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

1 INTRODUCTION

Concurrent programming is hard. It is difficult for humans to think about the vast number of potential interactions between processes, and this makes concurrent programs hard to get right in the first place. Further, concurrency errors are difficult to debug and even reproduce after they have been observed, making them time-consuming to fix. Because of both its difficulty and importance, formal reasoning about concurrency has been studied for over 40 years and has seen many research advances. However, despite numerous interesting ideas not much of the work has made it out of the research setting and into deployment, where it can help programmers in their daily jobs.

At Facebook, we are working on techniques for automated reasoning about concurrency. RacerD, our new race detector, searches for data races—unsynchronized memory accesses, where one is a

*Work done while employed as a part-time contractor at Facebook.

Authors' addresses: Sam Blackshear, Facebook, USA, shb@fb.com; Nikos Gorogiannis, Facebook, UK, Middlesex University London, UK, nikosgorogiannis@fb.com; Peter W. O'Hearn, Facebook, UK, University College London, UK, peteroh@fb.com; Ilya Sergey, Yale-NUS College, Singapore, University College London, UK, i.sergey@ucl.ac.uk.

2475-1421/2018/11-ART144
https://doi.org/10.1145/3276514

write—in Java programs. Races are one of the most basic forms of concurrency errors, and removing them can help simplify the mental task of understanding a concurrent program. RacerD is a static analysis: it employs formal symbolic reasoning to cover many paths through a program without running it. It has been in production at Facebook for over a year, during which time over 2500 concurrency issues it has flagged have been fixed by engineers before reaching production. Also, even though it is possible to generate artificial false negatives (data races that the analysis does not flag), only three have been reported from production during the past year, and these were due to (since corrected) implementation errors on our part.

Even more significantly, RacerD has been used to support the conversion of the layout portion of the News Feed in Facebook's main Android app from a single-threaded to multi-threaded architecture. One of the Facebook Android engineers, Benjamin Jaeger, stated the impact as follows:

"*Without Infer, multithreading in News Feed would not have been tenable.*"[1]

This conversion also had a positive effect on the performance of the Android app. Thus, not only is our static analysis used to find and fix problems, it is enabling the new development of important industrial code. (Note that the quote is not saying that Infer/RacerD is the only tool that might ever exist that could have enabled multi-threading in News Feed, but rather that they would not have been able to tackle the project without it or some other similarly effective data race detector. That practicing engineers regard a static analysis as enabling code they write can be taken to be a positive for static race detection in general, not only RacerD.)

From the point of view of its users, RacerD has four important features:

(i) speed, reporting issues quickly;
(ii) scale, ability to run on large codebases;
(iii) low friction, can be run without huge startup time for developers in terms of annotating code;
(iv) effective signal, its reports are seen as valuable and not ignored by developers.

Speed and scale together are necessary to cope with the kind of *high momentum software engineering* practiced at Facebook, where a codebase in the millions of lines of code undergoes frequent modification by thousands of programmers. RacerD performs *continuous reasoning* (O'Hearn 2018a) by analyzing these code changes (known as *diffs*) as they are submitted to the internal code review process. With each diff, an instance of RacerD is spun up in Facebook's continuous integration system, so it acts as a bot participating in code review by automatically commenting on the code.

Low friction allows RacerD to achieve incremental impact on existing codebases, without starting from scratch. In particular, RacerD does not rely on annotations for saying which pieces of memory are guarded by what locks. These kinds of annotations, which are used in `@GuardedBy` type systems for Java (Ernst et al. 2016; Goetz et al. 2006) and in Rust (*e.g.*, the `Mutex` type), are useful when present, but *requiring* them to get started causes considerable friction; enough that relying on them in the conversion of Facebook Android News Feed would have been a show stopper.

Effective signal is related to the concept of *false positive rate*, but is not the same thing. It is entirely possible to have an analysis with a low false positive rate, but where developers ignore the reports; this phenomenon has been observed multiple times in industrial program analysis (Ayewah et al. 2008; Bessey et al. 2010; Sadowski et al. 2018). Effectiveness of signal is partly dependent on false positive rate, but also on speed of reporting and on the deployment model. Our continuous reasoning takes place as part of code review, in the developers' workflow, when they are thinking about a code change; it avoids the mental effort of *human context switch* which comes with, for example, batch deployment.

[1]RacerD is part of the Infer analysis platform. This quote was from https://code.facebook.com/posts/1985913448333055/multithreaded-rendering-on-android-with-litho-and-infer/

```java
@ThreadSafe
class RaceWithMainThread {
  int mCount;
  void protectedWriteOnMainThread_OK() {
    OurThreadUtils.assertMainThread();
    synchronized (this) { mCount = 1; }
  }
  int unprotectedReadOnMainThread_OK() {
    OurThreadUtils.assertMainThread();
    return mCount;
  }
  synchronized int protectedReadOffMainThread_OK() {
    return mCount;
  }
```

```java
  synchronized void
  protectedWriteOffMainThread_BAD() {
    mCount = 2;
  }
  int unprotectedReadOffMainThread_BAD() {
    return mCount;
  }
```

Fig. 1. Top: simple example capturing a common safety pattern used in Android apps: threading information is used to limit the amount of synchronization required. As a comment from the original code explains: "`mCount` is written to only by the main thread with the lock held, read from the main thread with no lock held, or read from any other thread with the lock held." Bottom: unsafe additions to `RaceWithMainThread.java`.

From the point of view of the analysis designer, the distinguishing technical property of RacerD is that it is *compositional*: the analysis result of a composite program can be obtained from the results of its parts. This means, in particular, that an "analysis result of a part" is meaningful independently of the context (composite program) it is placed in. Compositionality opens the way to analyzing a code fragment (*e.g.*, the code change under review) without knowing the context that it will run in. This is useful for checking a piece of code for thread-safety issues before placing it in a concurrent context. It is also fairly straightforward to parallelize a compositional analysis, helping it scale to big code, which is important given the size of codebases at Facebook.

Our analysis differs from much previous static analysis work in that we are not aiming to prove absence of races, but rather to report a set of high-confidence races; we are favoring reduction of false positives over false negatives. This presents different challenges, but also allows us to make tradeoffs (*cf.* Section 8) that would not be justified if the aim were to show absence of races.

In the remainder of the paper we describe RacerD's design, implementation, deployment and evaluation.[2] We present a number of design decisions which can be seen alternately as limitations or, more positively, as engineering compromises that unlock the potential for impact. The primary evidence for our claim of effectiveness comes from RacerD's deployment at Facebook. As far as we are aware, no previous static race detector has demonstrated comparable impact as part of the software development process for major industrial products. We additionally present case studies comparing RacerD to other open-source Java race detectors.

Note that we are using the terminology 'race detector' to refer to a tool which accepts a piece of code and then searches for races within that, without the human first adding annotations describing (say) the relationships between locks and fields. The latter form of tool, though still useful, is less automatic and functions like a type checker. We will describe how RacerD fits into the broader context of race detectors and other concurrency checking tools in Section 7.

2 A BEGINNING EXAMPLE

To help ground the discussion that follows we begin with a simple example in Figure 1 (top) that illustrates some of RacerD's ideas. The example is motivated by a coding pattern used in `Components*.java` classes in the Litho codebase to avoid excess synchronization.[3]

[2]A tutorial on RacerD is available at http://fbinfer.com/docs/racerd.html.
[3]https://github.com/facebook/litho

If we run RacerD on this code it doesn't find a problem. The unprotected read at line 10 and the protected write at line 6 do not race with one another because they are known to be on the same thread. This code illustrates just one way that developers try to write thread safe code while avoiding synchronization in places for performance reasons.

RacerD works by first computing a *summary* for each method in a class, which records potential accesses together with information such as whether they are protected by a lock or confined to a thread. While analysing a method, RacerD might consult summaries for methods it calls if they are present in cache, or make a recursive call to the analyzer to provide such summaries *on demand* if they are not present. The creation of summaries is done in a context-independent way; a procedure body will not need to be processed multiple times taking into account different call sites.

Next, RacerD looks through all the summaries for the class in question checking for potential conflicts. In this example, the summaries record an approximation of the information that the accesses to `mCount` are both protected by being on the same thread:

```
protectedWriteOnMainThread_OK()
Thread: true, Lock: true, Write to this.RaceWithMainThread.mCount at line 6
unprotectedReadOnMainThread_OK()
Thread: true, Lock: false, Read of this.RaceWithMainThread.mCount at line 10
protectedReadOffMainThread_OK()
Thread: false, Lock: true,  Read of this.RaceWithMainThread.mCount at line 13
```

Thus, RacerD's summary processing phase concludes that a race is not possible because the threading information for the accesses indicates mutual exclusion. On the other hand, if we include additional methods that do conflict, then RacerD will report potential races, as in Figure 1, bottom. The summaries for the additional methods record information about the accesses as follows:

```
protectedWriteOffMainThread_BAD()
Thread: false, Lock: true, Write to this.RaceWithMainThread.mCount at line 17
unprotectedReadOffMainThread_BAD()
Thread: false, Lock: false, Read of this.RaceWithMainThread.mCount at line 20
```

When we look at the summary for `protectedWriteOffMainThread_BAD` we can see the potential conflict with the unprotected read on the main thread. One is protected by a lock, and the other is protected by knowledge that it is on the main thread, but these are not sufficient to provide mutual exclusion: RacerD reports a race between this access and the one at line 8. Similarly, the access at line 20 in `unprotectedReadOffMainThread_BAD` is protected by neither a lock nor a thread. RacerD will report it as racing with both the access at line 6 and the access at line 17.

In general, RacerD accepts a Java program as input and checks each of its classes for data races between the non-**`private`** methods of the class when run in parallel with one another. It looks for "deep" races arising from arbitrarily long chains of method calls into other classes.[4] RacerD checks classes marked `@ThreadSafe` in the program, as well as classes marked with other annotations that we treat as aliases of `@ThreadSafe` (this aliasing facility was used in the application to News Feed multithreading). In addition, RacerD performs *concurrent context inference* to trigger race checking on classes that use locks or otherwise indicate that they are intended to be used concurrently, even when they are not annotated as `@ThreadSafe`. We have also developed a system of annotations in collaboration with Android engineers for documenting certain assumptions, such as that a method behaves functionally or that a method is only used on the UI thread. These annotations help the user experience, but are not necessary for getting started. RacerD can be run on any Java program without adding annotations.

[4]According to a recent survey, 65% of the races found by Infer required inspecting multiple procedures and/or multiple files: https://code.facebook.com/posts/1537144479682247/finding-inter-procedural-bugs-at-scale-with-infer-static-analyzer/

3 ANALYSIS REQUIREMENTS

In early 2016 we began working on a static analysis for concurrency, devising algorithms for proving that no data race errors whatsoever could occur within a program, The technical timeline we had in mind for creating our tool for proving data race freedom was roughly 2 years for launching a proof of concept, and then further time to refine the developer experience.

