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

Anonymous Author(s)

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 software. We present a novel static 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 through static analysis, but can be also specified by developers. More importantly, origins provide an efficient way to precisely reason about shared memory and pointer aliases.

We have implemented O2 for both C/C++ and JVM 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 have evaluated with, including Linux kernel, Redis, OVS, Memcached, Hadoop, Tomcat, ZooKeeper and Firefox Android apps. Moreover, O2 scales to millions of lines of code in a few minutes, on average 70x faster (up to 568x) compared to existing static analysis tools, and reduces 77% false positives. Moreover, O2 outperforms the state-of-the-art static race detection tool, RacerD, by the similar performance and 6x less reported false positives on average. At the time of writing, O2 has revealed more than 40 unique previously unknown races that have been confirmed or fixed by developers.

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 [45, 70], 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 applications. However, most successful techniques [11, 21, 32, 43, 47, 78] are limited to test runs or small code bases. The classical dynamic race detector, ThreadSanitizer [57], adopts the epoch-based happens-before relation [27] to be scalable. More recently, TSVD [32] combines active testing and learning to detect races. Unfortunately, dynamic techniques face an inherence challenge of performance overhead and low code coverage. In contrast, static detection techniques have had only very limited success despite decades of research [1, 10, 38, 44, 48, 72]. A crucial reason is that reasoning about races typically requires sophisticated pointer alias analysis to attain accuracy, which is difficult to scale.

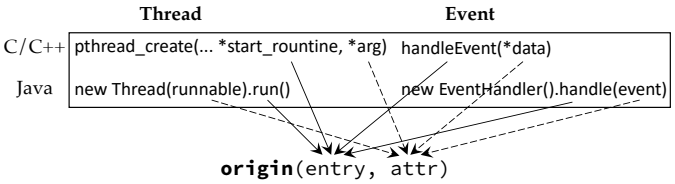


Figure 1. An "origin" view of threads and events.

Races in event-driven programs have attracted much attention recently [12, 18–20, 24, 41, 50, 54]. Event-based races are even more challenging to detect than thread-based races because most events are asynchronous and the event handlers can be triggered in many different ways. Moreover, the difficulty in detecting event-based races can be 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 [1, 32, 57] do not perform well in detecting event-based races, due to the large space of casual orders among event handlers and threads, as well as the large number of events and event sequences.

1.1 Key Insights

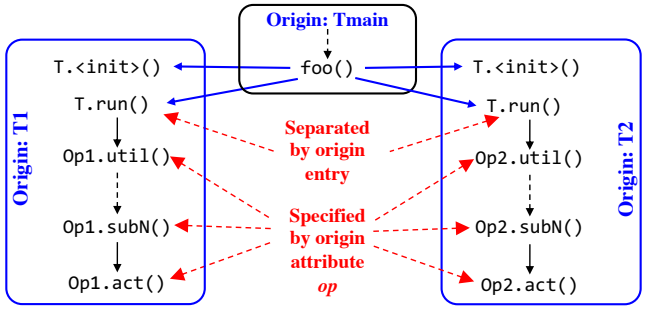
In this paper, we present O2, an effective technique for detecting data races in complex software involving both threads and events. The key insight behind O2 is a novel abstraction called *origins*, which unify threads and events as the context to achieve significant improvements in the scalability and precision of static analysis. As depicted in Figure 1, there exists a unified view through origins for both threads and events in C/C++ and Java, in which the entry point corresponds to the thread start or event handler functions, and the attribute corresponds to the data passed to the thread or event handler. The data can be represented by a pointer to memory objects that are expected to be used in the thread or event handler.

At its core, each origin has two parts: an entry point that represents the beginning of a thread or an event handler, and a set of data pointers that capture important attributes associated with the entry point and additional semantics, such as thread ID, event type, or shared memory objects. By

