Experience developing and deploying concurrency analysis at Facebook

Peter O'Hearn

Facebook

Abstract. This paper tells the story of the development of RacerD, a static program analysis for detecting data races that is in production at Facebook. The technical details of RacerD are described in a separate paper; we concentrate here on how the project unfolded from a human point of view. The paper describes, in this specific case, the benefits of feedback between science and engineering, the tension encountered between principle and compromise, and how being flexible and adaptable in the presence of a changing engineering context can lead to surprising results which far exceed initial expectations. I hope to give the reader an impression of what it is like to develop advanced static analyses in industry, how it is both different from and similar to developing analyses for the purpose of advancing science.

1 Introduction

Static program analysis is a technical subject with well developed principles and theories. We have dataflow analysis and abstract interpretation, inter-procedural and compositional analysis methods, and a great variety of specific abstract domains and logical reasoning techniques. Built upon this foundation we are seeing analysis being applied increasingly to correctness and other problems in the codebases of major companies. It is a very interesting time to work in the area industrially because the applications are so fresh: while there is a wealth of technical information to draw upon, the applications faced are so varied, and the built-up engineering experience so comparatively sparse, that one very quickly encounters challenges at or beyond the edge of both the research and engineering sides of the subject.

In this paper I want to provide an example of what working at the edge of the subject is like in an industrial setting. The presentation is necessarily limited and personal. Nonetheless, I hope that my account might be useful to people who are interested in knowing more about the practice of doing program analysis work from an industrial perspective.

There's no need to build up suspense: The project I describe, RacerD, went much better than I had ever dared hope. It has found over two thousand data race bugs which have been fixed by Facebook programmers before code reaches production, it has been instrumental in the conversion of part of Facebook's

Android app from a single-threaded to a multi-threaded architecture, and, unusually for static analysis, it even has received press attention[1]. A research paper presents the technical results of the project, including information on the analysis algorithm, its empirical evaluation, and its impact [1]. I won't repeat this information here.

In most research papers one gets a description of where a project got to, but not how it got there. We get the results of science and/or engineering, but not a picture of how it developed. For the practice of doing science and engineering I've always thought the issue of "how it's done" is an important one, and that examples describing how can have some value. I wouldn't want to force this view on anyone; some might not feel it is worth it to bother with recounting the process in an example, and might therefore prefer to go directly to the research paper and stop reading here. But I personally have derived lots of value from stories I heard from people on how work was done.

In this paper I will tell you about the development of the RacerD project, its twists and its turns, the compromises and design decisions we made to achieve an impactful analysis, and how jumping back and forth between the perspective of an engineer and that of a scientist helped. My aim is to convey what it's like to work on an open research problem in static analysis, while at the same time pursuing the industrial goal of helping people, and how these two activities can even boost one another.

2 A Small Bold Bet

At the start of 2016 I was a manager at Facebook supporting the Infer static analysis team. Infer is an analyzer applied to Java, Objective C and C++ code [5], reporting issues related to memory safety, concurrency, security (information flow), and many more specialized errors suggested by Facebook developers. Infer is run internally on the Android and iOS apps for Facebook, Instagram, Messenger and WhatsApp, as well as on our backend C++ and Java code. It has its roots in academic research [6], which led to a startup company (Monoidics Ltd) that was acquired by Facebook in 2013. Infer was open sourced in 2015[2], and is used as well at Amazon, Spotify, Mozilla, JD.com and other companies.

I had been in the management role for 2 years, but despite the team doing well (or perhaps, rather, because of it) I was itching to get back to doing technical work myself. I began toying with the idea to go after a problem I had thought about for years, but had never been brave enough to really try to solve to the point of supporting working programmers: concurrency analysis. I had described my aims the year before in an interview with Mike Hicks for his "Programming Languages Ehthusiast" blog:

[1] e.g., thenewstack.io/facebook-engineering-takes-bite-concurrency-racerd/, www.infoworld.com/article/3234328/java/racerd-detects-hard-to-find-race-conditions-in-java-code.html and www.techrepublic.com/article/facebook-open-sources-racerd-a-tool-thats-already-squashed-1000-bugs-in-concurrent-software/

[2] code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/

> I still want to understand concurrency, scalably. I would like to have analyses that I could deploy with high speed and low friction (e.g., not copious annotations or proof hints) and produce high-quality reports useful to programmers writing concurrent programs without disturbing their workflow too much. Then it could scale to many programmers and many programs. Maybe I am asking for too much, but that is what I would like to find.[3]

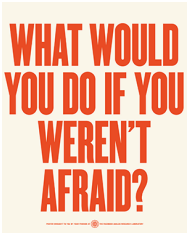
It was time to see if this was really asking for too much.