In late 2016, engineers working on Facebook for Android caught wind of our project and reached out asking to leverage our work. They were embarking on a major venture: converting part of the Android News Feed from a sequential to a multi-threaded architecture. This was challenging because the transformation could introduce concurrency errors. We decided to pivot our research, by shifting our attention from the idealized goal of eventually, perhaps in several years, automatically proving that no races could occur in real production code, to the aim of helping people now by quickly finding many races in existing code, with high signal. Together with the Android engineers, we sketched out the requirements for a *minimal viable product* to serve their needs:

(1) High signal: detect actionable races that developers find useful and respond to.
(2) Interprocedural: ability to track data races involving arbitrarily nested procedure calls.
(3) No reliance on manual annotations to specify which locks protect which bits of data.
(4) Fast: able to report in 15 minutes on modifications to a millions-of-lines codebase, so as to catch concurrency regressions during code review.
(5) Accurate treatment of coarse-grained locking (*i.e.*, not relying on custom-made synchronisation via, *e.g.*, *compare-and-swap*), as used in most of code written in production, but no need for precise analysis of intricate fine-grained synchronization (the minority, and changing rarely).

A companion paper says more on the human aspects of how the project unfolded, particularly on the interaction between science and engineering as the project unfolded (O'Hearn 2018b). For the purposes of this paper, the important point is that requirements helped us determine what to focus on, and what not to focus on.

For example, one of the most straightforward first steps in concurrency static analysis is to require that fields be annotated with the locks that guard them, and then to have the analyzer check them. But while this makes life easier for the analysis designer, it shifts effort from the analyzer to the human and would have been a non-starter for the job of making News Feed multithreaded: the Feed team needed to refactor many thousands of lines of code, and they needed to understand where concurrency issues might be lurking when doing so. It would have taken many more programmers to do this manually, too many—better for the computer to perform that search.

On the other hand, the Feed team asked for interprocedural analysis from the outset. The reason is that UI architecture is arranged as trees, such as in Figure 2, where each node corresponds to an object in a different class. Furthermore, concurrent UI layout raised the issue of "data races at a distance", where two methods in one class are called concurrently, but they race only on fields in classes accessed via interprocedural call chains. In fact, the design model of the Litho library, which is used for multi-threaded UI, is such that data races are almost never localized to the parent classes. This, together with the requirement to catch regressions during code review, meant that we had to do interprocedural race detection, fast. Neither a slow interprocedural analysis, say, an analysis taking *over an hour* for a modest size diff (hundreds of lines), on a codebase of a million lines of code, nor a fast intraprocedural analysis would do.

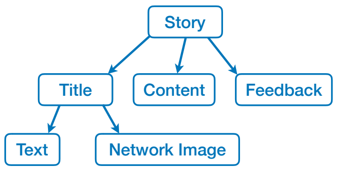

Fig. 2. UI layout of Facebook News Feed.

4 DESIGN PRINCIPLES

RacerD's goal is to find data races. Its design is based on the following ideas:

(1) Be compositional; don't do whole-program analysis.
(2) Reason sequentially about memory accesses, locks and threads; don't explore interleavings, rather, reason out of methods' summaries *wrt.* employed synchronization and memory accesses.
(3) Report races between syntactically identical access paths; don't attempt a general alias analysis.

As explained in Section 1, the first decision is motivated by the need to run RacerD in a "move fast" deployment where it prevents regressions at code review time. Compositionality is helpful both for analyzing a code change at diff time and for checking for concurrency errors before code is placed into a context to run. It is also well known that compositionality is one useful way to make an interprocedural analysis scale to large codebases (Calcagno et al. 2015, 2011).

The next two decisions come from observing the difficulty of balancing precision and scalability in the history of program analysis for concurrency. The second requirement is motivated by the sheer number of potential interleavings between threads. For example, the number of interleavings for 2 threads with n instructions each is the binomial coefficient $C(2n, n)$. This is a quantity that grows extremely rapidly. For example, suppose that $n = 40$ instructions and we have an analyzer that can process interleavings at the blazing rate of 1 billion per second. It would still take more than 3.4 million years to process them all. The inescapable conclusion is that, even though computers are powerful, we can't explore all the possibilities by brute force.

A more positive view of the second requirement is based on programming intuition. Humans do not design code by considering all potential interleavings in detail—they cannot—and we may hope that an analyzer could behave similarly. The theory of Concurrent Separation Logic (Brookes and O'Hearn 2016) encodes a programming discipline for race-free programs, and it has shown that we can get surprisingly far with purely sequential reasoning; it takes the extreme step of not exploring interleavings at all, instead tracking properties that are true inside and outside of locking. RacerD follows this idea of doing sequential reasoning and recording facts about synchronization and threading information that can provide mutual exclusion between different accesses. It then uses this information, collected for sequential executions, when considering summaries of *pairs* of methods, in order to identify executions that can race with each other.

For the third principle, precise alias analysis is a classically hard problem even for sequential programs (Smaragdakis and Balatsouras 2015). Because we focus on finding definite races, we can get away without a precise analysis for saying when arbitrary pairs of accesses may alias. We look for potentially-concurrent occurrences of syntactically identical access paths, on the intuition that they are probable races. This can miss races that can occur via distinct access paths that refer to the same memory, but it vastly simplifies the analysis and lets us focus on finding likely races.

We grant that it might be possible to improve our analysis if there were a sufficiently precise alias analysis that worked compositionally and at scale. A claim of this work is, though, that alias analysis is not *necessary* for having an impactful race analysis.

We don't claim that these design principles are the only way one might go about obtaining an effective analyzer satisfying the requirements of the previous section. Indeed, we have thought of other directions ourselves, and we hope that researchers will be motivated to go further.

5 HIGH CONFIDENCE STATIC RACE DETECTION

We divide our explanation of RacerD into two parts: an algorithm for reporting data races given procedure summaries that can answer certain semantic questions (Section 5.1), and specific abstract domains we use for computing this information in a compositional fashion (Section 5.2). The first part constitutes a framework for detecting and reporting data races, and is in a sense more

variables	$x \in \mathsf{Var}$	
field names	$\mathsf{f} \in \mathsf{Field}$	
method names	$m \in \mathsf{Method}$	
access paths	$\pi \in \mathsf{Path}$	$::= \mathsf{Var} \times \mathsf{Field}^*$
expressions	$\mathsf{e} \in \mathsf{Exp}$	$::= \pi \mid \ldots$
commands	$c \in \mathsf{Cmd}$	$::= \pi_1 := \pi_2 \mid \pi := \mathsf{new}() \mid \pi := \mathsf{call}\ m(\vec{\mathsf{e}}) \mid \mathsf{lock} \mid \mathsf{unlock}$
statements	$s \in \mathsf{Stmt}$	$::= c_1; c_2 \mid \ldots$

Fig. 3. Syntax of minimal heap-manipulating language with locks.

fundamental. The second shows one way we have found to compute the entities needed for race detection in a way that is consistent with the analyzer requirements laid out in Section 3; see Section 6 for experimental justification for our claim of effectiveness. In a sense, the second part is about existence but not uniqueness: effective race reporting at scale seemed non-obvious when we set out, but there are many other tradeoffs in terms of precision, efficiency, and simplicity that one could conceivably make and still effectively utilize the detection-and-reporting framework.

5.1 Part 1: Abstract Domain and Reporting Algorithm

RacerD seeks to report data races that can result from client code calling methods of a class C concurrently from different threads. Given a procedure summary representing a set of "access snapshots" A_m for each method m in class C, RacerD aggregates the summaries of non-`private` methods of C into a single access snapshot set A_C for the class C, such that $A_C = \bigsqcup_{m \in \mathsf{methods}(C)} A_m$, where methods($C$) yields all the non-`private` methods in a class C. The access snapshot set A_C gives us an overapproximation of all the heap accesses in (or reachable from) methods of C. The representation of a heap access is intentionally quite basic—we use a syntactic *access path* (Jones and Muchnick 1979) (a base variable followed by a sequence of fields, as defined in Figure 3) along with its source location.

Given a pair of access snapshots, in order to report a race we must be able to answer two questions:

(1) Can the accesses touch the same address?
(2) Can the accesses happen concurrently?

RacerD's abstract domain needs to record information that will allow us to answer these questions in an underapproximate fashion (*i.e.*, we want a subset of accesses that touch the same address and happen concurrently). Remarkably, access paths alone *almost* convey enough semantic information to answer (1) on their own! If two access paths are syntactically equal, it's almost (but not quite) true that they must refer to the same address. Syntactically identical paths can refer to different addresses if (a) they refer to different instances of the same class or (b) a prefix of the path is reassigned along one execution trace, but not the other. Stated differently, conditions (a) and (b) cannot hold if an access path is *stable*; *i.e.*, if none of its proper prefixes appear in assignments during a given execution trace, then it touches the same memory as all other stable accesses to the same syntactic path.

A formal treatment of access path stability, including a theorem we have proven stating sufficient conditions for reports involving stable paths to be true positives, will be given in a sequel to this work. But for the purpose of explaining why RacerD is effective in practice (the goal of this paper), it's enough for the reader to accept that syntactic equality of access paths is a reasonably effective way to answer (1) in an underapproximate fashion.

lock count	$\ell \in \mathcal{L}$	$= \mathbb{N}^{\top}$
concurrent threads	$t \in \mathcal{T}$	$= \mathsf{NoThread} \sqsubset \mathsf{AnyThreadButMain} \sqsubset \mathsf{AnyThread}$
ownership value	$o \in \mathcal{O}$	$= \mathsf{OwnedIf}(\wp(\mathbb{N})) \sqsubset \mathsf{Unowned}$
ownership environment	$E \subseteq \mathcal{E}$	$= \mathsf{Path} \to \mathcal{O}$
access snapshots	$A \subseteq \mathcal{A}$	$= \{\langle \pi, k, \ell, t, o \rangle \mid k \in \{\mathsf{rd}, \mathsf{wr}\}\}$
domain	$d \in \mathcal{D}$	$= \mathcal{L} \times \mathcal{T} \times \mathcal{E} \times \mathcal{A}$

Fig. 4. Components of the RacerD abstract domain.

Question (2) arises because two accesses to the same memory only race if they can run concurrently. To answer this question, RacerD computes mutual exclusion information to associate with each access path as part of an access snapshot. The three domains RacerD uses for this (along with the semantic questions they answer) are as follows.

- **Locks domain $\mathcal{L}$**: detects mutual exclusion due to being protected by the same lock. The essential feature of this domain is determining when a lock *must not* be held during a heap access.
- **Threads domain $\mathcal{T}$**: detects mutual exclusion due to running on the same thread. The key feature of this domain is determining whether an access *must* occur on a thread that runs concurrently with other threads.
- **Ownership domain $\mathcal{E}$**: detects mutual exclusion owing to memory that is held exclusively by one thread, and thus not shared between threads. This is similar to the way a thread escape analysis is used in optimization or in overapproximate race detectors (Choi et al. 1999; Cohen et al. 2008; Naik et al. 2006). This domain must be able to determine when two syntactically identical access paths *may* refer to different addresses when run on different threads (*i.e.*, satisfies condition 1(a) above).

