YSC4231: Parallel, Concurrent and Distributed Programming

Data Races in Java

Races

A **race condition** occurs when the computation result depends on scheduling (how threads are interleaved on ≥1 processors.

- Only occurs if T1 and T2 are scheduled in a particular way
- As programmers, we cannot control the scheduling of threads
- Program correctness must be independent of scheduling

Race conditions are bugs that *exist only due to concurrency*

- No interleaved scheduling with 1 thread

Typically, the problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

We will distinguish between **data races** and **atomicity violations**, both of which are types of race condition bugs.

Data Races

A data race is a type of *race condition* that can happen in two ways:

- Two threads ***potentially*** write a variable at the same time
- One thread ***potentially*** write a variable while another reads

Not a race: simultaneous reads (provide no errors)

Potentially is important

- We claim that code itself has a data race independent of any particular actual execution

Java Stack Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  synchronized boolean isEmpty() {
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    return array[index--];
  }
}
```

A Race Condition: But Not a Data Race

```
class Stack<E> {
  …
  synchronized boolean isEmpty() {…}
  synchronized void push(E val) {…}
  synchronized E pop(E val) {…}

E peek() {
  E ans = pop();
  push(ans);
  return ans;
}
```

Note that peek() throws the StackEmpty exception via its call to pop()

In a sequential world, this code is iffy, ugly, and questionable *style*, but *correct*

This "algorithm" is the only way to write a **`peek`** helper method if this interface is all you have to work with.

`peek` in a Concurrent Context

`peek` has no *overall* effect on the shared data

- It is a "reader" not a "writer"
- State should be the same after it executes as before

This implementation creates an inconsistent *intermediate state*

- Calls to `push` and `pop` are synchronised, so there are no ***data races*** on the underlying array
- But there is still a ***race condition***
- This intermediate state should not be exposed
  - Leads to several *atomicity violations*

```
E peek() {
    E ans = pop();
    push(ans);
    return ans;
}
```

Example 1: peek and isEmpty

Property we want:
If there has been a **`push`** (and no **`pop`**),
then **`isEmpty`** should return **`false`**

With **`peek`** as written, property can be violated – how?

Time

Thread 1 (`peek`)	Thread 2
`E ans = pop();`	`push(x)`
	`boolean b = isEmpty()`
`push(ans);`	
`return ans;`	

Example 1: peek and isEmpty

Property we want:
If there has been a **`push`** (and no **`pop`**), then **`isEmpty`** should return **`false`**

Race causes error with:
T2: push(x)
T1: pop()
T2: isEmpty()

With **`peek`** as written, property can be violated – how?

Time

Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
boolean b = isEmpty()
```

Example 2: peek and push

Property we want:
Values are returned from pop in LIFO order

With **peek** as written, property can be violated – how?

Time

Thread 1 (peek)

```
E ans = pop();

push(ans);

return ans;
```

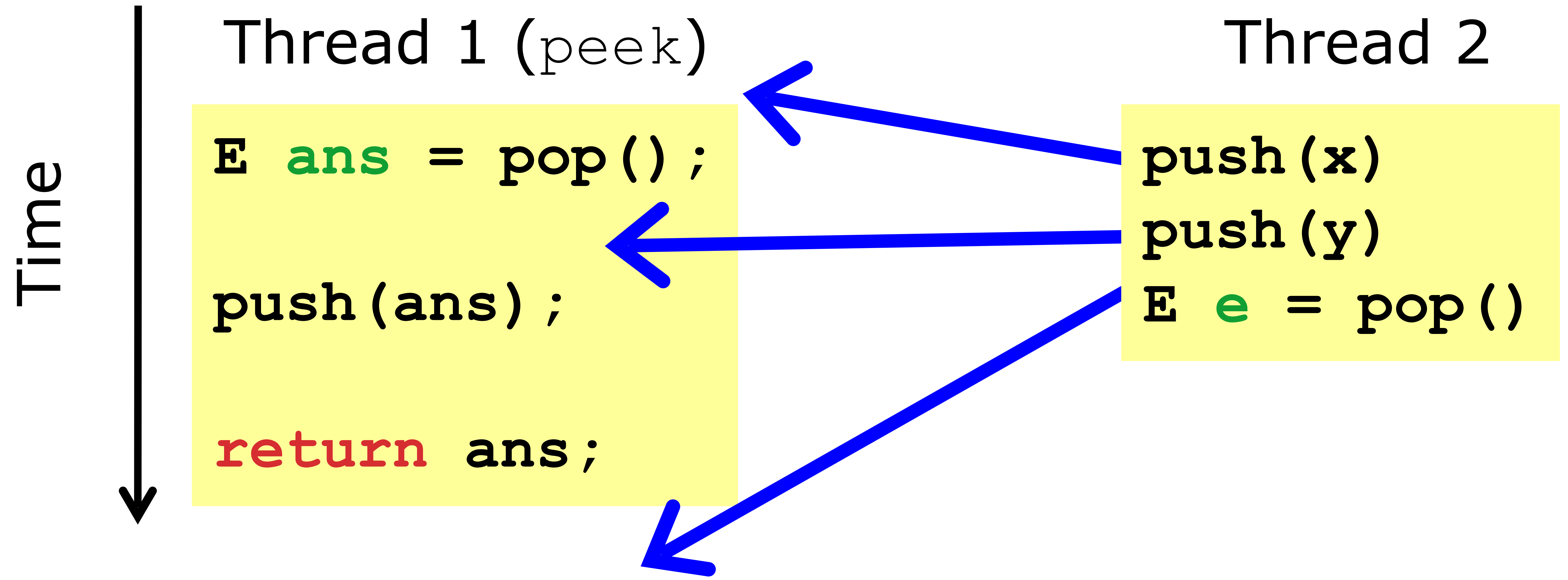
Thread 2

```
push(x)
push(y)
E e = pop()
```