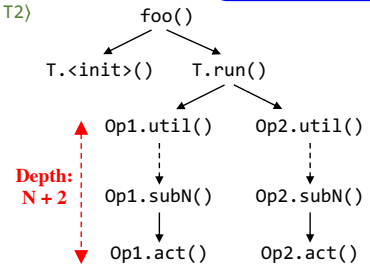
PL'21, January 01–03, 2018, New York, NY, USA
2018.

```
1  public void foo(){//main thread
2    Obj s = new Obj();//o1
3    Op op1 = new Op1();//o2
4    Op op2 = new Op2();//o3
5    new T(s, op1).run();//o4 → Origin: T1
6    new T(s, op2).run();//o5 → Origin: T2
7  }
8  public class T extends Thread{
9    Obj f; Op op;//super class of Op1 and Op2
10   public T(Obj a, Op b){
11     f = a; op = b; }
12   public void run(){
13     op.util(f, new Obj());//o6
14   } //with origin:⟨o6,T1⟩ and ⟨o6,T2⟩
15 }
16 void util(Obj x, Obj y){
17   sub1(x, y); ...
18 }
19 void sub1(Obj x, Obj y){
20   sub2(x, y); ...
21 }              ...
22 void subN(Obj x, Obj y){
23   y.do_something();
24   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. An example illustrating how origin separates the program into different parts. (a) The example code. (b) The origin-sensitive call graph, where each origin is consisted of a sequence of calls with an 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 OSA output.

automatically deriving the code paths starting from the entry points and the program states reachable from each origin, O2 leads to a powerful static race detector for multithreaded and event-driven programs.

To elucidate the advantage of origins, 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 [60]. 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 [42] (denoted *k-obj*), in which the sequence of methods `subN()` are invoked on different receiver objects.

Alternatively, it suffices to mark the method `run()` in each thread as a different *origin*. In this way, the allocation on line 13 can be distinguished by an origin unique to its thread. Similarly, the virtual method `act()` invoked by each thread on line 24 can be distinguished by the origin attributes: *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.

To conclude, origin-sensitive analysis is more precise than the conventional context-sensitive analyses for reasoning about threads and events. More importantly, because each origin can capture an arbitrarily long call chain, origin-sensitive analysis avoids context explosion and gains scalability compared to the classical cloning-based analysis methods [28, 75].

1.2 Technical Contributions

A central technical advance this work is the *origin-sensitive pointer analysis* (**OPA**), in which the conventional call-string-based or object-based context abstractions are replaced by origins. Although context-sensitive pointer analysis has been researched intensively, it has not been widely adopted due to its poor scalability in practice. The problem is that the number of contexts grows exponentially with the size of the call chain and the object access path [75]. Origin-sensitivity is an effective solution to this problem because:

- 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 in the origin;
- The number of origins is much smaller than the number of calling contexts of a function; and
- Computing *k*-most-recent calling contexts at every call site is redundant in many applications [67], e.g., when determining which objects are local to or are shared by which threads.

Table 1 summarizes the worst-case time complexity of different pointer analysis algorithms according to [69], 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 [55]. The selective context-sensitive techniques [14, 15, 34, 35, 39, 63] are also bounded

by the context depth, which have the same worst-case complexity as their corresponding full k-CFA and k-obj algorithms.

Table 1. The time complexity for 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	$O(p^3 \times h^2)$

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.

Based on OPA, we further develop a new type of analysis for multithreaded and event-driven programs, called *origin-sharing analysis* (**OSA**). OSA is comparable to thread-escape analysis, but has several advantages (more details are in Section 3.3). In addition to answering *whether* an object is shared, OSA provides more information on *how* the object is shared across the origins. More specifically, OSA automatically identifies the list of objects local to each origin, and the list of objects shared by each combination of multiple origins. Figure 2(d) shows a sample OSA output for the code in Figure 2(a).

1.3 Results

We implemented O2 together with the origin-sensitive analyses for both C/C++ and JVM applications based on LLVM [37] and WALA [73]. The race detection engine uses a lock-region-based algorithm powered by fast checking of happens-before relations and a compact representation of locksets, which achieves both efficiency and scalablility. We have applied O2 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 77% false positives compared to existing static analyses. Besides, we compare O2 with RacerD [1], a state-of-the-art static data race detector. On many of our evaluated programs, RacerD cannot finish the analysis due to exceptions. For the other programs, O2 achieves comparable performance to RacerD while detecting many new races and 6x less false positives on average. In most of the evaluated programs in which O2 detects new races, RacerD either fails to find any races or cannot finish the detection without exceptions.

Surprisingly, O2 *found real and previously unknown races in every single real-world code base we evaluted 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.

The rest of this paper is organized as follows: we discuss more related work in Section 2; we present origin-sensitive analyses in Sections 3, and our race detection algorithms in Section 4; we present the evaluation results including several real-world case studies Section 5, and conclude in Section 6.

2 Related Work

Thread vs. Event. In their seminal work [29] 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 [45] argued against using threads due to the difficulty of developing correct threaded code. Lee [30] 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. [70] remarked on the "stack ripping" problem of events, and advocated for using threads for their simple and powerful abstraction. In Capriccio [71], 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. [4] 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 program'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 [16] and Haskell [33], 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.

Table 2. 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 5.1.

Static Race Detection. By far, RacerD developed at Facebook is probably the most successful static race detector [1]. It is regularly applied to Android apps in Facebook and has flagged over 2500 issues that have been fixed by developers before reaching production. 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. Besides, RacerD works for Java and is on its way to be compatible with C/C++, whereas O2 can deal with low-level pointers in C/C++ such as indirect function targets and virtual tables.

Other classic static race detection tools (*e.g.,* RacerX [10], RELAY [72]) have their own difficulties to be applied to modern software. RacerX is not open source and contains many heuristics and engineering decisions, which is difficult to duplicate. RELAY depends on an out-dated compiler frontend (CIL [9]), which no longer supports modern major compilers. Technically, RELAY uses a context- and field-insensitive pointer analysis, which is the major source of false positives as mentioned in their paper. String-pattern based heuristics has been adopted by RELAY to filter out false aliasing, which reduces the number of its reported races from 5022 to 161 (with false positive rate down to 20%). Those heuristics are unsound with poor generality, since they heavily depend on the code conventions in the target program.

Origins and Contexts. Prior research has proposed a variety of ways to represent contexts such as *call-site* [58, 59], *receiver object* [42] and *type* [62]. Summary-based approach [76] addresses the scalability of context-sensitive pointer analysis by introducing function summary instead of repeatedly analyzing the same function under different contexts. However, it does not perform inter-procedural pointer alias analysis (but only intra-procedural), which leads to unsoundness as they discussed in their paper. Recently, selective context-sensitive techniques [14, 15, 22, 34, 35, 39, 63, 67] have also been proposed. Although much progress has been made, context-sensitive pointer analysis remains difficult to scale.

Using threads as context in pointer analysis [7, 49, 51, 66] is not new. However, existing techniques either compromise precision due to unsound treatment of thread interactions, or lose scalability due to expensive value-flow analysis. Moreover, they do not deal with events. Many dataflow analysis techniques [5, 23, 25, 40, 74, 77] are proposed for event-driven programs to correctly model event lifecycles and event handlers, but they only scale to hundreds of lines of code. Other algorithms for multithreaded programs are not general as they target specific analyses (*e.g.,* escape analysis and region-based allocation [56], synchronization elimination [17, 52]), or only work for structured multithreading (*e.g.,* Cilk [13, 53]).

3 Origin-Sensitive Analyses

In this section, we present our origin-sensitive algorithms, including origin-sensitive pointer analysis (OPA) and origin-sharing analysis (OSA). OPA leverages origins to reason about threads and events, and it is also a key component of OSA.

3.1 Automatically Identifying Origins

In general, 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 as code annotations, we aim to automatically extract them from common code patterns in multithreaded and event-driven programs. Our technique 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` `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 2 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`.
- *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.

Table 3. The rules of OPA for Java programs. Consider the following statements are in method `m()` with Origin $\mathbb{O}_i$, denoted $\langle m, \mathbb{O}_i \rangle$. The edges $\rightarrow$ are in the PAG and $\rightsquigarrow$ in the call graph.

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

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) [31] to represent points-to relations between pointers and objects. To achieve a good precision, the PAG constructed by OPA is built together with the call graph (a.k.a. on-the-fly pointer analysis [31]). The key difference is that the context of pointers here is represented by origins. The rules of OPA for Java are summarized in Table 3. A set of similar rules can be inferred for other programming languages.

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 $\langle o, \mathbb{O}_i \rangle \rightarrow \langle x, \mathbb{O}_i \rangle$ 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.

We do not distinguish between different array indexes in statements ❺❻, unless all array accesses in the program have a constant index value. Although there exists a large body of work that can infer the content of arrays, in general, the value of index *idx* is undecidable statically. Therefore, we abstract every array with a single field (represented by $*$) and consider all array accesses as to that single field.

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 $\mathbb{O}_k$. Because to precisely determine a virtual call target and its context (*e.g.,* the call on line 13 in Figure 2), we need the type of its receiver object o and the origin of which thread executes the target. Hence, the target origin needs to be consistent with its caller's, no matter the caller is an entry point or not.

