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

Yanze Li
Texas A&M University, USA
li14865@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

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.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**.

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454073

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

ACM Reference Format:
Bozhen Liu*, Peiming Liu*, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454073

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.

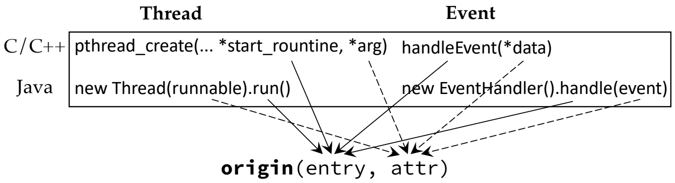

Thread
Event
C/C++
Java
pthread_create(... *start_rountine, *arg)
handleEvent(*data)
new Thread(runnable).run()
new EventHandler().handle(event)
origin(entry, attr)

Figure 1. An "origin" view of threads and events.

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 concept 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 automatically 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 conventional 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 identifying 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 compute 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 particular, 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.

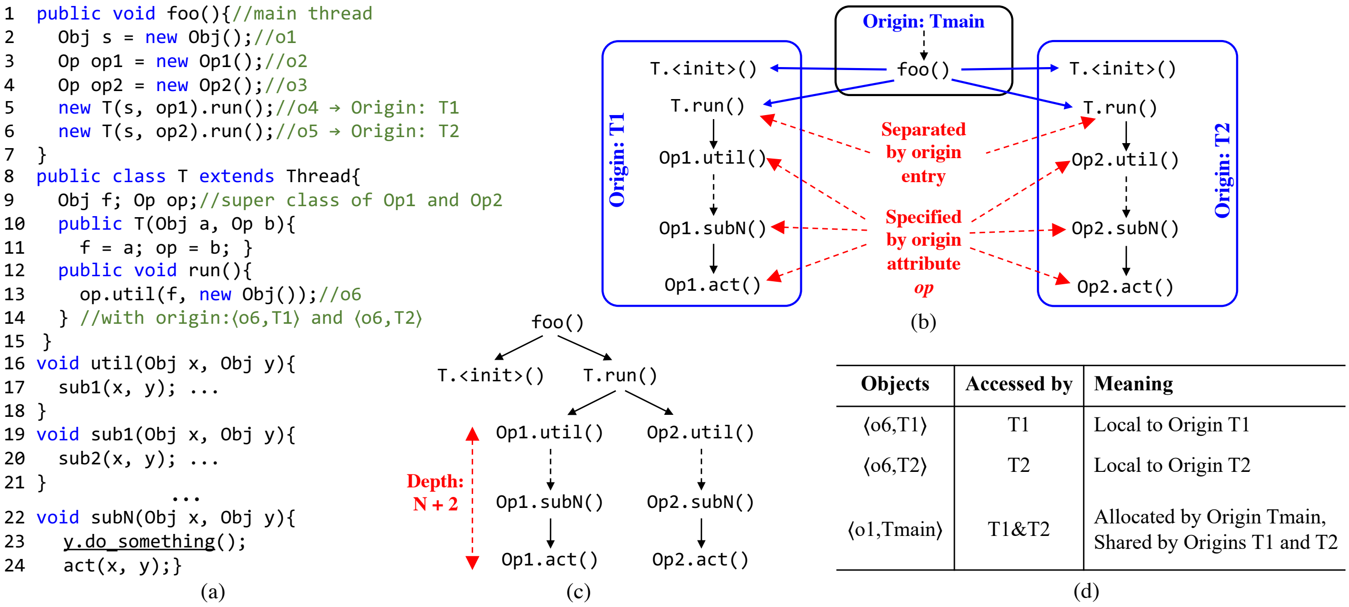
To illustrate these advantages, consider an example in Figure 2. To correctly infer that threads `T1` (line 5) and `T2` (line 6) do not access the same data on line 23, typically, a k-call-site analysis (denoted *k-CFA*) is performed, in which k is the depth of the call chain [59]. Additionally, a call-site-sensitive heap context is necessary to analyze the object allocation on line 13. This complexity is shared by k-object-sensitivity [41] (denoted *k-obj*), 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 reachability. 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 accesses;
- 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 evaluated programs in which O2 detects new races, RacerD either fails to find the races or cannot run due to compiler errors.



```
public void foo(){//main thread
  Obj s = new Obj();//o1
  Op op1 = new Op1();//o2
  Op op2 = new Op2();//o3
  new T(s, op1).run();//o4 → Origin: T1
  new T(s, op2).run();//o5 → Origin: T2
}
public class T extends Thread{
  Obj f; Op op;//super class of Op1 and Op2
  public T(Obj a, Op b){
    f = a; op = b; }
  public void run(){
    op.util(f, new Obj());//o6
  } //with origin:⟨o6,T1⟩ and ⟨o6,T2⟩
 }
void util(Obj x, Obj y){
  sub1(x, y); ...
}
void sub1(Obj x, Obj y){
  sub2(x, y); ...
}
      ...
void subN(Obj x, Obj y){
  y.do_something();
  act(x, y);}
```

(a)

(b)

(c)

Objects	Accessed by	Meaning
⟨o6,T1⟩	T1	Local to Origin T1
⟨o6,T2⟩	T2	Local to Origin T2
⟨o1,Tmain⟩	T1&T2	Allocated by Origin Tmain, Shared by Origins T1 and T2

(d)

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 owes 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 syntactical 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.