Example 2: peek and push

Property we want:
Values are returned from pop in LIFO order

With **`peek`** as written, property can be violated – how?

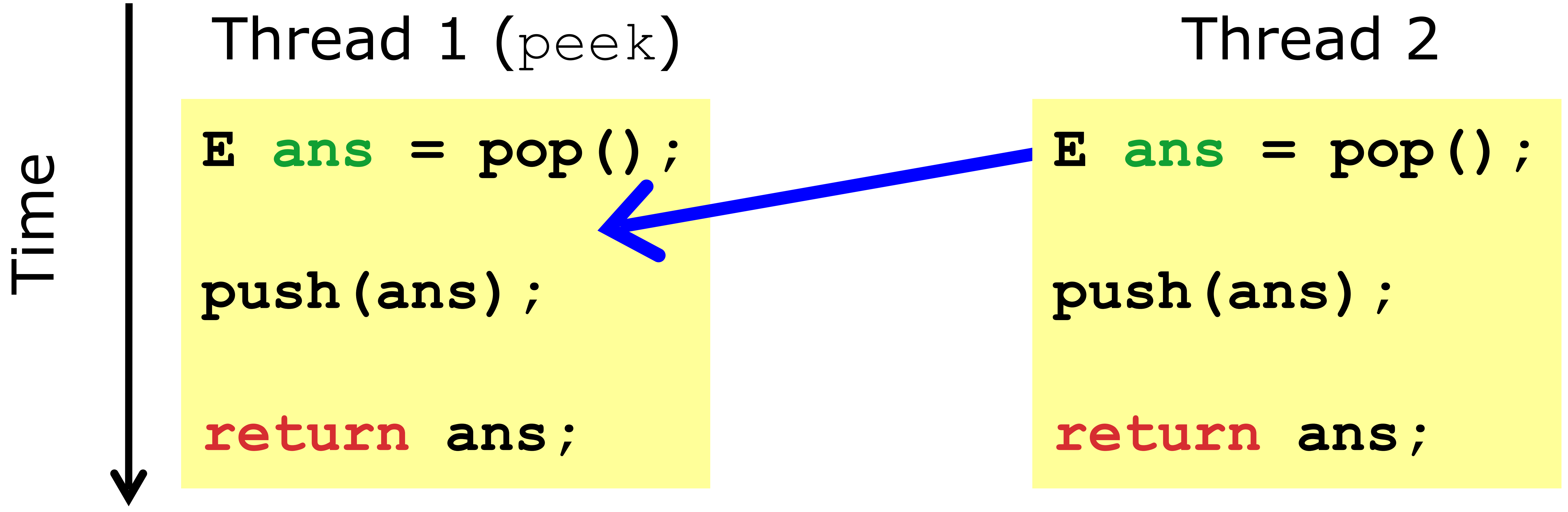


Race causes error with:
T2: push(x)
T1: pop()
T2: push(x)
T1: push(x)

Example 3: peek and peek

Property we want:
`peek` does not throw an exception unless the stack is empty

With **`peek`** as written, property can be violated – how?



The Fix?

`peek` needs synchronisation to disallow interleavings

- The key is to make a *larger critical section*
- This protects the intermediate state of `peek`
- Use re-entrant locks; will allow calls to **`push`** and **`pop`**
- Can be done in stack (left) or an external class (right)

```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

Another (Incorrect?) Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() {
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() {
    return array[index];
  }
}
```

Another Incorrect Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Why Wrong?

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

It *looks like* **`isEmpty`** and **`peek`** can "get away with this" because **`push`** and **`pop`** adjust the stack's state using "just one tiny step”

But this code is still *wrong* and depends on language-implementation details you cannot assume

- Even "tiny steps" may require multiple steps in implementation: **`array[++index] = val`** probably takes at least two steps
- Code has a data race, allowing very strange behaviour

Do not introduce a data race, even if every interleaving you can think of is correct!

Getting It Right

Avoiding race conditions on shared resources is difficult

- Decades of bugs have led to some conventional wisdom and general techniques known to work

We will discuss a way to automatically detect ***data races***.