The access snapshot domain $\mathcal{A}$ associates mutual exclusion information with a syntactic access. As we show in Figure 4, an individual access snapshot is represented a quintuple $\langle \pi, k, \ell, t, o \rangle$. The first two components of the tuple are the path accessed π and the kind k of the access (either a read or write). The final three components are mutual exclusion information represented as abstract values in the three previously described domains: lock count ℓ, concurrent threads t, and ownership value o. The domain itself is a set of individual access snapshots ordered by inclusion.

We will explain one way to implement these domains shortly in Section 5.2, but for now let us demonstrate the semantic questions the domains can answer to report high-confidence data races.

Reporting conflicting accesses. A potential data race between a pair of accesses can be prevented if the accesses touch different memory, if a lock is held during both accesses, if the accesses occur on threads that cannot run concurrently, or if neither of the accesses is a write. RacerD's domain allows us to answer semantic questions that rule out each of these preventative measures in turn and report a race when no measure is in place. Specifically, for each pair of access snapshots $\langle \pi_1, k_1, \ell_1, t_1, o_1 \rangle$, $\langle \pi_2, k_2, \ell_2, t_2, o_2 \rangle$ in the access snapshot set A_C (including reflexive pairings), RacerD reports a data race if all of the following conditions hold:

(1) $\pi_1 = \pi_2$ (accesses are to the same syntactic path)
(2) $o_1 \neq \mathsf{Owned} \land o_2 \neq \mathsf{Owned}$ (neither access is to thread-local memory)
(3) $\ell_1 = \mathbf{0} \lor \ell_2 = \mathbf{0}$ (at least one of the accesses is not protected by a lock)
(4) $t_1 \sqcup t_2 = \mathsf{AnyThread}$ (at least one of the accesses runs in a "background" thread distinct from the main thread)
(5) $k_1 = \mathsf{wr} \lor k_2 = \mathsf{wr}$ (at least one of the accesses is a write)

As we will explain in the next section, Owned is an abstract value whose concretization is a thread-local address. AnyThread is an abstract value whose concretization is a thread id assumed to run in parallel with other threads.

5.2 Part 2: Implementing Domains and Computing Summaries

Now that we have explained the domain structure for data race reporting Section 5.1, we will present specific domains that match this structure. We will use the simple programming language in Figure 3 to informally explain our transfer functions. This language resembles Java in ways that are relevant to race detection: it has heap allocation, reads and writes of fields via access paths, function calls, and locks. We leave the expressions and statements in the language unspecified for ease of explication, but our implementation handles the full Java language.

5.2.1 Locks Domain: Counting Locks Held. RacerD focuses on identifying accesses that are unsafe because they are not guarded by any lock rather than accesses that are unsafe because they are guarded by the wrong lock(s). It overapproximates the *number* of locks held at a given program point. This is consistent with our desire for underapproximate reporting of races; if we know that a lock may be held at the time when an access occurs, we will assume it is protected by the lock.

Abstract values ℓ in the lock domain are non-negative integers or a special $\top$ element. The ordering on integer elements is $\leq$, and all integer values are below the $\top$ element on the ordering. Widening any two abstract values produces the $\top$ value. We define a lifting of integer addition ($+^\top$) and subtraction ($-^\top$) to handle the $\top$ element in the obvious way (any value +/- $\top$ is also $\top$). Acquiring a lock via the lock command increments the abstract value by **1**, and releasing a lock via the unlock command decrements the abstract value by **1**.

Java's `synchronized` keyword is essentially syntactic sugar for calling lock at the beginning of the procedure and calling lock at the end. Thus, the initial lock count for a procedure is **1** if the procedure is marked `synchronized` and **0** otherwise. If the number of locks held at the end of a procedure is ℓ_{post}, we use $\ell_{post} -^\top \mathbf{1}$ if the procedure is marked `synchronized` and ℓ_{post} otherwise. Lexical `synchronized` blocks are handled in a similar way.

```
synchronized (lock1) {
  synchronized (lock2) {
    x.f = ...
  }
  y.g = ...
}
z.h = ...
```

The lock count is only used to determine whether a lock might be currently held during an access (*i.e.*, we only care whether it's zero or nonzero). Given this, a boolean abstraction might seem like a better choice than our counting domain. However, a boolean abstraction does not work well in the presence of nested locks. Consider the simple example to the right. A boolean domain will be unable to determine that a lock is held during the accesses to `x.f` and `y.g`, but not during the access to `z.h`. We use a counting domain instead, in order to avoid this issue; it also allows us to underapproximate Java's *reentrant* locks.

5.2.2 Threads Domain: Tracking Concurrent Threads. In order for RacerD to identify races, it needs to understand which accesses can run concurrently on separate threads. Since our goal is to report a small set of high-confidence races, we need to be fairly certain that an access can run in a concurrent context before implicating it in a race. RacerD does this using *concurrent context inference*: assuming that a code fragment cannot be run in parallel with other threads unless it sees "evidence" that the fragment is intended to be run in a concurrent context. The forms of evidence RacerD considers are:

- Lock usage. We assume that code only uses locks if it is meant to be run in a concurrent context.
- `java.util.concurrent` annotations like `@ThreadSafe` or `@GuardedBy`. Developers use these annotations to document that the annotated code is intended to be used in a concurrent context.

- Android utility functions such as `assertMainThread()` and `assertOnBackgroundThread()`. We assume these functions should only be used if there is a possibility that the code fragment can be run on a thread that is running concurrently with other threads.

Based on these forms of evidence, RacerD infers which threads may run in parallel with the thread of the current procedure. It uses three abstract values: the current procedure does not run concurrently with any other threads (NoThread), the current procedure runs on the main thread (AnyThreadButMain) and may run in parallel with code on background threads, or the current procedure runs on a thread that can interleave with any other thread (AnyThread). Let us emphasize that the concretization of these values is the set of threads that can run *in parallel* with the current thread, so (perhaps confusingly) a procedure that can only run on the main thread is AnyThreadButMain.

The default value for a procedure is NoThread. Annotating a procedure `@UiThread` or calling a utility method like `assertMainThread()` changes the abstract value to AnyThreadButMain. Annotating a procedure `@ThreadSafe`, tagging it with the **`synchronized`** keyword, or using a lock in the body of a method changes the abstract value to AnyThread. Annotating a class `@ThreadSafe` is the same as annotating all of its non-**`private`** methods as `@ThreadSafe`. Subtypes of the annotated class are treated in the same way. Thus, our interpretation of a `@ThreadSafe` class C is: "Any non-**`private`** method m of C can be safely called in parallel with all other non-**`private`** methods of C and its subclasses, including m itself." We have found that this generally fits well with the intention developers have when marking a class C as `@ThreadSafe`.

5.2.3 Ownership Domain: Identifying Clearly Safe Accesses. An individual access may be involved in a race if it can run concurrently with other accesses to the same memory. Some accesses are impossible to race with, even by (a copy of) themselves, by virtue of the fact that they touch addresses which are newly allocated and have not leaked to other threads. For example, if the method `ownedAccess()` to the right is run in parallel with itself, then we won't have a race, even though the accesses to `x.f` in both threads are syntactically identical. RacerD uses an ownership analysis to recognize such accesses. It leverages the fact that an access path π assigned to freshly allocated memory via the π := new() command is owned and aggressively propagates ownership across assignments.

```
void ownedAccess() {
  C x = new C();
  x.f = 42;
}
```

The ownership abstract domain maps access paths to ownership abstract values, which are abstractions of concrete addresses, where a map is a finite partial function. The abstract ownership states are Unowned and OwnedIf($\{n_1, \ldots, n_i\}$). An Unowned value is potentially unsafe to access; intuitively, it represents an address that may be reachable from local variables of multiple threads. OwnedIf represents ownership that is conditional on ownership of *all* of the formal parameters at the given indexes. For example, OwnedIf($\{1, 2\}$) means that the associated path is owned *if* both the first and second parameters of the current procedure are owned at the call site. OwnedIf($\emptyset$) means that the value is owned; we use Owned and OwnedIf($\emptyset$) interchangeably.

The OwnedIf values form a powerset lattice with OwnedIf($\emptyset$) as the bottom element. The Unowned value is used as a distinguished top value for the powerset lattice.

Initially, each local and global variable is mapped to Unowned, and the ith formal parameter is bound to OwnedIf($\{i\}$). In constructors, we assume that **`this`** is Owned. Ownership is propagated via the assignment command $\pi_1 := \pi_2$. Assignment to an access path propagates the ownership value bound to the right-hand side of the assignment π_2 to the left-hand side of the assignment π_1.

The current implementation of RacerD maintains an invariant: if an access path is owned, then the path and its suffixes remain owned for the duration of the current procedure. This can lead to false negatives. An example illustrating this is in the code to the right. A local object is leaked to another thread, and if the other thread accesses field `f` we will have a race. We attempted an escape analysis, paired with a more accurate treatment of ownership and an alias analysis, to detect data races like this, but we never deployed it because it caused large numbers of false positives. This is not a fundamental limitation of our approach; if a better analysis existed, it could be easily plugged into the detection framework described in the previous section. But the discussion also illustrates how our choice to focus on minimizing false positives justifies analyzer choices that both simplify the analysis and that are the opposite of ones (*e.g.*, use escape and alias analysis) that are often adopted.

```
void FN_escapeThenWriteLocalBad() {
  Obj local = new Obj();
  leakToAnotherThread(local);
  local.f = 1;
}
```

5.2.4 Access Snapshots Domain. The access domain captures a snapshot of the relevant state at the time of each heap access. As in Figure 4, an individual access snapshot is represented as a quintuple $\langle \pi, k, \ell, t, o \rangle$. The first two components are the path accessed π and the kind k of the access (either a read or write).[5] The final three components are mutual exclusion information represented as abstract values in the three previously described domains: lock count ℓ, concurrent threads t, and ownership value o. The domain itself is a set of individual access snapshots ordered by inclusion.

Any usage of an access path in $\pi_1 := \pi_2$, $\pi := \mathsf{new}()$, or $\pi :=$ call $m(\vec{e})$ will add access snapshots for each path used and all of its prefixes. For example, the code `x.f.g = y.h` would generate read access snapshots for `x.f`, and `y.h` and a write access snapshot for `x.f.g`. Each snapshot will be associated with the current mutual exclusion information tracked by the other three domains.

Our implementation supports accesses to containers and arrays as well as simpler accesses to fields. Supported containers (*e.g.*, `java.util.Map`) have models that specify which container methods perform writes and which perform reads. We omit containers from our simple language for ease of explication; our implementation models container accesses by using a dummy `contents` field.

5.2.5 Creating Procedure Summaries. A summary has the same type as an abstract state. We create a summary by running the analysis starting from an empty pre-state (described above), and installing the post-state as the procedure summary. This is done without taking any calling context information into account, and that is what makes the analysis compositional. Each procedure is visited only once by the analysis algorithm (or twice, in the case of recursive calls; we do not compute an interprocedural fixed point), and that enables it to scale.

This way of creating summaries is very simple, but having simple summaries is useless if they cannot be applied to obtain somewhat precise information at call sites. Summary application is a fundamental step for the analysis, so we will now describe it in some detail.