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 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.

Table 1. The origin entry points identified by O2.

Threads	**Event handlers**
java.lang.Thread.start()	actionPerformed(...)
java.lang.Runnable.run()	onMessageEvent(...)
java.util.concurrent.Callable.call()	handleEvent(...)
pthread_create(...)	onReceive(...)

* More details for Android events are in Section 4.2.

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 even-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 2. The OPA rules for Java. Consider the following statements are in method `m()` with Origin $\mathbb{O}_i$, denoted $\langle m, \mathbb{O}_i \rangle$. The edges $\longrightarrow$ are in the PAG and $\rightsquigarrow$ in the call graph.

Statement	Points-to Edge & Call Edge
❶ $x = new\ C()$	$\langle o, \mathbb{O}_i \rangle \longrightarrow \langle x, \mathbb{O}_i \rangle$
❷ $x = y$	$\langle y, \mathbb{O}_i \rangle \longrightarrow \langle x, \mathbb{O}_i \rangle$
❸ $x.f = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \longrightarrow \langle o, \mathbb{O}_k \rangle.f}$
❹ $x = y.f$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle.f \longrightarrow \langle x, \mathbb{O}_i \rangle}$
❺ $x[idx] = y$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle x, \mathbb{O}_i \rangle)}{\langle y, \mathbb{O}_i \rangle \longrightarrow \langle o, \mathbb{O}_k \rangle.*}$
❻ $x = y[idx]$	$\dfrac{\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle)}{\langle o, \mathbb{O}_k \rangle.* \longrightarrow \langle x, \mathbb{O}_i \rangle}$
❼ $x = y.f(a_1, ..., a_n)$ //non-origin entry	$\dfrac{\begin{array}{c}\forall \langle o, \mathbb{O}_k \rangle \in pts(\langle y, \mathbb{O}_i \rangle) \\ \langle f', \mathbb{O}_i \rangle = dispatch(\langle o, \mathbb{O}_k \rangle, f)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_k \rangle \longrightarrow \langle f'_{this}, \mathbb{O}_i \rangle \\ \langle a_h, \mathbb{O}_i \rangle \longrightarrow \langle p_h, \mathbb{O}_i \rangle, \text{ where } 1 \le h \le n \\ \langle f'_{ret}, \mathbb{O}_i \rangle \longrightarrow \langle x, \mathbb{O}_i \rangle \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle f', \mathbb{O}_i \rangle\end{array}}$
❽ $x = new\ O(b_1, ..., b_n)$ //origin allocation	$\dfrac{\begin{array}{c}\textbf{\textit{Compute new origin}}\text{: } \mathbb{O}_j \\ \langle init, \mathbb{O}_j \rangle = dispatch(-, init)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_j \rangle \longrightarrow \langle init_{this}, \mathbb{O}_j \rangle \\ \langle o, \mathbb{O}_j \rangle \longrightarrow \langle x, \mathbb{O}_i \rangle \\ \langle b_h, \mathbb{O}_i \rangle \longrightarrow \langle p_h, \mathbb{O}_j \rangle, \text{ where } 1 \le h \le n \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle init, \mathbb{O}_j \rangle\end{array}}$
❾ $x.entry(c_1, ..., c_n)$ //origin entry point	$\dfrac{\begin{array}{c}\forall \langle o, \mathbb{O}_j \rangle \in pts(\langle x, \mathbb{O}_i \rangle) \\ \langle entry', \mathbb{O}_j \rangle = dispatch(\langle o, \mathbb{O}_j \rangle, entry)\end{array}}{\begin{array}{c}\langle o, \mathbb{O}_j \rangle \longrightarrow \langle entry'_{this}, \mathbb{O}_j \rangle \\ \langle c_h, \mathbb{O}_i \rangle \longrightarrow \langle p_h, \mathbb{O}_j \rangle, \text{ where } 1 \le h \le n \\ \textbf{\textit{add call edge}}\ \langle m, \mathbb{O}_i \rangle \rightsquigarrow \langle entry', \mathbb{O}_j \rangle\end{array}}$

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

```
public static void main(){
  //Tmain
  TA a = new TA();//oa→Ta
  TB b = new TB();//ob→Tb
  a.start(); b.start();
}
Class TA extends T {
  TA() { super(); ... }
}
Class TB extends T {
  TB() { super(); ... }
}
Class T {
  Object f;
  T() { f = new Object();
 //without switch: ⟨of,Tmain⟩
 //with context switch:
 //⟨of,Ta⟩ and ⟨of,Tb⟩
  }
  public void run(){
    f.do_something();
  }
}
```

Figure 3. An example to explain why it is necessary to switch context at origin allocations.