RacerD:
Compositional Static Race Detection

RacerD: Compositional Static Race Detection

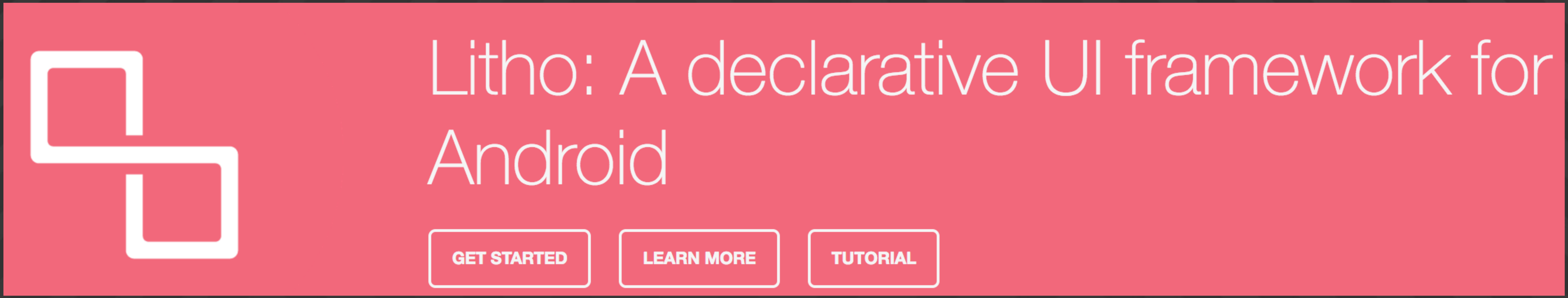
SAM BLACKSHEAR, Facebook, USA
NIKOS GOROGIANNIS, Facebook, UK and Middlesex University London, UK
PETER W. O'HEARN, Facebook, UK and University College London, UK
ILYA SERGEY*, Yale-NUS College, Singapore and University College London, UK

Automatic static detection of data races is one of the most basic problems in reasoning about concurrency. We present RacerD—a static program analysis for detecting data races in Java programs which is fast, can scale to large code, and has proven effective in an industrial software engineering scenario. To our knowledge, RacerD is the first inter-procedural, compositional data race detector which has been empirically shown to have non-trivial precision and impact. Due to its compositionality, it can analyze code changes quickly, and this allows it to perform *continuous reasoning* about a large, rapidly changing codebase as part of deployment within a continuous integration ecosystem. In contrast to previous static race detectors, its design favors reporting high-confidence bugs over ensuring their absence. RacerD has been in deployment for over a year at Facebook, where it has flagged over 2500 issues that have been fixed by developers before reaching production. It has been important in enabling the development of new code as well as fixing old code: it helped support the conversion of part of the main Facebook Android app from a single-threaded to a multi-threaded architecture. In this paper we describe RacerD's design, implementation, deployment and impact.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Concurrent programming structures**;

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom

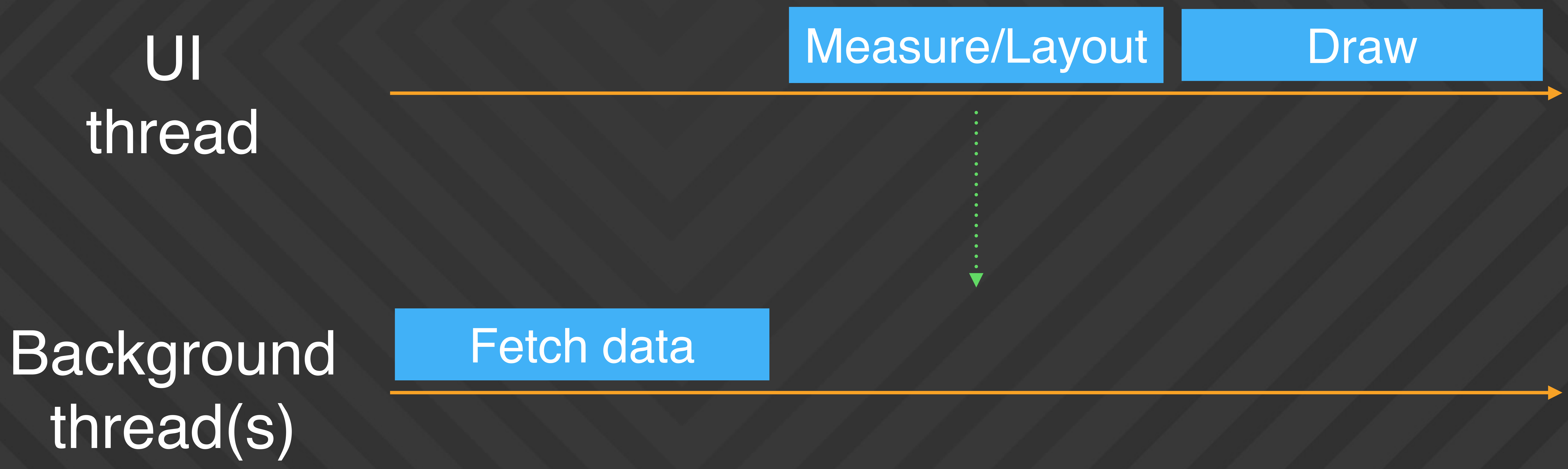
ACM Reference Format:
Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (November 2018), 28 pages. https://doi.org/10.1145/3276514



Litho Component

Fetch data	Talk to network
Measure/Layout	Determine size and position
Draw	Render and attach

Moving layout to background for better perf



BUT: to migrate, Measure/Layout step needs to be thread-safe. Otherwise...