```
1  public static void main(){
     //Tmain
2    TA a = new TA();//oa→Ta
3    TB b = new TB();//ob→Tb
4    a.start(); b.start();
5  }
6  Class TA extends T {
7    TA() { super(); ... }
8  }
9  Class TB extends T {
10   TB() { super(); ... }
11 }
```

```
12 Class T {
13   Object f;
14   T() { f = new Object();
      //without switch: ⟨of,Tmain⟩
      //with context switch:
      //⟨of,Ta⟩ and ⟨of,Tb⟩
15   }
16   public void run(){
17     f.do_something();
18   }
19 }
```

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

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 avoids 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 show two origins (`Ta` and `Tb`) allocated in origin `Tmain`, which share the same super constructor `T()`. If we analyze the two allocations in `Tmain`, only one object o_f will be allocated for field f on line 14. This causes $pts(o_a.f) = pts(o_b.f) = \{\langle o_f, Tmain \rangle\}$, 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 line 2 and 3.

To identify origin allocations on-the-fly, we check the type of the allocated object against the classes in Table 2, *i.e.*, if it implements interface `Runnable` or event handler `handleEvent()`. Context switch on ❽ can efficiently separate data flow 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. To be specific, 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.

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 several such cases in Redis where thread creations are nested.

3.3 Origin-Sharing Analysis

Section 1.2 introduces our origin-sharing analysis (OSA), including a sample output in Figure 2(d). This section presents the OSA algorithm. 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 [8, 17, 56] – 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 3), 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 3), 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 3) (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;

Algorithm 1 Origin-Sharing Analysis