Intra-Origin Constraints. Statements ❶-❼ are in method $\langle m, \mathbb{O}_i \rangle$, and all the program elements created by them share the same origin $\mathbb{O}_i$ to indicate where they are originated from. For example, the allocated object by statement ❶ is represented as $\langle o, \mathbb{O}_i \rangle$ and assigned to pointer $\langle x, \mathbb{O}_i \rangle$, and their relation is represented by a points-to edge $\langle o, \mathbb{O}_i \rangle \longrightarrow \langle x, \mathbb{O}_i \rangle$ in the PAG .

An object field pointer is distinguished by the origin of its receiver object. For statement ❹, each receiver object $\langle o, \mathbb{O}_k \rangle$ corresponds to an object field pointer $\langle o, \mathbb{O}_k \rangle.f$ that points to $\langle x, \mathbb{O}_i \rangle$. 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 ❺❻ 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 ❼ invokes a target method `f'` within the same origin $\mathbb{O}_i$ as its caller, even though its receiver object $\langle o, \mathbb{O}_k \rangle$ might be allocated from a different origin $\mathbb{O}_k$. To determine a virtual call target and its context (*e.g.,* the call on line 13 in Figure 2), we use the type of its receiver object o and the origin $\mathbb{O}_i$ 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 $\mathbb{O}_i$ to a new origin $\mathbb{O}_j$ for an origin allocation ❽ and an origin entry point ❾.

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

Table 3. The time complexity of different pointer analyses.

Analysis	Worst-Case Complexity
0-context	$O(p \times h^2)$
heap	$O(p^3 \times h^2)$
2-CFA + heap	$O(p^5 \times h^2)$
2-obj + heap	$O(p^5 \times h^2)$
1-origin + heap	$O(p^3 \times h^2)$

`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 $pts(o_a.f) = pts(o_b.f) = \{\langle o_f, Tmain\rangle\}$, 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 ❽ 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 $\mathbb{O}_j$ is created for this new allocation $\langle o, \mathbb{O}_j\rangle$.

Both ❽ and ❾ designate the attributes for the new origin $\mathbb{O}_j$, 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 ownership, the actual parameters use $\mathbb{O}_i$ as their contexts and the formal ones use $\mathbb{O}_j$. Meanwhile, call edges are added in the call graph, *e.g.*, $\langle m, \mathbb{O}_i\rangle \rightsquigarrow \langle init, \mathbb{O}_j\rangle$ for ❽ and $\langle m, \mathbb{O}_i\rangle \rightsquigarrow \langle entry', \mathbb{O}_j\rangle$ for ❾.

Wrapper Functions and Loops. In practice, both ❽ and ❾ 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:

$$\langle m, [\mathbb{O}_1, \mathbb{O}_2, ..., \mathbb{O}_{k-1}, \mathbb{O}_k]\rangle$$

where `m()` is invoked within Origin $\mathbb{O}_k$ that has a parent origin $\mathbb{O}_{k-1}$, 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.

Algorithm 1 Origin-Sharing Analysis

```
Global States:
OPA - origin-sensitive pointer analysis.
$visitedMethods \leftarrow \emptyset$;                 ▷ flag visitedMethods
$m \leftarrow main$;                                    ▷ the main method
VisitMethod($m$);

VisitMethod($m$):
$visitedMethods.add(m)$;
for $s \in m.statements$ do
  switch ($s$)
    case $x.f$:                          ▷ read/write object field
      $origins \leftarrow$ FindPointsToOrigins($p$);
      for $\mathbb{O} \in origins$ do
        ComputeOriginSharing($s$,$f$,$\mathbb{O}$,$read/write$);
      break;
    case $x[idx]$:                             ▷ read/write array
      $origins \leftarrow$ FindPointsToOrigins($a$);
      for $\mathbb{O} \in origins$ do
        ComputeOriginSharing($s$,$*$,$\mathbb{O}$,$read/write$);
      break;
    case $m(args)$:                           ▷ call a new method
      for $m' \in$ FindCalleeMethods($m(args)$) do
        if $!visitedMethods.contains(m')$ then
          VisitMethod($m'$);
      break;
    default: break;
  end switch

ComputeOriginSharing($s$, $f$, $\mathbb{O}$, $isWrite$):
Input: $s$ - the statement;
       $f$ - accessed field ($*$ means array access);
       $\mathbb{O}$ - origin;
       $isWrite$ - true means $write$ and false means $read$.
$WO \leftarrow$ GetWriteOrigins($s$,$f$);
$RO \leftarrow$ GetReadOrigins($s$,$f$);
if $isWrite$ && $!WO.contains(\mathbb{O})$ then
  $WO.add(\mathbb{O})$;
if $!isWrite$ && $!RO.contains(\mathbb{O})$ then
  $RO.add(\mathbb{O})$;
```