It's worth remembering why concurrency analysis is not easy. Say an analyzer wanted to examine each of the different ways to interleave two threads of 40 lines of code. Even if we could process 1 billion interleavings per second, it would take more than 3.4 million years to examine them all! If we extend the number of instructions to just 150 per thread, we get over over 10^{88} interleavings, which is greater than the estimated number of atoms in the known universe (10^{80}). The inescapable conclusion is that, even though computers are powerful, we can't explore all the possibilities by brute force.

Yes, there are ways to cut down the number of interleavings, and there have been many interesting research papers on concurrency analysis. But, fundamental problems remained in applying these techniques at scale. Note I am talking here about analyses that do inference about concurrency, not ones that check type hints (e.g., concerning what locks guard what memory) written by programmers. See the companion paper for more discussion of prior work [1].

When I was first thinking of tackling this problem my manager – Bryan O'Sullivan, director of the Developer Efficiency org inside Facebook – was hugely supportive. I remember saying: "you know, Bryan, this is risky, there's a good chance I'll fail completely." To which he replied: "even if you fail, we'll have learnt something and that will be valuable ... I'm completely supportive of you doing this." Bryan and I agreed that it would be good for me to shed a portion of my management responsibilities to free up time for technical work, and Dino Distefano kindly stepped up to take over some of these responsibilities[4].

So with this as backdrop, and surrounded by the upbeat sayings in posters on the Facebook office walls such as



[3] http://www.pl-enthusiast.net/2015/09/15/facebooks-peter-ohearn-on-programming-languages/

[4] I have since shed management responsibilities entirely, moving to a research/engineering role, essentially as a consequence of enjoying doing the technical work on this project.

to offer encouragement, I decided to have a go at making a scalable, low friction, high signal[5] concurrency analysis for the programmers.

3 Understanding the Engineering Problem, Part 1: The Code

At this point I felt had a solid understanding of the scientific side of the problem, and I had a potential way in. I had developed Concurrent Separation Logic (CSL,[9]), and by the work of numerous creative people it had shown how a surprising distance could be gotten in reasoning about concurrent programs without enumerating interleavings (see [4] for a survey). My very general idea was: I will make a CSL analysis that is highly compositional, it will as a result scale to millions of lines of code, and we'll automatically prove some properties of large industrial codebases (see [8] for background and open problems on compositional analysis at scale).

The first property I chose to focus on was absence of data races. This is one of the most basic problems in concurrency. In addition to races putting data structures into inconsistent states, getting rid of them minimizes interference, thus simplifying the task of understanding a program. And, I thought, an effective data race detector could open up other possibilities in concurrency analysis.

The idea "I will make a CSL analysis" is, however, rather abstract from an engineering point of view. I needed to better understand the engineering context where it would be used. My job was to help Facebook's programmers to move fast and produce more reliable code, and I thought I should spend time understanding the concurrent code that they write in order to see if I could serve them. So I spent the next three months scouring Facebook's Android codebases performing mental CSL proofs of race freedom (or, mental bug detection for failed proofs) to understand what I was up against. I made three discoveries.

The first is that Java's explicit thread spawning was seldom seen. This was handled by Android framework code, but also special code in a Facebook UI library called Litho. On the other hand, the programmers were annotating classes with a @ThreadSafe remark, as follows:

```
@ThreadSafe
class C{
    Fields
    T1 m1() { ... }
    ...
    Tn mn() { ... }
}
```

This shortened my path to getting started. I had spent several weeks designing an analysis algorithm for dealing with Java's thread creation, concentrating in particular on carrying around enough information to report errors when a proof failed. Now I would not need that algorithm. Rather, I would choose "the

[5] See [7] for further discussion on problems related to friction, signal, etc in program analysis.