Adding concurrency can introduce **data races**

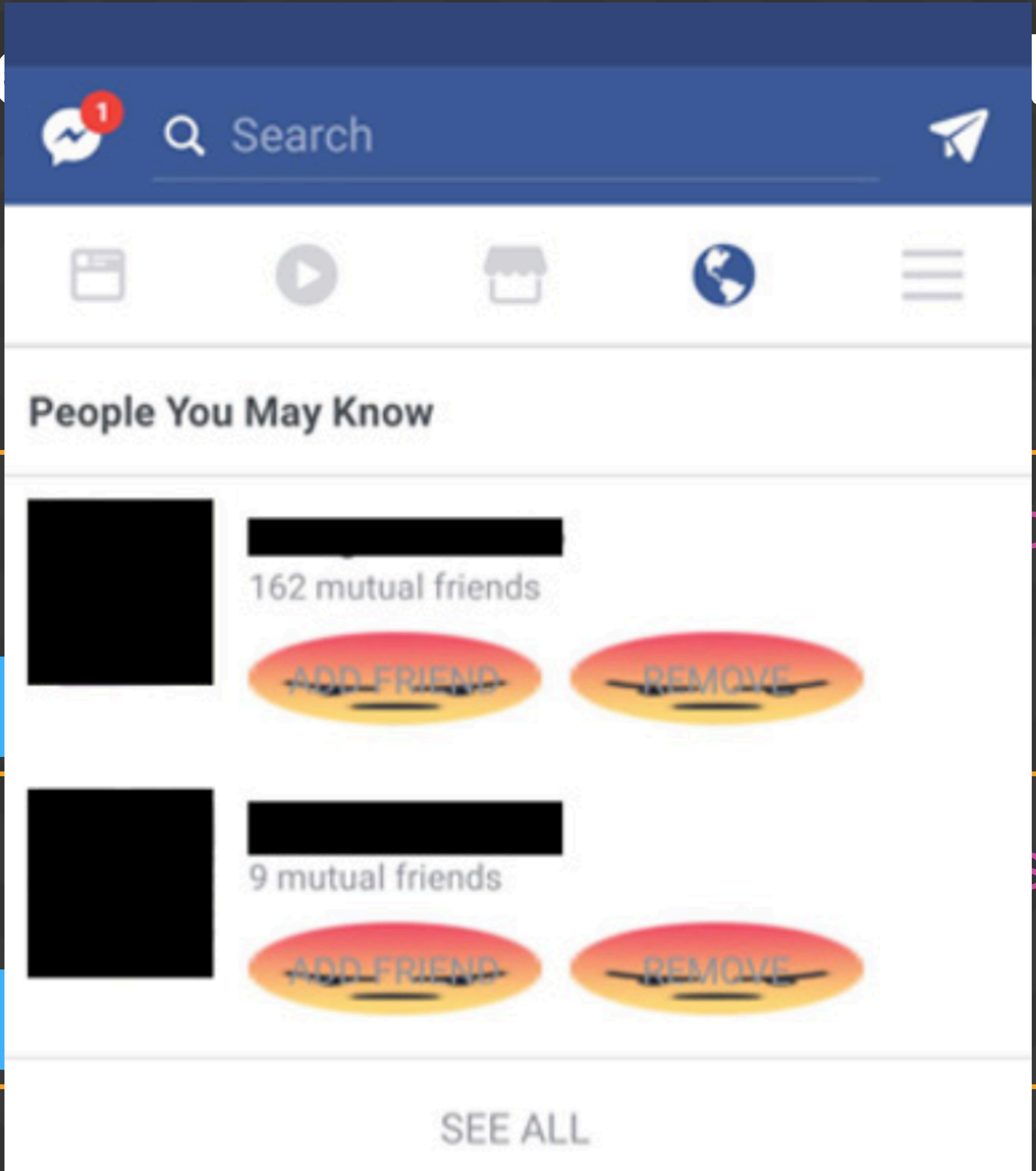
Data race:
two **concurrent** accesses to
the **same memory location**
where **at least one is a write.**

Concurr… ta races

UI thread

Background thread 1

Background thread 2



Draw

…cts

Conflicts

…s

Adding concurrency to sequential code is scary

Problem 1: 1000s of existing components. Where should we add synchronization to avoid races?

Problem 2: Nondeterminism makes it hard to test for races. How do we prevent regressions?

Static race detector can show us where to add synchronization + prevent regressions at code review time.

Devs need static analysis for migration

Talking with [redacted], one of our managers - we realized that the timeline of background layout in feed might be closely tied to the timeline of static analysis - I'm wondering if you have your roadmap already fleshed out.