```
1:  Global States:
2:  OPA - origin-sensitive pointer analysis.
3:  visitedMethods ← ∅;                          ▷ flag visitedMethods
4:  m ← main;                                    ▷ the main method
5:  VisitMethod(m);
VisitMethod(m):
6:  visitedMethods.add(m);
7:  for s ∈ m.statements do
8:    switch (s)
9:      case x.f:                                ▷ read/write object field
10:       origins ← FindPointsToOrigins(p);
11:       for ℙ ∈ origins do
12:         ComputeOriginSharing(s,f,ℙ,read/write);
13:       break;
14:     case x[idx]:                             ▷ read/write array
15:       origins ← FindPointsToOrigins(a);
16:       for ℙ ∈ origins do
17:         ComputeOriginSharing(s,∗,ℙ,read/write);
18:       break;
19:     case m(args):                            ▷ call a new method
20:       for m' ∈ FindCalleeMethods(m(args)) do
21:         if !visitedMethods.contains(m') then
22:           VisitMethod(m');
23:       break;
24:     default: break;
25:   end switch
ComputeOriginSharing(s, f, ℙ, isWrite):
26: Input: s - the statement;
27:        f - accessed field (∗ means array access);
28:        ℙ - origin;
29:        isWrite - true means write and false means read.
30: WO ← GetWriteOrigins(s,f);
31: RO ← GetReadOrigins(s,f);
32: if isWrite && !WO.contains(ℙ) then
33:     WO.add(ℙ);
34: if !isWrite && !RO.contains(ℙ) then
35:     RO.add(ℙ);
```

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 $x.f = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**$(\langle o, \mathbb{O}_k\rangle.f)$
❹ $x = y.f$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **read**$(\langle o, \mathbb{O}_k\rangle.f)$
❺ $x[idx] = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**$(\langle o, \mathbb{O}_k\rangle.*)$
❻ $x = y[idx]$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **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*: **call**$(\langle f, \mathbb{O}_i\rangle) \Rightarrow \mathbf{f_{first}}(\langle f, \mathbb{O}_i\rangle)$, $\mathbf{f_{last}}(\langle f, \mathbb{O}_i\rangle) \Rightarrow \mathbf{call_{next}}(\langle f, \mathbb{O}_i\rangle)$
① *synchronized*$(x)\{ \ldots \}$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **lock**$(\langle o, \mathbb{O}_k\rangle)$, **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*: **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*: $\mathbf{origin_{last}}(\mathbb{O}_j) \Rightarrow$ **join**$(\mathbb{O}_j, \mathbb{O}_i)$

- 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.

4 Static Data Race Detection

Since the use of origins enables a more precise pointer analysis (OPA) and identifying shared- and local-memory accesses by threads and events (OSA), it can benefit any analysis that requires analyzing pointers or ownership of memory accesses, *e.g.,* deadlock, over synchronization, and memory isolation. This section presents one key usage of origin-sensitive analyses: static data race detection. Our detection algorithm is highly optimized to achieve scalability and precision.

Powered by origins, we can 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 [46] (although our is static). More specifically, our detection represents happens-before relations by a static happens-before (SHB) graph [36, 79], 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 $\Rightarrow$): one points from the call node to the first node (f_{first}) of its target method `f` within the same origin $\mathbb{O}_i$, the other points from the last node (f_{last}) of$\langle f, \mathbb{O}_i\rangle$ to the next node after the call ($\text{call}_{\text{next}}$). For lock operation ① (*e.g.,* `synchronized` blocks and methods in Java programs), we create lock and unlock nodes to maintain the current lockset. Intra-origin HB edges are created by pointing from one intra-origin node to another in their statement order.

For origin entry method call 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 ($\text{origin}_{\text{first}}$) 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 ($\text{origin}_{\text{last}}$) of current origin ($\mathbb{O}_j$) to the join node.

Existing static race detection (such as [79]) 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 protection.

However, the efficiency is limited by the redundant work in graph traversals and lockset retrievals for all pairs of memory accesses. Moreover, the straw man approach cannot scale to real-world programs which can generate large SHB graphs with millions of memory accesses. To address these limitations, we make the following three 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 an unique *integer ID* to each node, which is monotonically increased during the SHB construction. Therefore, we convert the traversal of visiting all intra-origin node 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 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