(non-private) methods of a class don't race with one another when run in parallel" as an approximation of the intuitive concept of thread safety: my analyzer could choose specific parallel compositions to shoot for.

The second discovery was that CSL could in principle reason well about all of the code I had seen. I could see how to do manual proofs about the coding idioms I had encountered, the main questions were the level of automation I could achieve and how to give good feedback when proofs failed.

An interesting example I came across was from the file `ComponentTree.java` from Facebook's Litho library, where the following comment indicates a pattern where programmers try to avoid synchronization.

```
// This 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.
private LayoutState mMainThreadLayoutState;
```

Litho is open source (https://github.com/facebook/litho) so you can go find the code if you'd like. Boiling down what is happening to an example in "Java with parallel composition", the comment is indicating that

```
   synchronized (lock) { x = 42 } ; y = x;
|| synchronized (lock) { z = x; }
```

is race free. You can read from `x` outside of synchronization in the first thread because it is not written to in the second. There is a nice proof of this example using CSL with fractional permissions [3, 2], which looks like this:

```
   Resource Invariant: x:1/2.

   [x:1/2] synchronized (lock) { [x:1] x = 42; [x:1] };
             [x:1/2] y = x; [x:1/2]
||
   [x:0] sychronized (lock) {[x:1/2] z = x; [x:1/2] } [x:0]
```

With fractional permissions if an lvalue has permission 1 then you can read from or write to it, for fraction >0 but <1 then you can read but not write, and permission 0 means you can neither read nor write. The resource invariant is added when you enter a `synch` block and subtracted when you leave: we get permission 1 inside the first `synch` block just because $1/2 + 1/2 = 1$.

I also found examples of ownership transfer, similar to some of the early examples used to illustrate CSL, where the permission to dereference storage transfers from one thread to another. For example, again in `ComponentTree.java`:

```
// The semantics here are tricky. Whenever you transfer
// mBackgroundLayoutState to a local that will be accessed
// outside of the lock, you must set mBackgroundLayoutState
// to null to ensure that the current thread alone has access to
// the LayoutState, which is single-threaded.
private LayoutState mBackgroundLayoutState;
```

Reasoning about the correct usage of `mBackgroundLayoutState` can be done using a disjunctive resource invariant, like

```
mBackgroundLayoutState == null    OR    mBackgroundLayoutState |-> -
```

which we read as saying that either `mBackgroundLayoutState` is `null` or it is a non-null pointer that is owned by a lock. While we could indeed handle such examples with CSL, from looking at the code I hypothesized that the vast majority of it, I guessed 95% to 98%, did not need the kind of disjunctive invariants used to account for ownership transfer, but rather simpler ones like `x:1/2` as above.

The third discovery was that by far the most common synchronization in the codebase was expressed with the `synchronized (this)` construct. There were indeed cases where two locks were used in a single file, there were some instances of read/write locks, and even a counting semaphore. But, the majority of cases involved `synchronized (this)`. Note that these were by no means trivial, because of (for example) idioms like the `ComponentTree.java` example above where programmers would attempt to avoid synchronization. But it suggested: initial impact could be had even without an analysis that attempted to accurately distinguish different locks.

The detective work I did here – reading code and constructing mental proofs and understanding what coding patterns were more and less frequent – while not glamorous was, in retrospect, an incredibly valuable part of the project. The three discoveries above are not novel, but are just part of the kind of normal scoping work one does before embarking on an engineering project. Rather than novelty, the goal is to gather enough relevant information to inform engineering judgement as to the way forward. In this case, the original problem was considerably simplified as a bonus. At this point I had a much better understanding of the engineering context, enough to form a more concrete plan for progress.

4 Pre-α prototype

By July 2016 I was ready to get started with my tool. It would infer resource invariants for fractional permissions as well as preconditions for threads, and apply these to Facebook's Java codebases. At first the only lockiong I would treat would be by critical sections `synchronized (this){..}`, and I would aim for simple, un-conditional invariants. Programs using multiple locks within a file or ownership transfer examples, the minority, would be handled either via manually supplied invariants or by an improvement to the algorithm after the simpler majority had been dealt with.