Stringent requirements for helpful analysis

Interprocedural

Scalable + incremental

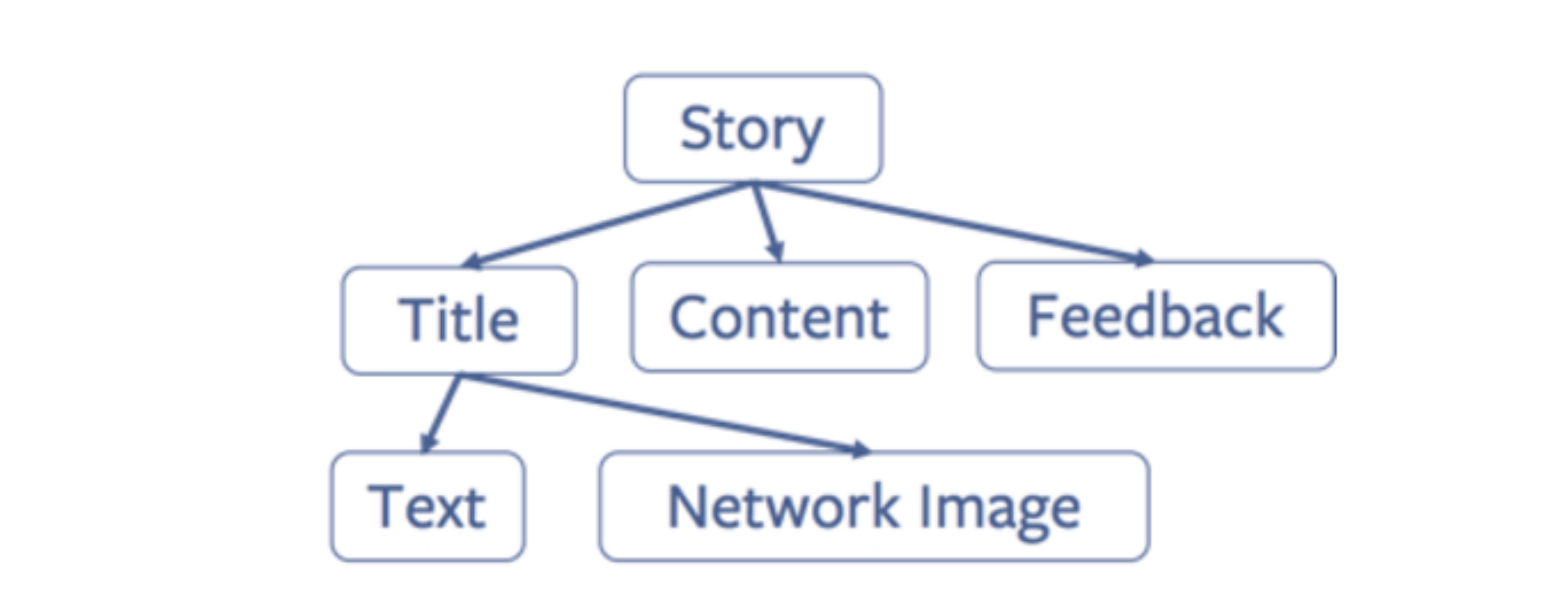
Low annotation burden

High signal >> catching all bugs

[redacted] cc [redacted] on @ThreadSafe

Will the eventual thread safe annotation be recursive? Will it check that dependencies, at least how they're used, are thread safe?

Like · Reply · Share · 2 · October 14, 2016 at 11:04pm

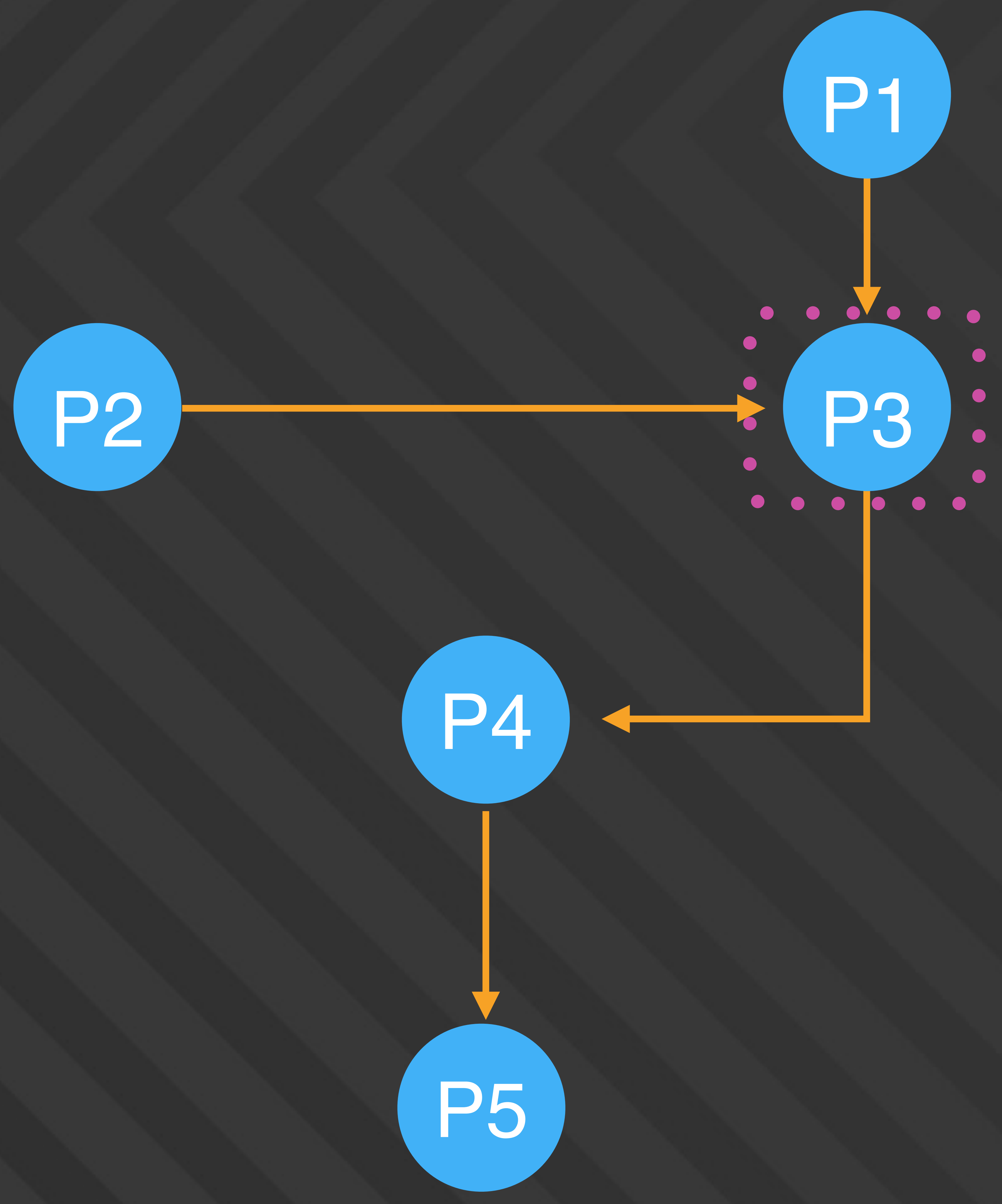



```
@GuardedBy("this")
Classname object;
```