5.2.6 Applying Procedure Summaries. At a high level, applying a summary to an abstract state at a call site involves two steps:

(1) Updating the caller's mutual exclusion information based on the locks acquired, threading information, and ownership value returned by the callee.
(2) Updating the callee's access snapshots with the caller's mutual exclusion information, then joining the updated snapshot set with the caller's access snapshots.

Figure 5 defines an apply function for combining caller and callee state for each of four components of the abstract state. We will now walk through each one and explain how it works.

[5]We remark that we only record snapshots on paths $\pi = x.\mathrm{f}_1.\cdots.\mathrm{f}_n$ where $n \neq 0$, as accesses to variables without field selectors do not touch the heap.

$$\mathsf{apply}(\ell_{caller}, \ell_{callee}) \triangleq \ell_{caller} +^{\top} \ell_{callee} \tag{A}$$

$$\mathsf{apply}(t_{caller}, t_{callee}) \triangleq \begin{cases} \mathsf{AnyThreadButMain} & \text{if } t_{callee} = \mathsf{AnyThreadButMain} \\ t_{caller} & \text{otherwise} \end{cases} \tag{B}$$

$$E(\mathsf{e}) \triangleq \begin{cases} o & \text{if } \mathsf{e} = \pi \text{ and } (\pi, o) \in E \\ \mathsf{Unowned} & \text{if } \mathsf{e} = \pi \text{ and } \pi \notin \mathrm{dom}\,(E) \\ \mathsf{OwnedIf}(\emptyset) & \text{otherwise} \end{cases} \tag{C}$$

$$\mathsf{apply}(E_{caller}, \vec{\mathsf{e}}, o_{callee}) \triangleq \begin{cases} \mathsf{Unowned} & \text{if } o_{callee} = \mathsf{Unowned} \\ \mathsf{OwnedIf}(\emptyset) & \text{if } o_{callee} = \mathsf{OwnedIf}(\emptyset) \\ \bigsqcup_{i \in N} E_{caller}(\mathsf{e}_i) & \text{if } o_{callee} = \mathsf{OwnedIf}(N) \end{cases} \tag{D}$$

$$\mathsf{apply}(E_{caller}, \vec{\mathsf{e}}, \pi_{ret}, E_{callee}) \triangleq E_{caller}[\pi_{ret} \mapsto \mathsf{apply}(E_{caller}, E_{callee}(ret), \vec{\mathsf{e}})] \tag{E}$$

$$\mathsf{subst}(\pi, \vec{\mathsf{e}}) \triangleq \begin{cases} y.\mathsf{f}_1.\cdots.\mathsf{f}_n.\mathsf{g}_1.\cdots.\mathsf{g}_m & \text{if } \begin{cases} \pi = x.\mathsf{g}_1.\cdots.\mathsf{g}_m, \\ x \text{ is the } i\text{th formal, and} \\ \mathsf{e}_i = y.\mathsf{f}_1.\cdots.\mathsf{f}_n \end{cases} \\ \pi & \text{otherwise} \end{cases}$$

$$\mathsf{apply}(A_{caller}, \ell_{caller}, t_{caller}, E_{caller}, \vec{\mathsf{e}}, A_{callee}) \triangleq A_{caller} \sqcup \left\{ \langle \pi', k, \ell', t', o' \rangle \;\middle|\; \begin{array}{l} \pi' = \mathsf{subst}(\pi, \vec{\mathsf{e}}), \\ \ell' = \mathsf{apply}(\ell_{caller}, \ell), \\ t' = \mathsf{apply}(t_{caller}, t), \\ o' = \mathsf{apply}(E_{caller}, o, \vec{e}), \\ \langle \pi, k, \ell, t, o \rangle \in A_{callee} \end{array} \right\} \tag{F}$$

Fig. 5. Applying a callee summary at a call site $\pi_{ret} :=$ call $m(\vec{\mathsf{e}})$.

(A) Instantiating lock summaries. The $\mathsf{apply}(\ell_{caller}, \ell_{callee})$ function for the locks domain adds the number of locks held in the caller ℓ_{caller} to the number of locks held in the callee ℓ_{callee} using the lifted addition operator $+^{\top}$. We wish to overapproximate the number of locks held after the call, and the sum will achieve this. In practice, ℓ_{callee} will typically be $\mathbf{0}$ unless the callee acquires a lock without releasing it (*e.g.*, `void acquire(){ mLock.lock(); }`).

(B) Instantiating thread summaries. If a caller runs concurrently, then clearly its callees do too. However, consider the case when we have evidence that a callee can be concurrent, but don't know anything about what context is expected for the caller. Inheriting the callee's context (*i.e.*, joining the abstract values of the caller and callee) seems like a natural choice. However, we found that this introduces a large number of false positives in practice. The primary problem is this: calling code that is safe to run concurrently does not necessarily imply that the caller will run concurrently.

To see why this is true, consider the example in Figure 6. The problem captured in this example is that in practice, `@ThreadSafe` is sometimes intended to mean only "safe to use in a concurrent context" (*e.g.*, immutable classes) and sometimes intended to mean "safe **and** intended to be used only in a concurrent context" (*e.g.*, a thread-safe data structure like `ConcurrentHashMap`). RacerD does not have a reliable way of understanding which meaning is intended.

To handle this, $\mathsf{apply}(t_{caller}, t_{callee})$ uses AnyThreadButMain as the state after the call if the callee's abstract value is AnyThreadButMain and the caller's abstract value otherwise. If the callee is known to run on the main thread (*e.g.*, it calls `assertMainThread`), then the caller must also. But in any other case, the callee's context doesn't necessarily tell us anything about the caller's context.

(C) - (E) Instantiating ownership summaries. Applying an ownership summary E_{callee} to an caller ownership environment E_{caller} at a call site $\pi_{ret} :=$ call $m(\vec{\mathsf{e}})$ is defined in three parts:

```
@ThreadSafe class ImmutableData {
  private final int mData;
  public ImmutableData(int data) { this.mData = data; }
  int getData() { return mData; }
}
public void sequential(ImmutableData data) { this.mField = data.getData(); }
```

Fig. 6. Developers often annotate immutable classes like `ImmutableData` with `@ThreadSafe`. `ImmutableData` can itself be safely used in either a sequential or concurrent context, but a class that uses `ImmutableData` may or may not be run in a concurrent context. If we assume that the `sequential` method runs in a concurrent context because it uses the `getData` method of a `@ThreadSafe` class, it will lead us to (incorrectly) report a self-race on the write to `this.mData`.

```
static Builder make() { return new Builder(); } // summary: [ret |-> Owned]
Builder setX(int x) { // ...
  this.x = x;
  return this;
} // summary: [ret |-> OwnedIf({0})]
void useBuilder() {
  Builder b1 = make(); // [b1 |-> Owned]
  Builder b2 = b1.setX(1); // [b1 |-> Owned, b2 |-> Owned]
}
```

Fig. 7. Interprocedural ownership analysis is required to understand how ownership is propagated in the commonly used Java `Builder` pattern for constructing immutable objects.

(C) A utility function $E(\mathrm{e})$ to determine whether expression e is owned in ownership environment O. If e is an access path π, this function retrieves the ownership value for π in E (if present) or Unowned (if not present). If e is a non-access path expression (*e.g.*, a constant), the function returns OwnedIf($\emptyset$).

(D) The apply($E_{caller}, o_{callee}, \vec{\mathrm{e}}$) function for updating the ownership abstract value o_{callee} based on the caller's ownership environment E_{caller} and list of caller actual parameters $\vec{\mathrm{e}}$. This function leaves the callee ownership value unchanged if it is either Unowned or OwnedIf($\emptyset$). Intuitively, there's nothing the caller can do to change the ownership of a callee access that has no relationship to the actuals passed by the caller.
On the other hand, if the callee ownership value is OwnedIf(N) for some nonempty set of callee formal indexes N, the caller looks up the caller ownership value for the ith actual expression e_i associated with an index $i \in N$. The new ownership value is the join of all the corresponding caller ownership values for N. Intuitively, the ownership of an access rooted in a formal of the callee depends on the ownership status of the actual bound to that formal in the caller.
This function is also used for updating the ownership value associated with a callee access.

(E) The apply($E_{caller}, E_{callee}, \vec{\mathrm{e}}, \pi$) function for updating the caller's ownership environment E_{caller} by binding the caller return value π_{ret} to the callee return value ret from the callee ownership environment E_{callee}. This function discards the callee ownership environment except for the return value.

Figure 7 shows RacerD's interprocedural ownership analysis in action. The `make` procedure returns a freshly allocated `Builder`, which the analysis summarizes as the ownership environment `ret |-> Owned`. The `setX` procedure writes to the `x` field of its receiver `this` and then returns it. The summary expresses that the return value is owned if the receiver `this` is owned at the call site.

```
static void multiOwn(Obj o1, Obj o2) { // [o1 |-> OwnedIf({0}), o2 |-> OwnedIf({1})]
  Obj local;
  if (*)        // non-deterministic choice
    local = o1; // [local |-> OwnedIf({0})]
  else
    local = o2; // [local |-> OwnedIf({1})]
  // [local |-> OwnedIf({0, 1})]
  local.f = 7;
}  // summary: <local.f, Write, locks: 0, threads: NoThread, OwnedIf({0, 1})>
static void useMultiOwn(Object x) {
  Obj y = new Obj(); // [x |-> OwnedIf({0}), y |-> Owned], locks: 0, threads: NoThread
  synchronized (...) { // [x |-> OwnedIf({0}), y |-> Owned], locks: 1, threads: AnyThread
    multiOwn(x, y); // <local.f, Write, locks: 1, threads: AnyThread, OwnedIf({0})>
  }
}
```

Fig. 8. Updating callee accesses with the caller's locking and ownership information.

Finally, the procedure `useBuilder` calls both `make` and `setX`, which requires instantiating the summaries for each procedure. The summary for `make` is applied to caller return value `b1` using the apply($E_{caller}, E_{callee}, \vec{e}, \pi$) function, which yields a state where `b1` is owned. Then, the call to `b1.setX(1)` triggers application of the summary for `setX` to return value `b2` and caller actuals `b1` and `1`. The summary for `setX` says that the return value is owned if the actual at index 0 is owned in the caller. That is `b1`, which is indeed owned in the caller according to the ownership environment. Thus, the caller return value `b2` is also bound to Owned.

(F) Summarizing callee access snapshots. Finally, we explain how we apply a summary of accesses performed in a callee via the apply($A_{caller}, \ell_{caller}, t_{caller}, E_{caller}, \vec{e}, A_{callee}$) function. Intuitively, the caller applies its mutual exclusion information to each of the access snapshots and then joins the updated set of snapshots with its local snapshot set A_{caller}. This means that if a lock is held in the caller, RacerD will know that a lock was also held at the time of the access in the callee (apply(ℓ_{caller}, ℓ) part). Similarly, the apply($E_{caller}, o, \vec{e}$) part updates the ownership status of a callee access based on the caller actuals were bound at the call site. Also, the caller applies the subst function on each callee snapshot, translating access paths rooted at callee formals by substituting in the expression provided as argument for that formal.