Table 5. Comparing performance of O2 with other techniques on JVM programs (Time: s).

	0-ctx		O2			RacerD	1-CFA		2-CFA		1-obj		2-obj	
App	**PA**	**Total**	**#O**	**PA**	**Total/SD**	**Total**	**PA**	**Total/SD**	**PA**	**Total/SD**	**PA**	**Total/SD**	**PA**	**Total/SD**
Avrora	*13.42*	20.70	4	*15.56*	17.85/**-14%**	47.44	*17.81*	30.63/0.48x	*56.06*	615.42/29x	*50.30*	1064/50x	*>4h*	-
Batik	*7.22*	14.47	4	*9.83*	14.93/3%	87.01	*49.28*	84.01/4.81x	*2606.15*	2648/182x	*>4h*	-	*>4h*	-
Eclipse	*5.03*	7.21	4	*6.52*	8.03/11%	*	*8.43*	20.35/1.82x	*8.71*	40.36/4.60x	*11.84*	641.38/88x	*>4h*	-
H2	*49.95*	58.13	3	*111.37*	169.63/192%	61.42	*192.71*	263.00/3.52x	*1396.69*	3208/54x	*>4h*	-	*>4h*	-
Jython	*25.77*	163.49	4	*66.34*	537.63/229%	161.22	*16.09*	100.35/-0.39x	*58.85*	172.46/0.05x	*>4h*	-	*>4h*	-
Luindex	*10.23*	14.43	3	*15.73*	20.19/40%	*	*16.74*	31.62/1.19x	*26.38*	1634/112x	*>4h*	-	*>4h*	-
Lusearch	*5.06*	7.09	3	*5.66*	7.99/13%	*	*31.60*	33.79/3.77x	*2383.72*	2401/338x	*6.48*	8.58/0.21x	*6.63*	15.05/1.12x
Pmd	*5.50*	25.07	3	*5.75*	13.32/**-47%**	34.80	*6.04*	57.21/1.28x	*8.00*	122.93/3.90x	*>4h*	-	*>4h*	-
Sunflow	*3.13*	20.67	9	*5.68*	26.01/26%	1.23	*5.08*	297.05/13x	*5.44*	2408/116x	*5.12*	3007/144x	*>4h*	-
Tomcat	*3.77*	7.58	6	*10.18*	16.87/123%	5.67	*30.47*	66.48/7.77x	*2311.45*	2829/372x	*5.89*	2918/384x	*676.59*	9589/1264x
Tradebeans	*4.96*	8.19	3	*6.05*	12.05/47%	91.73	*6.76*	16.49/1.01x	*7.54*	111.80/13x	*>4h*	-	*>4h*	-
Tradesoap	*6.25*	10.22	3	*7.44*	10.77/5%	88.12	*8.33*	24.32/1.38x	*9.31*	149.53/14x	*>4h*	-	*>4h*	-
Xalan	*31.30*	34.71	3	*35.73*	42.87/24%	2.09	*65.51*	79.33/1.29x	*3921.65*	5722/164x	*213.99*	3h/305x	*>4h*	-
ConnectBot	*2.40*	2.49	11	*5.45*	5.57 /124%	*	*23.85*	23.99/8.63x	*3512.66*	3512.93/1409x	*>4h*	-	*>4h*	-
Sipdriod	*5.80*	16.02	15	*31.48*	228.33/1327%	*	*14.33*	40.88/1.55x	*3436.02*	3452.33/215x	*>4h*	-	*>4h*	-
K-9 Mail	*6.56*	8.59	23	*14.73*	19.49/127%	*	*30.88*	33.32/2.88x	*4284.43*	4288.31/498x	*>4h*	-	*>4h*	-
Tasks	*6.90*	7.10	7	*12.72*	12.90/82%	*	*117.63*	117.77/15.59x	*8080.92*	8081.12/1137x	*>4h*	-	*>4h*	-
FBReader	*6.66*	7.49	15	*20.16*	23.33 /211%	*	*45.26*	52.79/6.05x	*2.97h*	3.10h/1482x	*>4h*	-	*>4h*	-
VLC	*5.35*	5.39	4	*46.40*	46.44/762%	*	*25.40*	25.44/3.72x	*3234.61*	3234.68/599x	*>4h*	-	*>4h*	-
FireFox Focus	*3.84*	4.08	8	*15.46*	15.76/286%	12.61	*17.96*	18.34/3.50x	*>4h*	-	*>4h*	-	*>4h*	-
Telegram	*20.82*	41.76	134	*199.79*	372.93/793%	*	*83.31*	171.42/3.10x	*>4h*	-	*>4h*	-	*>4h*	-
Zoom	*36.77*	37.62	15	*148.01*	149.01/296%	*	*198.59*	200.47/4.33x	*>4h*	-	*>4h*	-	*>4h*	-
Chrome	*6.14*	7.35	34	*108.76*	111.79/1421%	*	*18.43*	22.72/2.09x	*>4h*	-	*>4h*	-	*>4h*	-
HBase	*41.96*	>4h	16	*494.64*	1.34h/**-66.5%**	526.32	*61.75*	>4h	*>4h*	-	*>4h*	-	*>4h*	-
HDFS	*29.03*	>4h	12	*102.83*	499.53/**-28x**	493.17	*40.35*	>4h	*165.05*	>4h	*>4h*	-	*>4h*	-
Yarn	*416.37*	>4h	14	*603.70*	1.7h/**-57.5%**	126.63	*61.42*	>4h	*55.58*	>4h	*>4h*	-	*>4h*	-
ZooKeeper	*14.40*	>4h	40	*33.45*	271.20/**-53x**	18.88	*15.32*	>4h	*33.31*	>4h	*>4h*	-	*>4h*	-

"-": The corresponding pointer analysis runs out of time. "*": RacerD cannot successfully run on this program due to exceptions.

locksets, are protected by the same lock(s), and the intersection IDs between two locksets can be cached for later checks.

Lock-region-based Race Detection. We observe that a synchronization block or method always protects a large sequence of memory accesses on the same origin-shared object(s), which incurs redundant race checking. Instead, we treat those memory accesses within the same lock region as a single memory access, 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.

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*.

Benchmarks and Metrics. We also run O2 on the 13 multithreaded programs in Dacapo benchmark [6], which are commonly used to evaluate pointer analysis algorithms [26, 34–36, 62, 63, 68]. For the evaluation of C/C++ applications, we use three popular applications (*Memcached*, *Redis* and *Sqlite3*) to show the scalability of OPA.

The execution of OPA, OSA and our static race detection are all fully automated. Users are welcome to add annotations to specify origins, however, it is not required. We use no annotation in our evaluation of O2.