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 $(\#\mathbb{O} \times \mathbb{O}\%)$, where $\#\mathbb{O}$ is the number of origins and $\mathbb{O}\%$ 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 ❸❹ 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 $\mathbb{O}$, we call the procedure **ComputeOriginSharing**(s, f, $\mathbb{O}$, *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 **GetReadOrigins** and **GetWriteOrigins**, 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 ❺❻ 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 ❼❽❾ in Table 2) (the receiver object is also included in the arguments `args`), we use OPA again to determine the possible callee 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 reads and writes to provide fine-grained access information. This is particularly useful for static race detection and performance optimizations.

Table 4. SHB Graph with Origins: the following statements are in method `m()` with Origin $\mathbb{O}_i$.

Intra-Origin Happen-before Rules	
Statement	**Intra-Origin Node & HB Edge**
❸ $x.f = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, $\mathbf{write}(\langle o, \mathbb{O}_k\rangle.f)$
❹ $x = y.f$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, $\mathbf{read}(\langle o, \mathbb{O}_k\rangle.f)$
❺ $x[idx] = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, $\mathbf{write}(\langle o, \mathbb{O}_k\rangle.*)$
❻ $x = y[idx]$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, $\mathbf{read}(\langle o, \mathbb{O}_k\rangle.*)$
❼ $x = y.f(a_1, ..., a_n)$	$\forall\langle f, \mathbb{O}_i\rangle \in dispatch(\langle y, \mathbb{O}_i\rangle, f)$, *add HB edge*: $\mathbf{call}(\langle f, \mathbb{O}_i\rangle) \Rightarrow \mathbf{f_{first}}(\langle f, \mathbb{O}_i\rangle)$, $\mathbf{f_{last}}(\langle f, \mathbb{O}_i\rangle) \Rightarrow \mathbf{call_{next}}(\langle f, \mathbb{O}_i\rangle)$
🔓 $synchronized(x)\{ \ldots \}$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, $\mathbf{lock}(\langle o, \mathbb{O}_k\rangle)$, $\mathbf{unlock}(\langle o, \mathbb{O}_k\rangle)$

Inter-Origin Happen-before Rules	
Statement	**Inter-Origin Node & HB Edge**
❾ $x.entry(c_1, ..., c_n)$	$\forall\langle entry, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, entry)$, *add HB edge*: $\mathbf{entry}(\mathbb{O}_i, \mathbb{O}_j) \Rightarrow \mathbf{origin_{first}}(\mathbb{O}_j)$
❿ $x.join()$	$\forall\langle join, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, join)$, *add HB edge*: $\mathbf{origin_{last}}(\mathbb{O}_j) \Rightarrow \mathbf{join}(\mathbb{O}_j, \mathbb{O}_i)$

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 [45] (although ours is static). More specifically, our detection represents happens-before relations by a static happens-before (SHB) graph [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 ❸-❻ by creating read and write nodes. For statement ❼, we create a method call node (call) with two happens-before (HB) edges (denoted ⇒): one points from the call node to the first node ($\mathrm{f_{first}}$) of its target method `f` within the same origin $\mathbb{O}_i$, the other points from the last node ($\mathrm{f_{last}}$) of $\langle f, \mathbb{O}_i\rangle$ to the next node after the call ($\mathrm{call_{next}}$). Intra-origin HB edges are created by pointing from one intra-origin node to another in their statement order.

For lock operation 🔓, 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 atomics (*e.g.*, `std::atomic`) and semaphores in our future work, by adding new happens-before rules from different origins to the atomic/semaphore operations.

For calls to an origin entry point ❾, we create an origin entry node (entry) to represent the start of a new origin $\mathbb{O}_j$ from its parent origin $\mathbb{O}_i$. And we add an inter-origin HB edge pointing to the first node (origin$_{\text{first}}$) of $\mathbb{O}_j$. For thread join statement ❿, we create a join node (join) to indicate the end of $\mathbb{O}_j$ that finally joins to $\mathbb{O}_i$. An inter-origin HB edge is created from the last node (origin$_{\text{last}}$) of current origin ($\mathbb{O}_j$) 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_s), which incurs redundant race checking. Instead, we treat all the memory accesses on o_s within the same lock region as a single memory access on o_s, 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.

