For example, Eclipse has a complex building procedure, Sipdroid requires Android command which RacerD does not support, and other Android benchmarks involve legacy SDK that could not be resolved. O2 (69s on average) and RacerD (71s on average) have similar performance on Dacapo benchmarks, while RacerD is 90% slower on average on the two Android benchmarks (i.e., K-9 Mail and Firefox Focus). For the four distributed systems, O2 (48min, including the execution of a whole program pointer analysis), has 9.6x slowdown on average comparing with RacerD (5min). We also tested RacerD on the three C/C++ programs in Table 6. However, RacerD could not run successfully on Memcached and Redis, and it reports no violations on Sqlite3.
5.2 Precision of Origin-Sensitivity and O2

Tables 8 and 9 report the number of detected races with different pointer analyses. We use the number of reported races as the metric to evaluate the end-to-end precision of different analyses. The baseline is the open-source tool developed in D4 [35], which utilizes the points-to result from 0-ctx for static race detection.

In summary, O2 reduces warnings by 77% on average, while 1- and 2-CFA reduce 46% and 60%, and 1- and 2-obj reduce 35% and 19% respectively. For the majority of benchmarks, O2 reports significantly fewer races, e.g., Eclipse. For other benchmarks, O2 is much faster to achieve a similar precision. For example, O2 reports 38 races for Avrora, both 1- and 2-CFA report 45 races, and 1-obj reports 47 races (while being 60x slower than O2).

**O2 vs RacerD.** RacerD reports two types of thread safety violations in the evaluated benchmarks: (1) read/write races, and (2) unprotected write violations where a field access at a program location is outside of synchronization. To perform a fair comparison, we translate the violations in the RacerD report to a number of potential races: we add up the numbers of read/write races and of the pairs of conflict field accesses shown in unprotected writes. For distributed systems, we report the detected races in Table 9, since all the other context-sensitive detections run out of time (>4h).

On average, O2 reports 4.33x fewer warnings compared to RacerD (reduces false positives by 82% from Table 8). O2 detects new races in Firefox Focus, T Engine, OVS, Memcached and Redis; while RacerD either reports no races or cannot complete its detection on those programs. The majority of false positives reported by O2 are due to infeasible paths, which is inherent to static analysis tools.

5.4 New Races Found in Real-World Software

O2 has detected new races in every real-world code base we tested on, as summarized (partially) in Table 10. Most of them are due to a combination of threads and events. If considering events only or threads only, or considering them separately, these races will be missed. In the following, we elaborate the races found in several high-profile C/C++, Android apps, and distributed systems.

**Linux Kernel.** We evaluated O2 on the Linux kernel (commit 5b8b9d0c as of April 10th, 2020), compiled with tiny-config64, clang/LLVM 9.0. We define four types of origins: system calls with function prefix: __x64_sys_xx, driver functions over file operations (owner, llseek, read, write, open, release, etc), kernel threads with origin entries kthread_create_on_cpu() and kthread_create_on_node(), and interrupt handlers with origin entries request_threaded_irq() and request_irq(). There are 398 system calls included in our build. For each system call, we create two origins representing concurrent calls of the same system call, and a shared data pointer if the system call has a parameter that is a pointer (e.g., __x64_sys_mincore). In total, 1090 origins are created, including 796 from system calls and 294 from others.

In total, O2 detects 26 races in less than 8 minutes. We manually inspected all these races and confirmed that 6 are real races, 7 are potential races, and the other 13 are false positives. The 6 real races are all races to the Linux kernel bugzilla, and all of them have been confirmed at the time of writing. The 7 potential races are difficult to manually inspect due to very complex code paths involving the races. For the false positives, a majority of them are due to mis-recognition of spinlocks (such as arch_local_irq_save.38) or infeasible branch conditions which O2 does not handle. The code snippet below shows a real bug found by O2, which detects concurrent writes on the same element of array vdata (with array index CS_HRES_COARSE).

```c
1 void update_vsyscall_tz(void){// in class time.vsyscall
2 struct vdso_data *vdata = __arch_get_k_vdso_data();
3 vdata[CS_HRES_COARSE].tz_minuteswest = sys_tz.
4 //mit
5 ...
6 }
```
In addition, we found that among the 71459 allocated objects by the kernel (within the configured origins), 329 of them are origin-shared. And 1051 accesses are on origin-shared memory locations from a total of 36321 memory accesses. The result indicates that the majority of memory used by the kernel is origin-local, which can be beneficial to region-based memory management.

We also discovered that the system call paths do not create any new kernel threads or register interrupts. However, driver functions can do both operations. For example, the driver of GPIO requests a thread to read the events by the kernel API request_threaded_irq \(^1\). And the interrupt requests can create kernel threads by API kthread_create \(^2\).

**Memcached.** Memcached is a high performance multi-threaded event-based key/value cache store widely used in distributed systems. We applied O2 to commit 14521bd8 (as of May 12th, 2020). O2 is able to finish analyzing memcached within 5s, and reports 16 new races in total. All these races are previously unknown. We manually confirmed that 11 of them are real and the rest of them are potential races. A majority of the real races are on variables such as stats, settings, time_out, or stop_main_loop. There are also three races that are not on these variables but look more harmful. We reported the three races to the developers and all of them have been confirmed. The other five potential races all involve pointer aliases on queued items.