For the evaluation of pointer analysis, we use context-insensitive pointer analysis (denoted *0-ctx*) as the baseline, and compare OPA with different context-sensitive pointer analysis algorithms (*i.e.,* 1-CFA, 2-CFA, 1-obj, 2-obj from LLVM and WALA implementation).

The baseline of static race detection evaluation is the open source tool [36] utilizing the points-to result from *0-ctx*. We compare O2 (OPA and our static data race detection algorithm) with the existing tool that utilizes the results of previously listed context-sensitive algorithms.

Moreover, we compare O2 with the state-of-the-art static race detector, RacerD (v0.17), which adopts an ownership

Table 6. Performance comparisons in Doop.

app	#loc	#pointer	#obj	#3-CFA	origin	0-Ctx
lucene	637K	1.8M	29K	1283s	67s	51s
jython	402K	2M	53K	>90mins	2955s	324s

analysis for anti-aliasing of allocated objects. Meanwhile, user annotations are useful to improve the precision of RacerD, but are not necessary for its race detection. Hence, no annotation is used in our evaluation.

We set the time budget of analyzing a JVM benchmark to 4 hours (for pointer analysis and race detection, respectively) and a C/C++ benchmark to 12 hours (for pointer analysis only). To complete most experiments within time, we excluded certain libraries (*e.g., java.awt.** and *java.time.**).

We compare OSA with an open-source escape analysis TLOA [17], which is integrated in the state-of-the-art static analysis framework Soot [3]. TLOA uses context-sensitive information flow analysis to determine whether a field can be accessed by multiple threads. We set this time limit to 1 hour.

5.1 Unify Threads with Android Events

Mobile applications are a representative class of modern software that contains complex interations 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 [64]. 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 that 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 a 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.

Table 7. Comparing performance of OPA with other pointer analyses on C/C++ benchmarks (Time: s).

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%	-

5.2 Performance for Java, Android and C/C++

Pointer Analysis. Table 5 summarizes the performance of different pointer analysis algorithms (denoted *PA*) on the JVM benchmarks. Overall, OPA significantly outperforms all the other context-sensitive pointer analysis algorithms by 1x, 152x, at least 390x and at least 465x speedup on average, respectively, and it has only a small performance slowdown compared to 0-ctx (1.76x on average). In particular, the majority of benchmarks running 1-obj and 2-obj cannot terminate within 4 hours.

We 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 and k-obj (which cannot finish in 4 hours).

Table 6 reports the results of another implementation of OPA in Doop [61], a popular static analysis framework that implements a variety of pointer analyses using Soufflé datalog [65]. It provides many algorithms with the state-of-the-art performance. We compared the performance with its built-in k-CFA algorithm ($k = 2$) and context-insensitive analysis for analyzing two large codebases: *Lucene* and *Jython*. For *Lucene*, origin-sensitive analysis is only slightly slower than context-insensitive analysis. For *Jython*, origin-sensitive analysis finishes in 2955s, though it is 9x slower than context-insensitive analysis. This is because *Jython* has a more complex pointer-assignment graph than *Lucene* (though with a smaller codebase). However, for both benchmarks, 2-CFA cannot finish in 90 minutes (default timeout in Doop).

Table 7 reports the performance for three C/C++ applications. 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) while our approach 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 5.5.

Table 8. Performance of OSA (including the time of running OPA).

App	#O	#Shared Access	Time
Avrora	4	16	16.72s
Batik	4	293	7.79s
Eclipse	4	343	9.22s
H2	3	2,207	2.3min
Jython	4	13,121	4.5min
Luindex	3	2,001	1.6min
Lusearch	3	252	7.01s
Pmd	3	300	8.57s
Sunflow	9	1,603	10.15s
Tomcat	6	700	15.39s
Tradebeans	3	45	7.43s
Tradesoap	3	37	9.12s
Xalan	3	14	1.2min

OSA vs Escape Analysis. Table 8 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.

Static Race Detection. Table 5 reports the performance for race detection, including the time (denoted *Total*) of running the race detection and its corresponding pointer analysis. The slowdown (denoted *SD*) uses different context-sensitive algorithms compared with 0-ctx. 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 context-insensitive analysis, 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.