	Pointer Analysis							Race Detection						
App	**0-ctx**	**#O**	**OPA**	**1-CFA**	**2-CFA**	**1-obj**	**2-obj**	**0-ctx**	**O2**	**1-CFA**	**2-CFA**	**1-obj**	**2-obj**	**RacerD**
Avrora	13.42	*4*	15.56	17.81	56.06	50.30	>4h	20.70	17.85/**-14%**	30.63/0.48x	615.42/29x	1064/50x	-	18.36
Batik	7.22	*4*	9.83	49.28	2606	>4h	>4h	14.47	14.93/3%	84.01/4.81x	2648/182x	-	-	1min12s
Eclipse	5.03	*4*	6.52	8.43	8.71	11.84	>4h	7.21	8.03/11%	20.35/1.82x	40.36/4.60x	641.38/88x	-	*
H2	49.95	*3*	111.37	192.71	1397	>4h	>4h	58.13	169.63/192%	263.00/3.52x	3208/54x	-	-	38.25
Jython	25.77	*4*	66.34	16.09	58.85	>4h	>4h	163.49	537.63/229%	100.35/-0.39x	172.46/0.05x	-	-	1min47s
Luindex	10.23	*3*	15.73	16.74	26.38	>4h	>4h	14.43	20.19/40%	31.62/1.19x	1634/112x	-	-	2min39s
Lusearch	5.06	*3*	5.66	31.60	2384	6.48	6.63	7.09	7.99/13%	33.79/3.77x	2401/338x	8.58/0.21x	15.05/1.12x	2min39s
Pmd	5.50	*3*	5.75	6.04	8.00	>4h	>4h	25.07	13.32/**-47%**	57.21/1.28x	122.93/3.90x	-	-	2min15s
Sunflow	3.13	*9*	5.68	5.08	5.44	5.12	>4h	20.67	26.01/26%	297.05/13x	2408/116x	3007/144x	-	14.77
Tomcat	3.77	*6*	10.18	30.47	2312	5.89	676.59	7.58	16.87/123%	66.48/7.77x	2829/372x	2918/384x	9589/1.3kx	1min31s
Tradebeans	4.96	*3*	6.05	6.76	7.54	>4h	>4h	8.19	12.05/47%	16.49/1.01x	111.80/13x	-	-	9.38
Tradesoap	6.25	*3*	7.44	8.33	9.31	>4h	>4h	10.22	10.77/5%	24.32/1.38x	149.53/14x	-	-	9.38
Xalan	31.30	*3*	35.73	65.51	3922	213.99	>4h	34.71	42.87/24%	79.33/1.29x	5722/164x	3h/305x	-	32.87
ConnectBot	2.40	*11*	5.45	23.85	3513	>4h	>4h	2.49	5.57 /124%	23.99/8.63x	3513/1.4kx	-	-	*
Sipdroid	5.80	*15*	31.48	14.33	3436	>4h	>4h	16.02	228.33/1.3k%	40.88/1.55x	3452/215x	-	-	*
K-9 Mail	6.56	*23*	14.73	30.88	4284	>4h	>4h	8.59	19.49/127%	33.32/2.88x	4288/498x	-	-	4min56s
Tasks	6.90	*7*	12.72	117.63	8081	>4h	>4h	7.10	12.90/82%	117.77/15.59x	8081/1.1kx	-	-	*
FBReader	6.66	*15*	20.16	45.26	2.97h	>4h	>4h	7.49	23.33 /211%	52.79/6.05x	3.10h/1.5kx	-	-	*
VLC	5.35	*4*	46.40	25.40	3235	>4h	>4h	5.39	46.44/762%	25.44/3.72x	3235/599x	-	-	*
FireFox Focus	3.84	*8*	15.46	17.96	>4h	>4h	>4h	4.08	15.76/286%	18.34/3.50x	-	-	-	2min5s
Telegram	20.82	*134*	199.79	83.31	>4h	>4h	>4h	41.76	372.93/793%	171.42/3.10x	-	-	-	*
Zoom	36.77	*15*	148.01	198.59	>4h	>4h	>4h	37.62	149.01/296%	200.47/4.33x	-	-	-	*
Chrome	6.14	*34*	108.76	18.43	>4h	>4h	>4h	7.35	111.79/1.4k%	22.72/2.09x	-	-	-	*
HBase	41.96	*16*	494.64	61.75	>4h	>4h	>4h	>4h	1.34h/**-66.5%**	>4h	-	-	-	8min12s
HDFS	29.03	*12*	102.83	40.35	165.05	>4h	>4h	>4h	499.53/**-28x**	>4h	>4h	-	-	3min22s
Yarn	416.37	*14*	603.70	61.42	55.58	>4h	>4h	>4h	1.7h/**-57.5%**	>4h	>4h	-	-	8min5s
ZooKeeper	14.40	*40*	33.45	15.32	33.31	>4h	>4h	>4h	271.20/**-53x**	>4h	>4h	-	-	21.18