The example in Figure 8 illustrates the process of updating. The `multiOwn` procedure contains an access to `local.f` whose ownership status is somewhat complicated: it is an access to owned memory if the actuals bound to formals `o1` and `o2` are both owned: OwnedIf($\{0, 1\}$). At the call site in the `useMultiOwn` procedure, the actual `y` is owned and the actual `x` is a formal of `useMultiOwn`. Using the apply function on the caller and callee states, we convert the original OwnedIf($\{0, 1\}$) value to $E_{caller}(\mathtt{x}) \sqcup E_{caller}(\mathtt{y})$ = Owned $\sqcup$ OwnedIf($\{0\}$) = OwnedIf($\{0\}$). This reflects that the access in `multiOwn` can only be involved in a race if the formal `x` is not owned at the call site of `useMultiOwn`.

In addition, a lock is held in the caller via a **`synchronized`** `block`. Updating the lock count for the callee access works in a similar way. At the `multiOwn` call site, the lock count is **1**. When we use apply on the access in the callee, we update the callee lock count to $\mathbf{1} +^{\top} \mathbf{0} = \mathbf{1}$. This reflects that a lock is held at the time of the access in `multiOwn`.

5.3 Implementation

RacerD is implemented as a specialized program analysis using the Infer.AI analysis framework.[6] The framework provides an API to Infer's backend compositional analysis infrastructure, based on

[6] http://fbinfer.com/docs/absint-framework.html

abstract interpretation (Calcagno and Distefano 2011; Calcagno et al. 2015). You give the framework an 'abstract domain' (symbolic values for the analysis to track), transfer functions that specify how program statements transform these values, and a way to make a 'summary' of a procedure, which is independent of its calling context. Out pops a program analysis that works compositionally and can (often) scale to big code.

In addition to the pure logic for race detection, RacerD implements a non-trivial amount of scaffolding related to error reporting. The basic issue is that data races are challenging to explain to developers, and RacerD takes several steps to try to make its reports easier to understand:

- *Reporting a full stack trace to each access involved in a race.* Though we didn't formalize this above, our access snapshot domain tracks the location where a syntactic access occurred. When a snapshot is propagated to a callee via summary application, the domain records the call site for the callee. This gives us enough information to produce a full call stack to any access involved in a race by repeatedly "unrolling" summaries until the location of the original access is reached.
- *De-duplicating 'similar' reports.* An important principle of effective reporting is 'don't spam the developers'. On the other hand, RacerD's race identification is effective enough to be able to identify tens or even hundreds of potential races caused by a code change; effective reporting needs to contend with this effective identification. In the beginning we sometimes reported tens of races on the same line, and also tens of reports of races on a single field from different source lines. Interprocedural errors were particularly susceptible to a multiplication of issues.
  It is not necessarily helpful for a developer to be see hundreds of reports of a similar nature and with (perhaps) related causes. To avoid spamming developers RacerD 'de-duplicates' its reports to show at most one for a given syntactic access and at most one report per pair of procedures. This can suppress true alarms in the case that a warning filtered by de-duplication is a distinct issue from the one displayed to the developer. But in this case, the developer will see the suppressed warning after fixing the displayed warning and re-running the tool. Also, if a class has more than zero reports prior to de-duping, then it will also have more than zero after.
- *Explaining which access(es) occur with a lock held or in a concurrent context.* Because RacerD's lock and thread domains track locking and threading information interprocedurally, a racy access may occur far away from a lock acquisition or other evidence that a procedure runs in a concurrent context. RacerD's error messages point out whether an access occurs inside of a lock and explain why RacerD thinks the access may run in a concurrent context.
- *Assumed safe features.* Recall above our determination to go after coarse-grained locking errors but not fine-grained. The semantics of **`volatile`** are quite complicated, so if you mark a field **`volatile`**, RacerD will assume that you know what you are doing and will not report any races. This crude heuristic has proven to be stunningly effective in practice. For instance, double-checked locking is well known to suffer from subtle flaws (Bacon et al. 2012), but we find that it is often the **`volatile`** part that folks get wrong, not the guards. In fact, RacerD has caught several significant errors related to double-checked locking when the **`volatile`** was left out.
  RacerD also supports the annotations `@Functional`, `@ThreadConfined`, `@ReturnsOwnership`, and `@VisibleForTesting`, for guiding the analyzer to understand some subtle, but correct, patterns common in Facebook code. These annotations are fully explained in RacerD's online documentation.[7]

6 EVALUATION

This section presents an evaluation of RacerD. The first part describes the use of RacerD at Facebook. In the second part, we compare RacerD against other static and dynamic race detectors.

[7] http://fbinfer.com/docs/racerd.html

We could not find other directly comparable race detectors that allow for incremental analysis of large codebases like RacerD does, so we compare using a "from-scratch" analysis setup (*i.e.*, other tools' home turf, not RacerD's).

6.1 Deployment and Impact at Facebook

At Facebook, developers primarily use RacerD for two purposes:

- Preventing regressions in safe concurrent code
- Adapting sequential code for safe use in a concurrent context

Both use-cases leverage RacerD's deployment as a bot participating in code review for Facebook, Messenger, Instagram and other apps. RacerD runs as part of the Facebook continuous integration (CI) system; the developer never needs to make the decision to run the tool. For each code change a developer submits, the CI runs RacerD alongside other jobs for compilation and testing. This deployment applies to all Android developers, whether they are thinking about concurrency or not.

The CI deployment only reports bugs that were introduced by each code change, which is ideal for the first use-case: catching regressions. If RacerD thinks a code change introduces a data race, it comments on the offending line and reports a full stack trace to each of the heap accesses involved.

For the second use-case, developers are interested in adapting code that was previously intended to run in a sequential context (*e.g.*, a Litho class that wishes to use Litho's background layout feature, as we will explain below) to be safely used in a concurrent context. By default, RacerD will not report any races on code that appears to run sequentially (see Section 5.2.2 for more details). Developers use a workflow where they send a diff adding `@ThreadSafe` annotations to the classes of interest and wait for the CI deployment to report RacerD warnings on the annotated code. The developers resolve as many warnings as they see fit to, send an update to the diff to fix the warnings, and make sure that RacerD thinks the fix is ok. Once the code is free of warnings, the diff can be landed and the code is (hopefully!) safe to use in a concurrent context.

Deployment. In order for RacerD to be deployed effectively as part of the CI system, the analysis must report issues quickly and use resources efficiently. We have found compositional analysis to be useful for meeting these requirements. With a whole-program analysis, the only obvious way to analyze a code change is to re-analyze the entire app. On a machine much more powerful than the ones used in the CI, running RacerD on the Facebook Android app takes about 130 minutes. Figure 9 highlights the benefits of running on code changes instead: RacerD can analyze much less code and run much faster. This diff-based workflow is thus much more user-friendly than the "from-scratch" analysis mode we will speak about below.

Case study: Litho background layout migration. Litho is an open-source Facebook framework for creating performant scrolling UI's,[8] which is used by the Facebook App, Instagram, Messenger, FB Lite, and Messenger Android codebases. All of those codebases have a lot of concurrent code that does not involve Litho, which we analysed using RacerD as well; however, here, we focus the discussion on Litho exclusively.

One of the important features of Litho is its ability to run as much work as possible on non-blocking background threads instead of the main UI thread. In early 2017, the Litho team had just finished implementing a new feature: the ability to perform the computationally expensive *layout* step of UI rendering (*i.e.*, sizing UI elements appropriately and determining their relative positioning). Moving this step off of the main thread had the potential to unlock a significant performance improvement.

[8]https://fblitho.com/

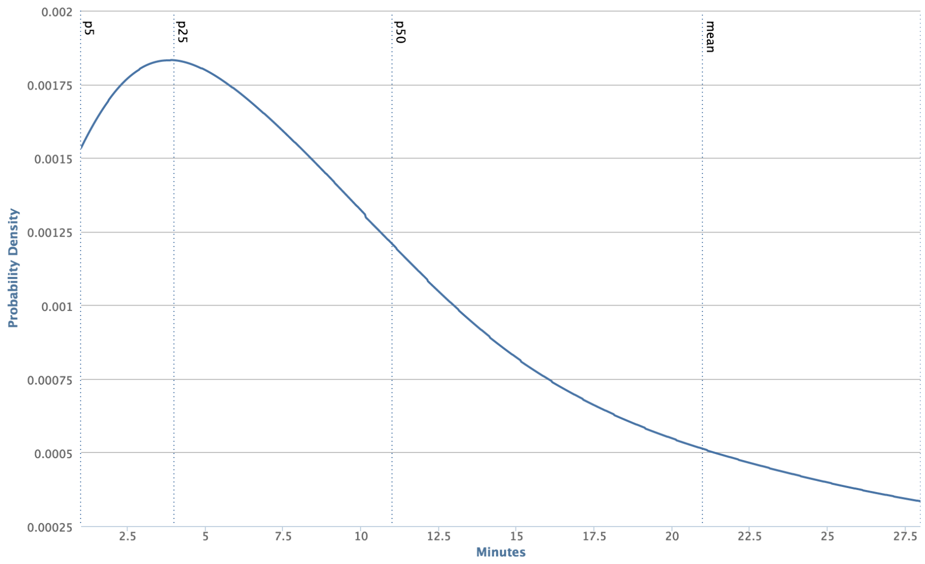

Fig. 9. Kernel density estimate of RacerD running times on code changes in a typical week. This time includes building the code, running RacerD and several other analyzers, and reporting errors. The plot shows that more than half of the code changes are analyzed in less than 12 minutes. This is an order of magnitude faster than running a from-scratch analysis, and it is also much less resource-intensive (which is very important for being a good citizen of the CI). We note that the time taken to actually run the RacerD analysis is a tiny fraction of the time in this plot. As we will show in Table 1, RacerD can analyze tens of thousands of lines a second. The vast majority of the diff analysis time is consumed by the supporting steps.

But there was a problem: the Facebook app already had hundreds of Litho-based classes running in production (totaling many millions of lines including dependencies), and none of the layout steps had been written with concurrency in mind. Enabling background layout is as simple as flipping a flag, but blindly enabling multithreading for previously-sequential chunks of code is doomed to cause data races. Manual inspection of the code to be migrated isn't tractable because of its sheer size: races can occur anywhere in a class itself or in its massive tree of transitive dependencies.

Fortunately, the automated bug detection of RacerD empowered a very small group of concurrency experts to migrate all of the classes in a few months and safely enable background layout in production.[9] The experts used the `@ThreadSafe`-annotation based CI workflow described above to gradually adapt Litho classes to use the background layout feature. The CI deployment also prevented product engineers who weren't aware of the migration from accidentally introducing new concurrency errors in an already-migrated component.

Impact. From the safety standpoint, the migration was a success. All told, hundreds of Litho classes have been checked by RacerD during the migration, and over 2,500 analyzer issues detected by the two CI-based workflows described above have been addressed in the past 6 months (the majority were regressions). The issues addressed are split between non-functional changes where annotations are added that document threading assumptions, and functional changes that avoid races, with each category having seen fixes numbering in the hundreds.