RacerD Design Principles

- Be *compositional*; don't do whole-program analysis
- Report races between *syntactically* identical access paths; don't attempt a general alias analysis
- Reason sequentially about memory accesses, locks, and threads; don't explore interleaving
- Occam's razor; *don't* use complex techniques (unless forced)

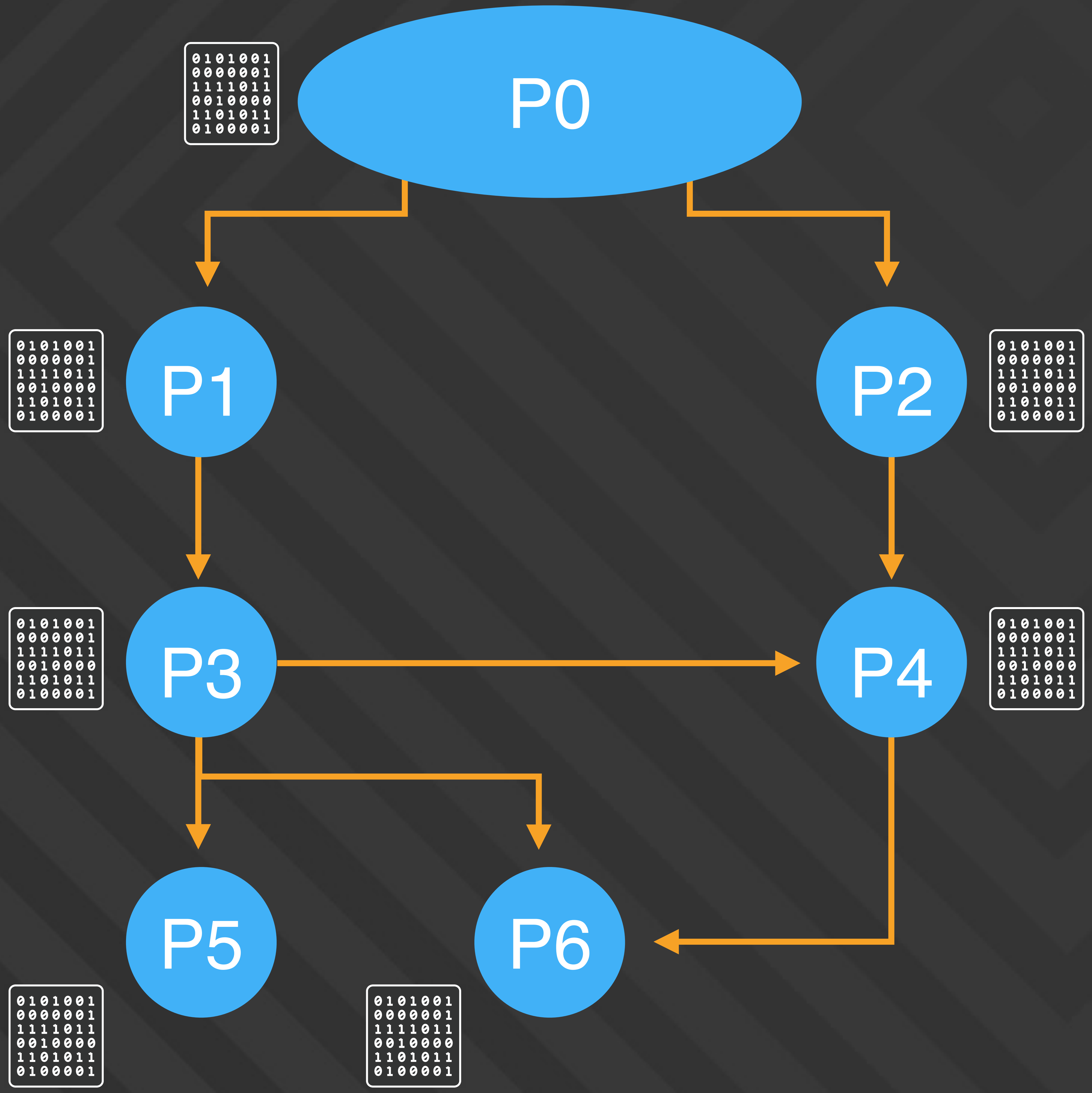
Background: compositional analysis



When analyzing P3:

- Will have summary for callee P4
- But don't know anything about callers P1, P2, or transitive callee P5
- Need to compute summary for P3 usable in *any* calling context

Background: compositional analysis



- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point

Scalable: analyze each procedure just once (without cycles)

Computing procedure summaries

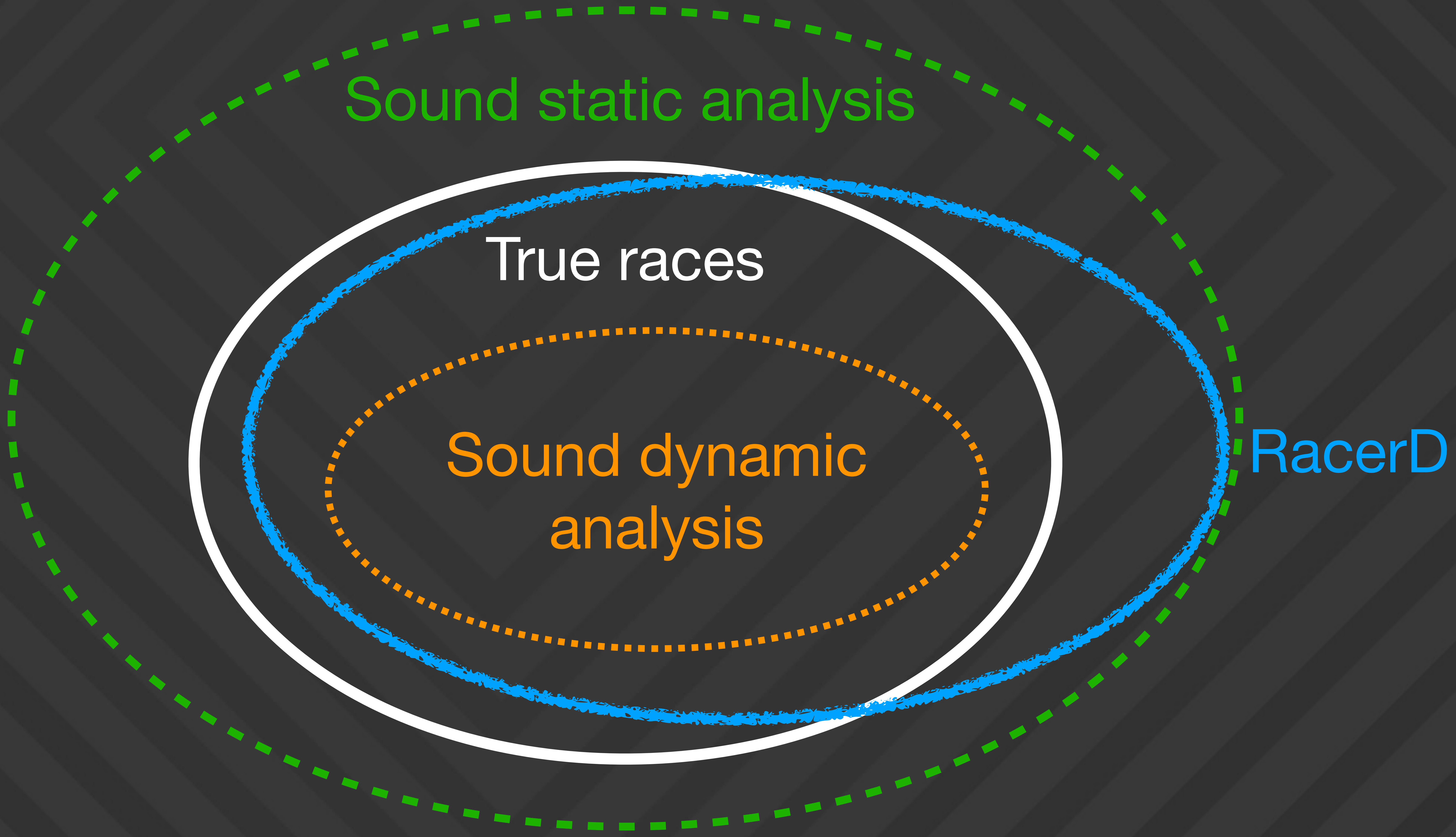
`Summary = { (access path, kind, locks) }`

get	{ (this.mCount, READ, 0) }
set	{ (this.mCount, WRITE, 0) }
reset	{ (this.mCount, WRITE, 1) }

get and reset access same memory location
reset performs a write under synchronization
get uses no synchronization

```
class Counter {
  private int mCount;

  int get() {
    return this.mCount;
  }

  private void set(int i) {
    this.mCount = i;
  }

  synchronized void reset() {
    set(0);
  }

  ...
}
```