We reports the execution time of RacerD (*Total*) after its compilation. However, for 12 out of 27 benchmarks, RacerD throws exceptions during its analysis and cannot successfully complete the detection. Both O2 (69.09s on average) and RacerD (58.07s on average) have the similar performance for Dacapo benchmarks and Firefox Focus. For the evaluation of distributed systems, O2 (48min, including the execution of a whole program pointer analysis), only has 12x slowdown (on average) comparing with RacerD (4min).

Since RacerD started to support C/C++ program, we also evaluate RacerD on the three programs in Table 7 to compare with O2. However, RacerD reports no violation on *Sqlite3*, and cannot successfully run on *Memcached* and *Redis*.

Table 9. Comparing the result of O2 with other static data race detection tools.

App	0-ctx	O2	RacerD	1-CFA	2-CFA	1-obj	2-obj
Avrora	12,633	38	130	45	45	47	-
Batik	4,369	186	1,562	4,229	640	-	-
Eclipse	958	7	*	944	822	945	-
H2	9,698	2,817	6,980	7,832	6,322	-	-
Jython	7,997	3,651	72,304	2,402	2,358	-	-
Luindex	3,218	1,792	*	2,821	2,271	-	-
Lusearch	567	341	*	538	494	529	526
Pmd	307	256	7	296	293	-	-
Sunflow	9,238	1,925	62	6,868	5,899	2,288	-
Tomcat	751	307	2,792	701	693	585	575
Tradebeans	193	75	90	171	168	-	-
Tradesoap	264	64	90	179	177	-	-
Xalan	6	1	404	6	6	6	-
Reduced FP	0%	77%	89%	46%	60%	35%	19%

"-": the corresponding pointer analysis runs out of time.
"*": RacerD cannot successfully run on this program due to exception.

Table 10. #Races and #Origin-shared objects in distributed systems.

App	0-ctx #Shared	O2 #Shared/#Race	RacerD #Race	1-CFA #Shared	2-CFA #Shared
HBase	1,269	903/687	727	1,799	-
HDFS	2,322	1,066/910	2,797	3,139	6,605
Yarn	5,387	1,162/1,164	2,619	3,083	2,146
ZooKeeper	1,389	1,271/747	404	2,511	4,299

"-": the corresponding pointer analysis runs out of time.

5.3 Precision of Origin-Sensitivity

Our precision evaluation is different from the other existing pointer analysis techniques [26, 34–36, 62, 63, 68] that adopt metrics such as the numbers of reachable methods, call graph edges, cast operations that may fail, or virtual call sites that cannot be disambiguated into monomorphic calls. Those metrics are designed for general purpose, which cannot reflect the imprecision problem only available in multithreaded and event-driven programs. Moreover, multithreaded and event-driven programs have different precision requirements from their applications. Hence, it is more convincing to use the number of reported races to show the end-to-end precision.

Table 9 and 10 report the number of detected races with different pointer analyses. The report of RacerD includes two types of thread safety violations for the evaluated benchmarks: (1) read/write race, and (2) unprotected write violation 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 10, since all the other context-sensitive detections run out of time (>4h).

The last row in Table 9 (*Reduced FP*) shows the false positives reduced by different detections comparing with 0-ctx (exclude RacerD). In summary, 1-origin reduces 77% false positives, while 1- and 2-CFA reduce 46% and 60% false positives and 1- and 2-obj reduce 35% and 19%. For the majority of benchmarks, 1-origin reports significantly fewer races, *e.g.*, *Eclipse*. For other benchmarks, O2 is much faster to achieve a similar precision. For example, 1-origin reports 38 races for *Avrora*, both 1- and 2-CFA report 45 races, and 1-obj reports 47 races (while being 60x slower than 1-origin).

O2 reports, on average, 6x less races compared to RacerD from both tables (reduced 89% false positives from the data in Table 9). More importantly, O2 is sound and will not miss races: as shown in Table 11, 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.

5.4 Discussions

Trade-off between Precision and Performance. Table 5 compared O2 with k-CFA and k-obj for $k \leq 2$. O2 significantly outperforms k-CFA and k-obj on not only performance but also precision. This is a clear indication to us that origin is a more precise context than k-CFA and k-obj when k is not large enough. 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.

Trade-off between OPA and O2 detection. The performance of 1-origin 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 origin-shared objects as shown in Table 10, which means less workload in both checking happens-before relations and computing common locksets.

5.5 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 11. 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 manaully 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`).