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

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

App	**#KLOC**	**Metrics**	**0-ctx**	**O2**	**2-CFA**
Memcached (#O = 12)	20.4	Time/SD	5.3	5.8/9%	7.5/41%
		#Pointer	8,400	12,883	15,772
		#Object	2,420	2,468	2,765
		#Edge	5,395	10,415	17,116
Redis (#O = 15)	116	Time/SD	9.3	15.0/61%	275.9/28x
		#Pointer	44,535	54,690	281,524
		#Object	14,458	14,913	32,401
		#Edge	598,981	963,654	13,530,084
Sqlite3 (#O = 3)	245	Time/SD	213	273/28%	OOM
		#Pointer	57,657	61,796	-
		#Object	10,093	10,310	-
		#Edge	7,909,626	8,879,155	-
Avg.	126	Time/SD	75.8	97.9/30%	-

Table 7. Performance and #Shared memory accesses (#S-access) of OSA.

App	#S-access	Time
Avrora	16	16.72s
Batik	293	7.79s
Eclipse	343	9.22s
H2	2,207	2.3min
Jython	13,121	4.5min
Luindex	2,001	1.6min
Lusearch	252	7.01s
Pmd	300	8.57s
Sunflow	1,603	10.15s
Tomcat	700	15.39s
Tradebeans	45	7.43s
Tradesoap	37	9.12s
Xalan	14	1.2min

Time includes the time of OPA.

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.

App	0-ctx	O2	1-CFA	2-CFA	1-obj	2-obj	O2	RacerD
Avrora	12,633	38/99.7%	45/99.6%	45/99.6%	47/99.6%	-	38	117
Batik	4,369	186/95.7%	4,229/3.2%	640/85.4%	-	-	186	1,562
Eclipse	958	7/99.3%	944/1.5%	822/14.2%	945/1.4%	-	7	*
H2	9,698	2,817/71.0%	7,832/19.2%	6,322/34.8%	-	-	2,817	6,743
Jython	7,997	3,651/54.3%	2,402/70.0%	2,358/70.5%	-	-	3,651	52,872
Luindex	3,218	1,792/44.3%	2,821/12.3%	2,271/29.4%	-	-	1,792	172
Lusearch	567	341/39.9%	538/5.1%	494/12.9%	529/6.7%	526/7.2%	341	172
Pmd	307	256/16.6%	296/3.6%	293/4.6%	-	-	256	1
Sunflow	9,238	1,925/79.2%	6,868/25.7%	5,899/36.1%	2,288/75.2%	-	1,925	69
Tomcat	751	307/59.1%	701/6.7%	693/7.7%	585/22.1%	575/23.4%	307	3,257
Tradebeans	193	75/61.1%	171/11.4%	168/13.0%	-	-	75	90
Tradesoap	264	64/75.8%	179/32.2%	177/33.0%	-	-	64	90
Xalan	6	1/83.3%	6/0.0%	6/0.0%	6/0.0%	-	1	754

"-": Time out.

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 upto 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 these 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 legency 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*.

Table 9. #Races and #Thread-shared objects (#S-obj) from different pointer analyses for distributed systems.

	#Race		#S-obj			
App	**O2**	**RacerD**	**0-ctx**	**1-CFA**	**2-CFA**	**O2**
HBase	687	727	1,269	1,799	-	903
HDFS	910	884	2,322	3,139	6,605	1,066
Yarn	1,164	1,246	5,387	3,083	2,146	1,162
ZooKeeper	747	407	1,389	2,511	4,299	1,271

"-": Time out.

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, TDengine, 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.

5.3 Trade-Off between Precision and Performance

Our results show that O2 significantly outperforms k-CFA and k-obj ($k \leq 2$) in terms of both performance and precision (Table 5). The reason for the improved precision is that the use of origins as context significantly improves the analysis precision on the thread- and event-local objects that are created within an origin. Such origin-local objects would be falsely analyzed as shared by k-CFA or k-obj if k is smaller than the depth of the call chain inside the thread or event, whereas such objects can be correctly analyzed as thread-local by O2.