The migration was also a success from a performance standpoint. Switching to background layout improved scroll performance in the Facebook for Android News Feed by more than 5%.

[9] https://code.facebook.com/posts/1985913448333055/multithreaded-rendering-on-android-with-litho-and-infer/

News Feed is a core product component that has been highly optimized over many years, so a performance improvement at this level is noteworthy.

False positives, false negatives, and fix rate. We closely monitor the *(immediate) fix rate* of RacerD: the percentage of issues reported as introduced in a diff that are fixed before committing the same diff. The fix rate over the last six months was 50%. It's important to keep in mind that fix rate is not the same as false positive rate.[10] False positives are only one of many legitimate reasons why a developer may choose not to fix an issue in the same code change. To name a few others that we commonly see: the issue may be fixed in a follow-up diff (or series of diffs), the diff may rename a class/file/method/field in a way that makes RacerD think a pre-existing issue is a new one, the root cause of the surfaced issue might be outside of the code being changed, and so on. One example we have often seen is that of 'distant' bugs, where a data race is flagged in class A which is changed by a developer, but the root cause is in class B which is called by the developer but owned by someone else. Typically, class A is product code and class B is framework code; the product engineer is then not well advised to change the class B, and the issue should not be addressed in the same diff.

Because there are several reasons for not immediately fixing a warning within the same diff, we posit that even a fully precise diff analysis that reports no false positives will still pay a fix rate "tax". Our best estimate on how large this tax is comes from the fix rate of another analysis we have that *is* fully precise. We have an analysis that flags dead stores of variables in C++. This check is implemented using a basic liveness analysis and is fully precise modulo bugs in the analyzer. Even though this analyzer reports no false positives in theory, its fix rate over the last six months is only 70%. Thus, we think that the fix rate tax a diff analysis must pay is somewhere around at least 30%. If we speculatively subtract this tax from the RacerD fix rate, we can guess that the RacerD false positive rate is around 20%. In fact, it is likely even lower: we expect that RacerD's tax is higher than that of dead stores owing to more "distant" bugs being surfaced.

Note that these comments about fix rate tax have to do with Facebook's particular deployment and code review model. The tax rate might vary from one organization to another.

We have monitored the ignored reports over the past year, and we look at a sample from time to time. Our observations from monitoring are consistent with the impression above that the false positive rate is around 20%. We say "around" because we are cautious in making definitive claims about false positive rate. Not infrequently, neither a programmer nor a diff reviewer can determine if a report is a true positive even after hours of inspection; a judgment call is made before such a determination is reached. In a large-scale deployment the false positive rate is, for all practical purposes, non-measurable and hence un-knowable. What we find useful is to measure and track the fix rate, and to observe and respond to reported false positives that arise; knowing *specific* false positives is useful, without leading us to claim that we know precisely what the *general* rate is.

On the flip side, RacerD isn't designed to catch all possible races, which raises questions about how often it has missed bugs. As it turns out, fixing all RacerD warnings has been an effective proxy for preventing races in practice thus far. Only a small number of races slipped by RacerD into production during the Litho background migration – we are aware of only three – and these were due to bugs in the tool (later fixed) rather than more fundamental issues related to our design goals. Our experience with false negatives on diffs has been similar; to date, we have only received one report of a regression that RacerD missed (a double-checked locking bug). But it turned out that RacerD had in fact reported the bug on the diff in question; the warning was just misunderstood. Therefore, to date there have been *no confirmed false negatives* reported from production over hundreds of thousands of lines of code, except for those from implementation errors on our part.

[10] See the paper by Sadowski et al. (2018) and the presentation at https://www.youtube.com/watch?v=xc72SYVU2QY.

The implementation errors have the interesting side effect of confirming that RacerD *does* find bugs that could occur in production.

In making these remarks we emphasize that "no confirmed false negatives" from production does not say that no false negatives have occurred; they might have occurred by not been noticed. Nevertheless, that no false negatives were not observed took us completely by surprise. It is possible to generate hypotheses on why this is so, but there is little built-up engineering experience we are aware of on false negative rates from production in static analysis as a whole; we have been wrong so often in making hypotheses about what works in production that we are hesitant to speculate. Rather, we would say generally that deeper understanding of false negatives is a worthwhile problem for both academics and industrialists.

Summary. We summarize our findings with the following two claims.

(1) Static race detection can be effectively applied and provide positive impact as part of an industrial software development process.
(2) Due to its speed, scale and incrementality, RacerD can be deployed at diff time on large code.

6.2 Comparison to Other Race Detectors

RacerD is a static analysis tool, but it is philosophically closer to dynamic race detectors in its goal of reporting high-confidence races with very few false positives. Thus, we have chosen to compare RacerD to both static and dynamic race detectors for Java.

6.2.1 Static Race Detectors. There is a surprising lack of publicly available race detectors for Java. A query[11] about open source static race detectors for Java or Android in the public "Abstract Interpretation" Facebook group (383 members) did not receive any responses. The only publicly available tool we could find was Chord, a pioneering static race detector for Java by Naik et al. (2006). The techniques behind Chord are intended to be sound for proving the absence of races, *i.e.*, providing an over-approximation, except for specific language features (*e.g.*, Java reflection) that are not accounted for. Chord is a whole-program analysis, and it relies on a suite of auxiliary static analyses, such as points-to and escape analysis,. Chord works for plain Java (once the user has specified entry points), but not Android, so we conducted our comparison using plain Java.

Experimental setup. The original Chord implementation dates back to 2006 and has not been maintained, so is difficult to run on current Java projects. However, the recent work on the Bingo tool by Raghothaman et al. (2018) has included a compilable version of the Chord algorithm, as well as a harness for running it on specific projects, as a part of a benchmark suite to demonstrate the use of Bayesian inference to improve static analysis results. The authors of Bingo have kindly shared their Chord-related artifacts with us and helped us carry out a comparison.

We used the benchmarks that were used to evaluate Chord in the context of the Bingo work (Raghothaman et al. 2018). The chosen programs are all concurrent applications from different domains. hedc is a web-crawler, ftp is Apache web server, weblech is a tool for downloading web pages, jspider is a web spider engine. The last four programs are taken from the DaCapo suite (Blackburn et al. 2006): avrora is an AVR microcontroller simulator, luindex is a tool for document indexing, sunflow is an image rendering system, and xalan is a tool for transforming XML to HTML. All measurements were performed on a 24-core 2.5 GHz Intel x64 machine with 55 GB RAM running Linux. Running times were averaged for multiple runs of each of the analyses.

Summary of results. Table 1 shows the results of our comparison. *Alarm Overlap* is the number of source lines of races reported by both RacerD and Chord (after de-duplication of reports, which

[11] https://www.facebook.com/groups/abstract.interpretation/permalink/10155965198993945/

Table 1. RacerD reports >3X fewer alarms than Chord on every benchmark, and an order of magnitude fewer alarms on four of the benchmarks. RacerD is at least one or two orders of magnitude faster on all benchmarks. The "Alarm Overlap" column indicates the percentage of racy lines implicated by Chord (in source files) that *also* appeared in a report from RacerD.

Program	# Files	# LOC	Chord		RacerD		Alarm
			# Alarms	Runtime	# Alarms	Runtime	Overlap
hedc	133	11,767	152	4m 47s	49	1.5s	11%
ftp	140	12,050	522	5m 14s	39	1.5s	20%
weblech	12	1,309	30	11m 02s	9	0.6s	63%
jspider	214	7,413	257	1m 54s	13	1.4s	10%
avrora	470	68,864	966	8m 49s	81	7s	18%
luindex	331	36,151	940	3m 26s	183	5s	64%
sunflow	170	21,960	958	42m 44s	43	2.8s	49%
xalan	975	175,784	1870	1h 47m	421	21.5s	38%

Chord does not do). The key takeaway from the table is that RacerD reports at least 3 times less frequently and an order of magnitude faster than Chord in all cases. We also note that:

(1) Chord and RacerD have comparable precision on the benchmark results that we manually examined. Furthermore, RacerD finds all of the true bugs found by Chord on these projects except for a handful of benign races on jspider (see next section);
(2) Chord and RacerD have a high intersection of alarms by historical standards in static analysis;[12]
(3) RacerD has better cost-to-value ratio of alarms/minute;
(4) Chord could not be deployed at diff time on large code bases as RacerD as at Facebook because it is not incremental;
(5) Chord would also be difficult to run in nightly mode (*i.e.*, the full analysis of very large codebases from scratch, run by CI agents over night) because it has limited scalability.

Given that Chord uses more sophisticated techniques such as alias and escape analysis, we were surprised that RacerD compared as well as it did on smaller projects, particularly in the intersection of issues detected. The metric of alarms/minute is an important one when judging whether to deploy an analysis industrially, *e.g.*, informing considerations related to CI capacity budget.

Manual triaging. Table 1 shows that RacerD reports much less frequently than Chord, but does reporting less means missing true bugs instead of hiding false positives? To answer this question and gain a better understanding of the relation between Chord and RacerD in terms of precision, we manually examined results from the weblech and jspider benchmarks.

In weblech, Chord reports 30 potential concurrency issues, but only 6 are true bugs. All races reported by Chord occur either in the `weblech.spider.Spider` class (21 alarms) or in utility classes from the standard `org.apache.log4j` package. Because the latter class is a library JAR that does not include sources, RacerD does not report any races in it. RacerD does report 9 potential issues in the `Spider` class, including the 6 true bugs reported by Chord. All reported races involve concurrent accesses to the `Spider` object itself. The remaining 3 issues are true positives, all of which are missed by Chord. These races involve the `urlsDownloading` and `queue` fields, modified in a **`synchronized`** section as well as without synchronization. Here is an example of racy code involving the `queue` field of the `Spider` class:

```
public void readCheckpoint() { queue = (DownloadQueue) ois.readObject(); }
```

[12]See, *e.g.*, Figure 3 of the paper by Pendergrass et al. (2013).

```
public void run() {
  synchronized(queue) { queue.queueURLs(u2dsToQueue); downloadsInProgress--; }
}
```

We also examined the results on jspider benchmark. Both tools reported the same races (modulo the differences in reporting strategy explained below) in the `WorkerThread`, `DispatcherThread` and `SchedulerImpl` classes. RacerD also reported a confirmed race in `RuleFactory` that was missed by Chord. Chord raised a large number of alarms for (seemingly benign) races in the `ResourceInternal` class. These races are not flagged by RacerD because it does not have a container model for this class (see 5.2.4). Finally, Chord mistakenly reports a non-race in the class `DistributedLoadThrottleImpl`. All races reported by RacerD for weblech and jspider were (manually) confirmed to be true positives.

The Bingo paper reports numbers of true positives in the other benchmarks as well. We did not conduct a manual inspection of these other cases due to time constraints. But in both weblech and jspider, RacerD found all of the true positives that Chord did.