```
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.
        tz_minuteswest; //RACE
4   vdata[CS_HRES_COARSE].tz_dsttime = sys_tz.tz_dsttime;
        //RACE
5   ... }
```

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 thread 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 a recent version (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

[1] /linux-stable/drivers/gpio/gpiolib.c@1104:8
[2] /linux-stable/kernel/irq/manage.c@1279:7 and @1282:7

Table 11. 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

such as `stats`, `settings`, `time_out`, or `stop_main_loop`. There are also three races that are not on these variables and look more harmful. We reported them 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:

```
1 void *do_slabs_reassign(){ ... //event
2   if (slabsclass[id].slabs > 1){
3     return cur;//RACE: missing lock
4 }}
5 void *do_slabs_newslabs(){ ... //thread
6   pthread_lock();
7   p->slab_list[p->slabs++] = ptr;//with lock
8   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 tries to allocate new slabs. It may result in unexpected allocation failures upon occurrence. This case is interesting as it shows that unlike previous tools, which only reason about *inter-thread* races, O2is able to unify events and threads to find races in complex programs that leverage both concepts for concurrency.

FireFox Focus. O2 is 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:

```
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
11   ... }
```

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
14 }
```

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 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 new race in *HBase* has the same reason above, involving two concurrent accesses on a map, *keyProviderCache*, without proper locks from method `getKeyProvider()` (in class `org.apache.hadoop.hbase.io.crypto.Encryption`).

6 Conclusion and Future Work

We have proposed a novel abstraction, *origin*, that unifies threads and events to effectively reason about shared memory and pointer aliases in in multithreaded and event-driven software. Underpinned by origins, we have developed a new techique, O2, to automatically detect data races in real-world Java and C/C++ programs. Our extensive evaluation 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. At the time of writing, O2 has been integrated into a commerical static analyzer. We are in the process of implementing O2 for other languages such as Golang, C# and Rust.

References

[1] [n.d.]. Infer : RacerD. http://fbinfer.com/docs/racerd.html.

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

[3] 2019. Sable/soot: Soot - A Java optimization framework. https://github.com/Sable/soot.

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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*.

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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

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

[26] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

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

[28] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for

the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

[29] 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

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

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

[32] 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

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

[34] 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)*.

[35] 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.

[36] 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

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

[38] 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.

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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.

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

[60] 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.

[61] Yannis Smaragdakis. [n.d.]. Doop - Framework for Java Pointer Analysis. https://bitbucket.org/yanniss/doop/.

[62] 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

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

[64] 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.

[65] Soufflé. 2020. Soufflé - Logic Defined Static Analysis. https://souffle-lang.github.io.

[66] 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

[67] 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.

[68] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[69] 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

[70] 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

[71] 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

[72] 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

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

[74] 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

[75] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[76] Yichen Xie and Alex Aiken. 2007. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 16–es. https://doi.org/10.1145/1232420.1232423

[77] 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).

[78] 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

[79] 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)*.