The performance of OPA has obvious slowdown on the distributed systems: the max slowdown is 9.86x on *Yarn* compared with 2-CFA. However, its corresponding total time of race detection is at least 57% faster (up to 53x on *ZooKeeper*). The reason behind this significant speedup is the largely reduced number of thread-shared objects as shown in Table 9, which means less workload in both checking happens-before relations and computing common locksets.

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 tinyconfig64, 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).

```
void update_vsyscall_tz(void){//in class time.vsyscall
  struct vdso_data *vdata = __arch_get_k_vdso_data();
  vdata[CS_HRES_COARSE].tz_minuteswest = sys_tz.
       tz_minuteswest; //RACE
  vdata[CS_HRES_COARSE].tz_dsttime = sys_tz.tz_dsttime;
        //RACE
  ... }
```

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

	Linux	**TDengine**	**Redis/RedisGraph**	**OVS**	**cpqueue**	**mrlock**	**Memcached**	**Firefox**	**ZooKeeper**	**HBase**	**Tomcat**
#Races	6	6	5	3	7	5	3	2	1	1	1

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:

```
void *do_slabs_reassign(){ ... //event
  if (slabsclass[id].slabs > 1){
    return cur;//RACE: missing lock
}}
void *do_slabs_newslabs(){ ... //thread
  pthread_lock();
  p->slab_list[p->slabs++] = ptr;//with lock
  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:

```
// called from Gecko background thread
public synchronized IChildProcess bind(){ ...
 Context ctx = GeckoAppShell.getAppCtx();//RACE
  ... }
// called from MainActivity.onCreate()
@UiThread
public void attachTo(Context context){ ...
 Context appCtx = context.getAppCtx();
 if(!appCtx.equals(GeckoAppShell.getAppCtx())){
   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 between 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:

```
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   HashSet<String> list = ephemerals.get(eowner);
11   if (list == null){ list = new HashSet<String>();
12     ephemerals.put(eowner, 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/gpiolib.c@1104:8

[2] /linux-stable/kernel/irq/manage.c@1279:7 and @1282:7

two code snippets (line 4-7 and line 10-13) are not protected by common locks or mechanism from *ConcurrentHashMap*. Hence, the null checks from two threads on variable *list* may return null, but only one initialized set can be stored in *ephemerals* and all the paths added by another thread are missing. The race in *HBase* has the same reason as above, involving two concurrent accesses on a map, *keyProviderCache*, without locks from method `getKeyProvider()` (in class `org.apache.hadoop.hbase.io.crypto.Encryption`).

6 Conclusion and Future Work

We have presented O2, a new system for static race detection. O2 is powered by a novel abstraction, *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 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. O2 has been integrated into Coderrect, a commerical static analyzer [3]. In future work, we plan to implement and evaluate 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

[1] 2019. Sable/soot: Soot - A Java optimization framework. https://github.com/Sable/soot.

[2] 2020. Processes and threads overview | Android Developers. https://developer.android.com/guide/components/processes-and-threads.

[3] 2021. Coderrect race detector. https://coderrect.com/download/.

[4] 2021. Infer : RacerD. http://fbinfer.com/docs/racerd.html.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 289–302. http://dl.acm.org/citation.cfm?id=647057.713851

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[8] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514

[9] Byeong-Mo Chang and Jong-Deok Choi. 2004. Thread-Sensitive Points-to Analysis for Multithreaded Java Programs. In *Computer and Information Sciences - ISCIS 2004*, Cevdet Aykanat, Tuğrul Dayar, and İbrahim Körpeoğlu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 945–954.

[10] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*.

[11] CIL. 2016. CIL - Infrastructure for C Program Analysis and Transformation. https://people.eecs.berkeley.edu/~necula/cil/.

[12] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[13] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[14] Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAdroid: Statically Detecting Ordering Violations in Android Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 62–74. https://doi.org/10.1145/3168829

[15] Dirk Grunwald and Harini Srinivasan. 1993. Data Flow Equations for Explicitly Parallel Programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPOPP '93)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/155332.155349

[16] Samuel Z. Guyer and Calvin Lin. 2003. Client-driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 214–236. http://dl.acm.org/citation.cfm?id=1760267.1760284

[17] Samuel Z. Guyer and Calvin Lin. 2005. Error Checking with Client-driven Pointer Analysis. *Sci. Comput. Program.* 58, 1-2 (Oct. 2005), 83–114. https://doi.org/10.1016/j.scico.2005.02.005

[18] Philipp Haller and Martin Odersky. 2007. Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages* (Paphos, Cyprus) *(COORDINATION'07)*. Springer-Verlag, Berlin, Heidelberg, 171–190. http://dl.acm.org/citation.cfm?id=1764606.1764620

[19] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. 2007. Component-Based Lock Allocation. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 353–364. https://doi.org/10.1109/PACT.2007.4336225

[20] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. 2017. AsyncClock: Scalable Inference of Asynchronous Event Causality. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi;an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/3037697.3037712

[21] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 326–336. https://doi.org/10.1145/2594291.2594330

[22] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-Based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 257–270. https://doi.org/10.1145/3173162.3173173

[23] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction.. In *PLDI*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 337–348. http://dblp.uni-trier.de/db/conf/pldi/pldi2014.html#HuangMR14

[24] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

[25] Ranjit Jhala and Rupak Majumdar. 2007. Interprocedural Analysis of Asynchronous Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/1190216.1190266

[26] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/1595696.1595701

[27] Timo Kamph. 2012. *An interprocedural Points - To Analysis for Event-Driven Programs.* Diplomarbeit. TU Hamburg-Harburg.

[28] Hugh C. Lauer and Roger M. Needham. 1979. On the Duality of Operating System Structures. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 3–19. https://doi.org/10.1145/850657.850658

[29] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. https://doi.org/10.1109/MC.2006.180

[30] Ondřej Lhoták. 2002. *Spark: A flexible points-to analysis framework for Java.* Master's thesis. McGill University.

[31] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[32] Peng Li and S Zdancewic. 2006. A Language-based Approach to Unifying Events and Threads. (2006).

[33] Yue Li, Tian Tan, Anders Møler, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[34] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 141.

[35] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3192366.3192390

[36] llvm. 2018. llvm. http://llvm.sourceforge.net/wiki/index.php/Main_Page.

[37] Alexey Loginov, Vivek Sarkar, Jong deok Choi, Jong deok Choi, Alexey Logthor, and I Vivek Sarkar. 2001. *Static Datarace Analysis for Multithreaded Object-Oriented Programs.* Technical Report. IBM Research Division, Thomas J. Watson Research Centre.

[38] Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

[39] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-Driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 505–519. https://doi.org/10.1145/2814270.2814272

[40] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 316–325. https://doi.org/10.1145/2594291.2594311

[41] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[42] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

[43] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[44] John OUSTERHOUT. 1995. Why Threads are a Bad Idea (for most purposes). (1995).

[45] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/781498.781528

[46] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[47] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 320–331. https://doi.org/10.1145/1133981.1134019

[48] J. Qian and B. Xu. 2007. Thread-Sensitive Pointer Analysis for Inter-Thread Dataflow Detection. In *11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'07)*. 157–163. https://doi.org/10.1109/FTDCS.2007.34

[49] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[50] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. 2001. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 43–55. https://doi.org/10.1145/504282.504286

[51] Erik Ruf. 2000. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 208–218. https://doi.org/10.1145/349299.349327

[52] Radu Rugina and Martin C. Rinard. 2003. Pointer Analysis for Structured Parallel Programs. *ACM Trans. Program. Lang. Syst.* 25, 1 (Jan. 2003), 70–116. https://doi.org/10.1145/596980.596982

[53] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-Based Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/2786805.2786836

[54] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1998. Solving Shape-analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20, 1 (Jan. 1998), 1–50. https://doi.org/10.1145/271510.271517

[55] Alexandru Salcianu and Martin Rinard. 2001. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (Snowbird, Utah, USA) *(PPoPP '01)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/379539.379553

[56] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[57] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences, Chapter 8, 189–233.

[58] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Technical Report.

[59] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[60] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[61] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 485–495.

[62] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *The 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 27–37.

[63] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse Flow-Sensitive Pointer Analysis for Multithreaded Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 160–170. https://doi.org/10.1145/2854038.2854043

[64] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

[65] K. Tuncay Tekle and Yanhong A. Liu. 2016. Precise Complexity Guarantees for Pointer Analysis via Datalog with Extensions. *CoRR* abs/1608.01594 (2016). arXiv:1608.01594 http://arxiv.org/abs/1608.01594

[66] Rob von Behren, Jeremy Condit, and Eric Brewer. 2003. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Lihue, Hawaii) *(HOTOS'03)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1251054.1251058

[67] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 268–281. https://doi.org/10.1145/945445.945471

[68] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[69] WALA. 2018. Wala. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[70] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Trans. Priv. Secur.* 21, 3, Article 14 (April 2018), 32 pages. https://doi.org/10.1145/3183575

[71] Ming-Ho Yee, Ayaz Badouraly, Ondřej Lhoták, Frank Tip, and Jan Vitek. 2019. Precise Dataflow Analysis of Event-Driven Applications. *arXiv preprint arXiv:1910.12935* (2019).

[72] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1095810.1095832

[73] Sheng Zhan and Jeff Huang. 2016. ECHO: Instantaneous In Situ Race Detection in the IDE. In *Proceedings of the ? International Symposium on the Foundations of Software Engineering* (Seattle, WA, USA) *(FSE '16)*.