Race detection and reporting in Chord and RacerD. We comment briefly on some important differences between the reporting strategies in RacerD and Chord that we noticed during manual triaging. In the reports produced by the Bingo version of Chord, every detected race is reported as a potential read/write conflict between accesses occurring at certain lines of a certain file. A race between two read/write instructions in a class `org.project.MyClass`, on lines 85 and 32 will be reported as:

```
5667:org/project/MyClass.java:85  5670: org/project/MyClass.java:32 ...
```

If reported accesses occur on the same line, this reporting scheme can make it difficult to determine the actual fields that are involved in a race. For example, one race reported by both Chord and RacerD in weblech references line 166 with the statement:

```
while ((queueSize() > 0 || downloadsInProgress > 0) && quit == false)
```

Because Chord provides no fields names, it is impossible to tell whether the reported race implicates the `queueSize`, `downloadsInProgress`, or `quit` field. The corresponding RacerD report provides both the field name and line number, which makes the issue much easier to understand:

```
Read/Write race. Non-private method `weblech.spider.Spider.run` reads without synchronization
from `weblech.spider.Spider.quit`. Potentially races with write in method `Spider.stop()`.
  165.        System.err.println("queueSize = "+queueSize());
  166. >      while((queueSize() > 0 || downloadsInProgress > 0) && quit == false)
```

In addition, the Chord approach can make it difficult to understand the high-level cause of a race. Imagine that a **`private`** method `f()` allows for an unsynchronized concurrent read/write into a field `x` of a class. Imagine further that this method can be called from two *public* methods of the same class, `g()` and `h()`. The *cause* of the actual conflict can be better understood by pointing out that it is dangerous to call `g()` and `h()` concurrently. Pointing directly at the problematic code location within `f()` isn't always helpful, particularly if the race involves a long call chain that bottoms out in the call to `f()`.

As explained in Section 5.3, RacerD attempts to mitigate these issues by using techniques such as reporting a full call stack to conflicting accesses and grouping races with the same high-level cause (*i.e.*, the **`public`** method whose invocation triggers a racy access).

6.2.2 Dynamic Race Detectors for Android. Because RacerD is primarily used to analyze Facebook's Android code, we sought to compare it with two state-of-the-art dynamic race detectors for Android: DroidRacer by Maiya et al. (2014) and EventRacer by Bielik et al. (2015). DroidRacer and EventRacer automatically attempt to discover inputs leading to races rather than asking

Table 2. RacerD reports more true bugs than DroidRacer. The "# Alarms" column gives the number of reports for each tool and the number of true positives in parentheses. We give a lower bound for RacerD's true positives on K9Mail because we only triaged a sample of the RacerD results. We manually added `@ThreadSafe` annotations to SGTPuzzles; see explanation below.

Program	# Files	# LOC	# Alarms (true bugs)		RacerD
			DroidRacer	RacerD	Runtime
SGTPuzzles	33	9,459	11 (10)	18 (18)	12s
OpenSudoku	62	9,021	1 (0)	14 (14)	14s
K-9 mail	3303	78,503	2 (1)	185 (>3)	2m 33s

for the user to provide them, which is analogous to the push-button nature of RacerD. We also considered comparing to the leading Java race detector FastTrack (Flanagan and Freund 2009), but its usage model differs significantly from RacerD, DroidRacer, and EventRacer in that it requires inputs to be manually provided.

DroidRacer and EventRacer both focus mostly on *event races*, races that take place due to non-deterministic scheduling of events (such as read/write operations) in Android the runtime. DroidRacer's execution model combines multi-threading and asynchronous event-based dispatch, which allows it to detect true concurrent data races that manifest during multi-threaded executions. Because RacerD does not detect event races, we only considered data races detected by the tools.

We ran both EventRacer and RacerD on the Android Connectbot and K-9 mail applications. EventRacer is not designed for multi-threading races, and, thus, any concurrency bugs reported it might report are incidental to event races. Unsurprisingly, EventRacer did not report any data races on the benchmarks, while RacerD reported over 100 on each. We manually confirmed that several of the RacerD reports were true positives and then halted the comparison.

DroidRacer works only with a specific Android version (4.0.1), and building it requires a particular machine configuration that we could not obtain in a timely fashion. However, with the help of the authors we managed to analyze the raw output from the corresponding conference paper (Maiya et al. 2014). We ran RacerD on the same versions of several of the larger projects that were used in evaluating DroidRacer (except for OpenSudoku; see below). All reported runtimes for RacerD were obtained on a 3.1 GHz Intel Core i7 Macbook Pro with 16 GB of RAM.

Table 2 summarizes the results of the comparison. We comment briefly on each benchmark:

- SGTPuzzles.[13] DroidRacer detected 11 data races, including 10 true positives in the `SmallKeyboard` and `SGTPuzzles` classes. However, we could not identify the fields involved in the races reported by DroidRacer, because its output does not contain the names of non-static fields in conflict, only their offsets. RacerD failed to find races at first, so we manually annotated containing key classes such as `SmallKeyboard` and `SGTPuzzles` with `@ThreadSafe`. RacerD reported 18 races on the annotated code, and all of them were true positives.
- OpenSudoku.[14] Unfortunately, we could not compile the version of OpenSudoku that was originally used to evaluate DroidRacer. Instead, we analyzed the latest available and hoped for an overlap in the reports. In the original version, DroidRacer reported one data race that was ruled a false positive. In the newer version, RacerD reported 14 concurrency errors in the `Cell` and `CellCollection` classes All of these reports are confirmed true positives.
- K-9 mail.[15] DroidRacer reported two races, including one true positive. The false positive involves the private static `pushingRequested` field of the `MailService` class. The read/write accesses implicated in the report both run on the UI thread, and thus cannot occur concurrently.

[13]Version 9561.1, https://github.com/chrisboyle/sgtpuzzles/releases/tag/9561.1
[14]Version 2.4.2, https://github.com/ogarcia/opensudoku/releases/tag/2.4.2
[15]Version 4.803, https://github.com/k9mail/k-9/releases/tag/4.803

RacerD reports 185 concurrency issues, including at least 3 true positives (we only triaged a small sample of the reports). It does not report the false positive on the `pushingRequested` field because RacerD's concurrent context inference does not observe evidence that the accesses to this field can occur on a background thread (see Section 5.2.2).

Observations on dynamic executions and missed races. In our comparison with DroidRacer, RacerD reports nearly twice as many legitimate data races in all cases. One obvious explanation for this is the fact that DroidRacer is a *dynamic* analysis tool that explores a subset of concrete executions, whereas RacerD is a static analysis tool that explores an approximation of all possible concrete behaviors. Specifically, dynamic tools might not report as many races (including possible false positives), because they could not "get past" certain functionality (*e.g.*, entering a password in a mail client), and thus would not have a chance to dynamically exercise *most of the code*. For example, in K-9 mail RacerD reported four legitimate thread safety violations in the classes `Pop3Store` and `EmailProviderCache`, but none of these races are present in the traces generated by DroidRacer, which might be because the latter has simply not reached that code.

On the other hand, DroidRacer's random generation of input effects and dynamic construction of happens-before orderings can (in principle) detect races that RacerD might miss if its concurrent context inference underapproximates the concrete concurrent behavior of the program.

The key takeaway from our comparison is that RacerD reports more true bugs than the leading dynamic race detectors (EventRacer and DroidRacer) for Android by a considerable margin. Although an apples-to-apples comparison of static and dynamic analyses is difficult, one other relevant metric for bugfinding tools is the alarms found per minute of CPU time (as we previously discussed in relation to Chord). We do not have enough detailed data on the performance of DroidRacer to compute its alarms/minute, but the DroidRacer paper said that it "took a few seconds to a few hours to analyze traces" Because RacerD finishes in less than three minutes in all cases and we have looked at the largest applications that they considered, it seems likely that RacerD performs better by this metric as well.

We might summarize the overall results in this section by saying that there is some evidence that RacerD represents a new sweet spot in the race detection design space: it reports fewer false positives than existing static tools, and it reports more true bugs than existing dynamic tools. In saying this, we acknowledge that our evaluation was hampered by both the lack of available tools for comparison and the difficulties we encountered in running them on comparable code. We therefore do not argue this conclusion too strongly, but we have sought to provide what evidence we could given these practical obstacles.

7 CONTEXT AND SELECTED RELATED WORK

Reasoning about concurrency has attracted much attention from researchers. The rapid growth in the number of interleavings is a key problem for tools that attempt exhaustive exploration. There has been important work which uses various techniques to attempt to reduce the number of interleavings while still in principle covering all possibilities (Flanagan and Godefroid 2005), but scale is still a challenge. Note that RacerD is not exhaustive: it has false negatives (missed bugs). But to compensate it is fast and effective: it finds bugs in practice and sees them fixed.

One reaction to the technical challenge of reasoning about concurrency is to ask the programmer to do more work to help the analyzer. Numerous full-blown verifiers have been developed in the research sector for proving strong properties of concurrent programs, including fine-grained synchronization. Leading examples include the semi-automatic tools VCC (Cohen et al. 2009), VeriFast (Jacobs et al. 2011), Viper (Müller et al. 2016), Caper (Dinsdale-Young et al. 2017), and Starling (Windsor et al. 2017), as well as interactive verifiers FCSL (Sergey et al. 2015) and Iris

Proof Mode (Krebbers et al. 2017) (the latter two based on Coq Proof Assistant). These tools can show a lot more about a program than can RacerD, but they are also more costly to use. Because a verification expert is typically needed to drive such tools, they are not widely used.

An intermediate approach is to ask the programmer for help, but require only relatively simple annotations that make the analyzer's life easier. This is the idea behind `@GuardedBy` annotations in Java (Goetz et al. 2006) (checked by ErrorProne[16] and other analyzers), `GUARDED_BY` and similar annotations in Clang (checked by Clang's Thread Safety Analyzer (Hutchins et al. 2014)), and `Mutex` (Turon 2015) locks on Rust (checked by the compiler). These tools aim to be (and can be) driven by engineers rather than dedicated verification experts. Facebook's Infer analyzer also includes a `@GuardedBy` checker, which can find some bugs that RacerD does not (namely, races related to using the wrong lock), but RacerD finds a greater number because it can work on un-annotated code. In the past year `@GuardedBy` checking has seen 382 fixes in Facebook's Android codebase, where RacerD's data race reports have seen over 2500.

RacerD has significantly lower startup time and ongoing friction for the developer than methods based on lock annotations do. A key finding of ours, then, is that is possible to have an effective race analysis without decreeing that such annotations must be present in order to provide signal. This was important for our deployment to News Feed, since manually inserting lock annotations would have been much less efficient for the task of converting many thousands of lines of code to a concurrent context. We believe that this finding should be transportable to new language designs.

RacerD is in a category of automatic data race detection tools that aim for lower friction than tools based on lock annotations or program specifications. Such tools allow a programmer to identify races without asking a human to describe the relationships between locks and fields. Automatic race detection has seen significant work in the research community. An early static race detector, RacerX by Engler and Ashcraft (2003), has been influential. It computed an approximation of the locks held at each program point, and it employed numerous heuristics designed to filter out likely false alarms. While it scaled to large code bases, it did not find many races; *e.g.*, it reported only 13 potential races when applied to Linux 2.5.62 (including three confirmed and fixed bugs, two suspected bugs, two benign races, and six false alarms). With the caveat that it applies to different code (Java and not C), RacerD has demonstrated a considerably better bugs/minute ratio, a key quantity in deciding whether to deploy. The Chord tool that we compared to in Section 6 is one of the most advanced approaches in this line of work. As we saw there, RacerD has considerable advantages concerning speed, while being competitive in terms of precision.