One of the reported races is shown below with the simplified code snippet:

```c
1 void *do_slabs_reassign(){ ... //event
2   if (slabclass[id].slabs > 1){
3       return cur; //RACE: missing lock
4   }
5 }
6 void *do_slabs_newslabs(){ ... //thread
7   pthread_lock();
8   p->slab_list[p->slabs++] = ptr; //with lock
9   pthread_unlock() ... }
```

The listed bug is related to Memcached’s slab-base memory allocation, which is used to avoid memory fragmentation by storing different objects using different slab classes based on their size. Since the accesses in the event handler is not protected by the lock, there is a data race between the event handler and all the running threads that try to allocate new slabs. Although another lock-protected check on the same variable is made later in the function, the data race can still lead to undefined behaviors. This case is interesting as it shows that unlike previous tools, which only reason about inter-thread races, O2 is able to unify events and threads to find races in complex programs that leverage both concepts for concurrency.

**FireFox Focus.** O2 was able to finish in 15s on FireFox Focus 8.0.15 (a privacy-focused mobile browser), and detected two previously unknown bugs (both reported in Bug-1581940) confirmed by developers from Mozilla. A simplified code snippet is presented below:

```java
1   // called from Gecko background thread
2   public synchronized IChildProcess bind(){
3       Context ctx = GeckoAppShell.getAppCtx(); //RACE
4   ... }
5   // called from MainActivity.onCreate()
6   @UiThread
7   public void attachTo(Context context){ ...
8       Context appCtx = context.getAppCtx();
9       if(!appCtx.equals(GeckoAppShell.getAppCtx())){
10          GeckoAppShell.setAppCtx(appCtx); //RACE
```

The code involves both FireFox Focus and FireFox’s browser engine, Gecko. Upon the app initialization, GeckoAppShell.getAppCtx() and GeckoAppShell.setAppCtx(appCtx) are called without synchronizations, one from Android UI thread (through onCreate event handler), the other from Gecko engine’s background thread. Although in reality, the creation order of UI thread and Gecko background thread keeps the race from happening, it is possible for Gecko engine to read an uninitialized application context thus leads to crash.

**Distributed Systems.** We discovered two new races in ZooKeeper 3.5.4 (reported in ZOOKEEPER-3819) and HBase 2.8.0 (reported in HBase-24374). O2 takes 4.5min to detect the new race in ZooKeeper by analyzing 40 threads and 88 events. The related code is shown below:

```java
1   // in class org.apache.zookeeper.server.DataTree
2   public void createNode(..., long ephemeralOwner){ ....
3       HashSet<String> list =ephemerals.get(ephemeralOwner);
4       if (list == null){ list = new HashSet<String>();
5           ephemerals.put(ephemeralOwner, list); }
6       synchronized(list){ //RACE
7           list.add(path); }
8   }
9   public void deserialize(InputArchive ia, String tag)
10      HashMap<String> list = ephemerals.get(OWNER);
11      if (list == null){ list = new HashSet<String>();
12          ephemerals.put(OWNER, list); }
13      list.add(path); //RACE: missing lock
```

These races are caused by interactions between threads and requests. ephemerals is a map in class DataTree to store the paths of the ephemeral nodes of a session. It is possible that a request of creating nodes for a session might arrive together with another request to deserialize the same session, and both requests are handled by different server threads (with super type ZooKeeperServer). The lock protection is missing on variable list on line 22, hence both threads can add paths concurrently to ephemerals. A worse case is that the

---

1/ [linux-stable/drivers/gpio/gpioiloh.c](https://github.com/torvalds/linux/blob/79723739e5de55632b3f269056b616110de90b0f/drivers/gpio/gpioiloh.c)\#1104:8
2/ [linux-stable/kernel/irq/manage.c](https://github.com/torvalds/linux/blob/8f766f96a4f945754031f2e211279797f52275ee/kernel/irq/manage.c)\#1282:7

---

**Table 10.** New Races Detected by O2 (Confirmed by Developers).

<table>
<thead>
<tr>
<th>Platform</th>
<th>Linux</th>
<th>TEngine</th>
<th>Redis/RedisGraph</th>
<th>OVS</th>
<th>cpqueue</th>
<th>mrllock</th>
<th>Memcached</th>
<th>Firefox</th>
<th>ZooKeeper</th>
<th>HBase</th>
<th>Tomcat</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Races</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

PLDI ’21, June 20–25, 2021, Virtual, Canada
Bozhen Liu*, Peiming Liu*, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang
two code snippets (line 4-7 and line 10-13) are not protected by common locks or mechanism from \texttt{ConcurrentHashMap}. Hence, the null checks from two threads on variable \texttt{list} may return null, but only one initialized set can be stored in \texttt{ephemerals} and all the paths added by another thread are missing. The race in \texttt{HBase} has the same reason as above, involving two concurrent accesses on a map, \texttt{keyProviderCache}, without locks from method \texttt{get\_Key\_Provider() (in class org.apache.hadoop.hbase.io.crypto.Encryption).

6 Conclusion and Future Work

We have presented \texttt{O2}, a new system for static race detection. \texttt{O2} is powered by a novel abstraction, \textit{origins}, that unifies threads and events to effectively reason about shared memory and pointer aliases. Our extensive evaluation with Java and C/C++ programs demonstrates the potential of \texttt{O2}, finding a large number of new races in mature open-source code bases and achieving dramatic performance speedups and precision improvement over existing static analysis tools. \texttt{O2} has been integrated into Coderrect, a commercial static analyzer [3]. In future work, we plan to implement and evaluate \texttt{O2} for other languages such as Golang, C# and Rust.

Acknowledgments

We thank our shepherd, Joseph Devietti, and the anonymous reviewers for their constructive feedback on earlier versions of this paper. We thank the Coderrect team for help with the tool’s development and evaluation. We also wish to thank Peter O’Hearn for many insightful discussions. This work was supported by NSF awards CCF-1552935 and CCF-2006450, and a Facebook Research Award to Jeff Huang.

References