RacerD vs Static and Dynamic Analysis tools



Finding data race regressions

Impact (1y)

~500

PROGRAMMERS REACHED

~7K

REPORTS

~4K

FIXES

Engineer Comments

better. The thread safety violations are doubly useful - since these help catch nasty and hard to debug bugs that can commonly happen in our multi-thread UI stack on Android

Infer was really instrumental in ensuring thread safety in Litho code. This allowed us to ship Newsfeed layout on a background thread and get huge wins in terms of scroll performance in FB4a

Without Infer, multithreading in News Feed would not have been tenable

Try RacerD

https://fbinfer.com/docs/racerd.html

or Google “Facebook RacerD”

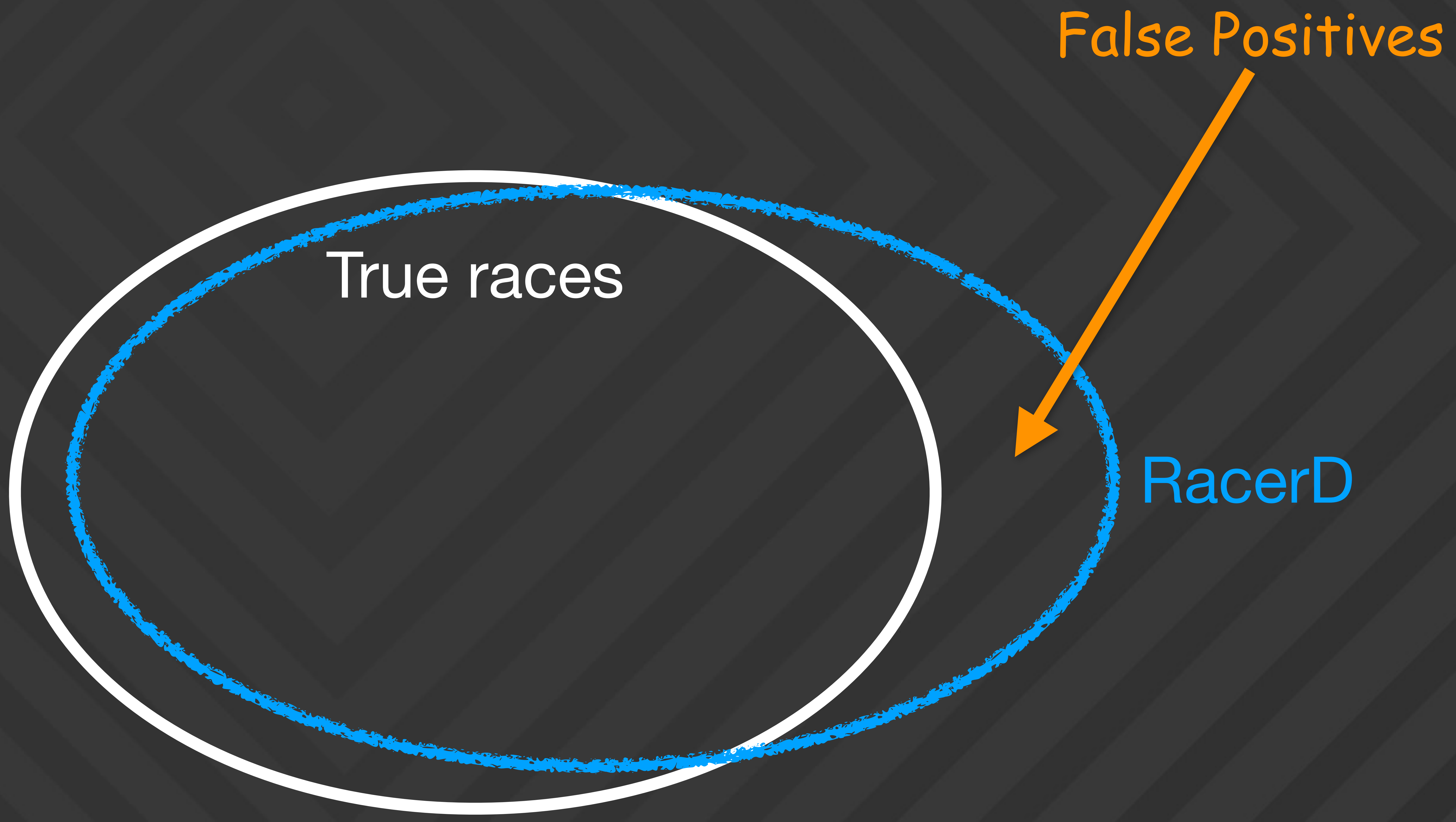
Demo

Using RacerD for simple data race detection

HW: Research Project

- Investigate **large** open-source Java projects
- Detect data races in them via RacerD
- Check the reports: False or True Positives?
- Suggest minimal fixes

Why double-checking?



HW: Research Project

- Investigate **large** open-source Java projects
- Detect data races in them via RacerD
- Check the reports: False or True Positives?
- Suggest minimal fixes

Next Week

- Functional Concurrent Programming in Java and Scala
- Controlling the Future
- Keeping the Promises