The industrial static analysis tool ThreadSafe by Contemplate targets proving thread safety of Java classes, but limits the amount of inter-procedural reasoning: "This analysis is interprocedural, but to keep the overall analysis scalable, only calls to **`private`** and **`protected`** methods on the same class are followed" (Atkey and Sannella 2015). In contrast, RacerD does deep cross-file and cross-class inter-procedural reasoning, yet still scales. The inter-class capability was one of the first requests from Facebook engineers. We inspected 100 recent data race fixes in our deployment at Facebook, and observed that 53 of them were inter-file (and thus involve multiple classes). Robert Atkey explained to us that although the ThreadSafe analysis itself does not do cross-file inter-procedural reasoning, at reporting time summaries from multiple classes can be consulted when deciding whether there is a potential race. Thus, we are unsure how it would compare to RacerD on precision. ThreadSafe is not implemented in an incremental way, so would be challenging to deploy at diff time, but it could likely be modified to be incremental.

Also after the work in this paper was completed we became aware of D4, which provides a fast differential analysis of data races (Liu and Huang 2018). The research on D4 was apparently done

[16] https://github.com/google/error-prone

virtually in parallel with that on RacerD. They do not demonstrate precision or effectiveness, but preserve the precision of an underlying analysis. Their analysis is very fast, targeting IDE time and not only diff time (*i.e.*, a few seconds instead of minutes). As mentioned earlier, our results in Table 1 suggest that RacerD's underlying algorithm is also fast enough for potential deployment in an IDE. In any case, perhaps ideas like in D4 and RacerD could complement one another in the future, for example to find bugs with high signal and then to confirm that altered code is correct.

The company Juliasoft provides a race detector, which infers information similar to `@GuardedBy` annotations as part of its proprietary offering (Spoto 2016). It is apparently a non-compositional, whole program analysis (Ferrara 2013) and, as such, would likely be challenging to deploy at diff time on a fast-changing, large codebase. The tool suite by Coverity also includes a proprietary data race detector which, according to invited POPL talk by Chou (2014), uses global statistical data. RacerD is open source.[17]

A reaction to challenges in static race detection has been to instead develop dynamic analyses. Google's ThreadSanitizer is a widely used and mature tool in this area, which has been used in production to find many bugs in C-family languages (Serebryany and Iskhodzhanov 2009). The ThreadSanitizer authors explicitly call out limitations with static race analyzers as part of their motivation: "It seems unlikely that static detectors will work effectively in our environment: Google's code is large and complex enough that it would be expensive to add the annotations required by a typical static detector".

We have worked to limit the annotations that RacerD needs for reasons similar to those expressed by the ThreadSanitizer authors. And we have sought to bring the complementary benefits of static analysis—possibility of computationally cheaper analysis with fast reporting, and ability to analyze code before it is placed in context—to industrial race detection. We were surprised to find in Section 6 that RacerD compared favorably state-of-the-art Android dynamic tools on their whole-program home turf.

What is perhaps even more surprising is how we have obtained useful results in practice using such simple techniques. As mentioned before, we have used neither alias or escape analysis techniques from static analysis, nor advanced techniques from dynamic tools such as happens-before reasoning. Instead, we have favored the less precise but fast methods based on tracking locks (see the paper by Flanagan and Freund (2009) for discussion). In developing RacerD we have rigorously applied occam's razor, adding techniques when dictated by examples from practice and not from artificial examples that we could easily generate. Keeping the design as simple as possible was a practical necessity. For instance, the ownership domain evolved in a feedback loop with engineers. Having a complicated analysis would have made this evolution more difficult to track.

RacerD represents a new sweet spot in the race detection design space: it reports less often than a state-of-the-art static tool, and it reports more often than state-of-the-art dynamic tools (where "reports" include both false and true positives). Our evaluation has shown that it could be difficult to replicate the impact of RacerD at Facebook using existing tools.

On the other hand, static and dynamic tools are often complementary. At Facebook we routinely run both static and dynamic tools that search for the same kind of bug; for instance, Infer and the dynamic tool Sapienz (Mao et al. 2016) both search for NPEs in our Android apps. In the future, we would be very interested in deploying an effective dynamic race detector alongside RacerD.

8 LIMITATIONS

There are a number of known limitations of RacerD's design:

- It looks for races involving syntactically identical access paths, but misses races due to aliasing.

[17]It is available as a part of Facebook Infer framework https://github.com/facebook/infer.

- It misses races that arise from a locally declared object escaping its scope (see Section 5.2.3).
- It abstracts lock counts rather than lock identities, and thus misses races where two accesses are mistakenly protected by different locks.
- It works for mutex locks, but will not catch races due to misuse of subtler forms of locking such as read/write locks.
- It assumes a "deep" ownership model (Clarke and Drossopoulou 2002), which misses races where local objects refer to or contain non-owned objects.
- It avoids reasoning about weak memory, Java's **`volatile`**s, and other fine-grained concurrency.

It is likely possible to make progress on a number of these limitations, though it would be essential to do so without compromising speed and effective signal[18]. But, perhaps the most surprising part of this work is that, even with these limitations, it is possible to have an analyzer that demonstrably helps engineers write concurrent code. Most of these limitations are consistent with the design goal of reducing false positives, even if they lead to false negatives. They also allow technical tradeoffs which are different than if we were to favor reduction of false negatives over false positives.

RacerD is limited to Java, but would be very worthwhile to have effective static race detectors in other languages, *e.g.*, C++, where much important concurrent code is written.

Finally, RacerD focuses on reducing false positives over false negatives, but the goal of proving thread safety at scale remains a worthwhile one. More precisely: an open problem is whether one can have an over-approximating analyzer that proves absence of races, but is similar to RacerD in its low friction, speed, scale and demonstrably effective signal.

9 CONCLUSIONS

RacerD demonstrates that a static concurrency analysis can be developed and effectively applied at the speed and scale demanded by Facebook's development model, where a large codebase is undergoing frequent modification by many programmers. It has enabled a project that converted Facebook's Android News Feed to a multi-threaded architecture, which in turn moved metrics related to performance. RacerD can scale to a codebase in the millions of lines of code, and it can deliver incremental results quickly on code changes, allowing it to participate as a bot during the code review process. The analysis has a high degree of automation, where the relationship between threads, locks and memory is discovered rather than specified by a person; this means that it can immediately scale to many programmers with low friction. Without these characteristics—speed, scale and low friction—using RacerD to support the conversion of the News Feed to a multi-threaded architecture would not have been possible.

Concurrency in general remains a fundamental and difficult problem area, but these results make us hopeful: with sufficient innovation, more techniques and tools might be developed to help working programmers in the challenging task of engineering concurrent systems.

ACKNOWLEDGMENTS

We wish to thank Martino Luca for his help with scripting for mining RacerD traces. We are very grateful to Sulekha Kulkarni, Mayur Naik, and Kihong Heo for providing us with the Bingo benchmark suite and for their assistance in running Chord. We thank Pallavi Maiya for her help with DroidRacer benchmarks, and Pavol Bielik for answering our questions about EventRacer. We benefited a lot from discussions with Mehdi Bouaziz and Francesco Logozzo, whose comments helped to improve the paper. We are grateful to Don Stewart for his encouragement and support. Finally, we wish to thank the anonymous referees for their many helpful suggestions.

[18]See (Harman and O'Hearn 2018) for further discussion on problems related to friction, signal, etc in program analysis.

REFERENCES

Robert Atkey and Donald Sannella. 2015. ThreadSafe: Static Analysis for Java Concurrency. *ECEASST* 72 (2015).

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29.

David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gün Sirer. 2012. The "Double-Checked Locking is Broken" Declaration. Available online: https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html. (July 2012).

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable race detection for Android applications. In *OOPSLA*. ACM, 332–348.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*. ACM, 169–190.

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65.

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 459–465.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (LNCS)*, Vol. 9058. Springer, 3–11.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66.

Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*. ACM, 1–19.

Andy Chou. 2014. From the Trenches: Static Analysis in Industry. (2014). Invited keynote talk at POPL'14. Available at https://popl.mpi-sws.org/2014/andy.pdf.

David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*. ACM, 292–310.

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs (LNCS)*, Vol. 5674. Springer, 23–42.

Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2008. A Practical Verification Methodology for Concurrent Programs. (2008).

Coq Development Team. 2018. *The Coq Proof Assistant Reference Manual - Version 8.8.*

Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP (LNCS)*, Vol. 10201. Springer, 420–447.

Dawson R. Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*. ACM, 237–252.

Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. 2016. Locking discipline inference and checking. In *ICSE*. IEEE/ACM, 1133–1144.

Pietro Ferrara. 2013. A generic static analyzer for multithreaded Java programs. *Softw., Pract. Exper.* 43, 6 (2013), 663–684.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *PLDI*. ACM, 110–121.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice.* Addison-Wesley.

Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Open Problems and Challenges in Static and Dynamic Program Analysis for Testing and Verification (keynote paper). In *International Working Conference on Source Code Analysis and Manipulation.*

DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. 2014. C/C++ Thread Safety Analysis. In *SCAM*. IEEE, 41–46.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS)*, Vol. 6617. Springer, 41–55.

Neil D. Jones and Steven S. Muchnick. 1979. Flow Analysis and Optimization of LISP-like Structures. In *POPL*. ACM, 244–256.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM, 205–217.

Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *PLDI*. ACM, 359–373.

Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *PLDI*. ACM, 316–325.

Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. ACM, 94–105.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*, Vol. 9583. Springer, 41–62.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. ACM, 308–319.

Peter W. O'Hearn. 2018a. Continuous Reasoning: Scaling the impact of formal methods. In *LICS*. IEEE, 13–25.

Peter W. O'Hearn. 2018b. Experience developing and deploying concurrency analysis at Facebook. In *SAS (LNCS)*, Vol. 11002. Springer, 56–70.

J. Aaron Pendergrass, Susan C. Lee, and C. Durward McDonell. 2013. Theory and Practice of Mechanized Software Analysis. In *Johns Hopkins APL Technical Digest, Volume 32, Number 2*. 499–508.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. Interactive Program Reasoning using Bayesian Inference. In *PLDI*. ACM, 722–735.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. *Proceedings of the Workshop on Binary Instrumentation and Applications*, 62–71.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*. ACM, 77–87.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

Fausto Spoto. 2016. The Julia Static Analyzer for Java. In *SAS (LNCS)*, Vol. 9837. Springer, 39–57.

Aaron Turon. 2015. Fearless Concurrency with Rust. (10 April 2015). The Rust Programming Language Blog, available at https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *CAV (I) (LNCS)*, Vol. 10426. Springer, 544–569.