I worked out a pen-and-paper analysis algorithm based on CSL with fractional permissions, and I circulated an internal note to the Infer team. I was very excited at the prospects, as I estimated that we could prove race freedom for hundreds or thousands of classes, and find tens of data race bugs, after the ideas were implemented and deployed. At this point I also sparked up a collaboration with academics, Ilya Sergey from University College London and Nikos Gorogiannis from Middlesex University, to help work on the theoretical foundations

of the analyzer. Their input on the theory, and later the practice, would turn out to be important (more on that in the last section of the paper).

The coding was a big task for me, but it was also great fun and very satisfying. I had not programmed for years, and had never in truth written that much code; I was a theorist for most of my career. Sam Blackshear had created a framework I was using, Infer.AI, for writing compositional abstract interpreters, and I convinced him to be my coding mentor (at the same time that I was officially still his manager).

I didn't begin by implementing the algorithm to infer resource invariants and preconditions; that would have been too intricate a starting point. Rather, to learn how to make an inter-procedural analysis which reports inter-procedural errors, I decided that my first prototype would simply search for certain obvious errors: writes outside of any synchronization. Any such write would give rise to a potential self-race, when its enclosing method was run in parallel with itself. Here is an example:

```
SelfRace.java:20: error: THREAD_SAFETY_VIOLATION
  Unprotected write. Non-private method ‘SelfRace.foo‘ writes to field ‘this.SelfRace.y‘
  outside of synchronization. Reporting because the current class is annotated ‘@ThreadSafe‘,
  so we assume that this method can run in parallel with other non-private methods in the
  class (including itself).
  18.    void foo() {
  19.      synchronized(this){ x = 42; };
  20. >    y = 84;
  21.    }
  22.
```

I did not concentrate on these self races because I thought they were so important, but rather to give me something concrete and simple to shoot for in the prototype. I knew that when the real version landed later I would need to provide engineers with effective signal when a potential bug was found, so I optimized my early coding towards that (which I needed to learn about) rather than proof (which I knew better).

By October 2016 I had knocked out a kLOC of code and completed a prototype which I was running on Facebook's Android codebases. The analyzer was not reporting issues to developers but was rather running from time to time silently, in stealth mode. I started planning on converting it over to a CSL prover. I thought I could see how to do this; I could imagine a first prover ready by June 2017 and, after α and β versions shipped to volunteers for feedback, a real launch by the end of 2017.

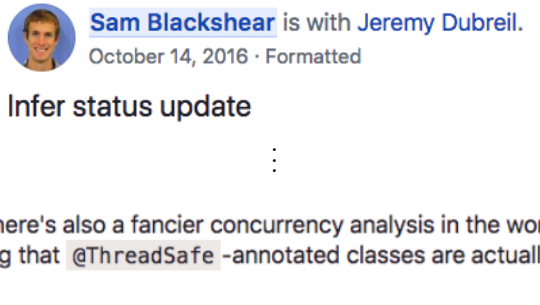
You might wonder why I thought it would take a whole year. Well, it was only me programming at this stage and, additionally, from the first internal release of Infer in 2014 I remembered how much the tool improved during the $\alpha\beta$ training period. I didn't want to underestimate the importance of the training period. I thought I could calmly and carefully approach a successful launch of a tool for doing concurrency proofs at scale.

So things were going well. I thought I'd have my CSL prover in production in about a year. That would have been scientifically novel I thought, and would likely have produced non-negligible impact and a springboard for doing more. I had no concrete impact thus far after working for 10 months on the project, but

my manager Bryan was very happy with this progress and remained supportive. (In fact, he said that he *preferred* that I bet on uncertain long-term work that could fail, rather than more certain short-term work.)

5 Intervention and Pivot

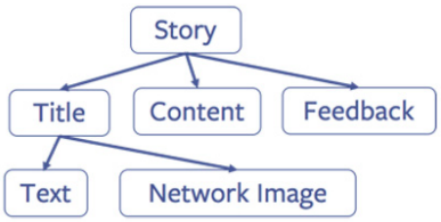
In October 2016 Android engineers inside Facebook caught wind of the project when Sam let the cat out of the bag in a side remark during a status update to an internal Android group.



The first question that came in response to Sam's post asked about what in program analysis jargon we would call inter-procedural capabilities.

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

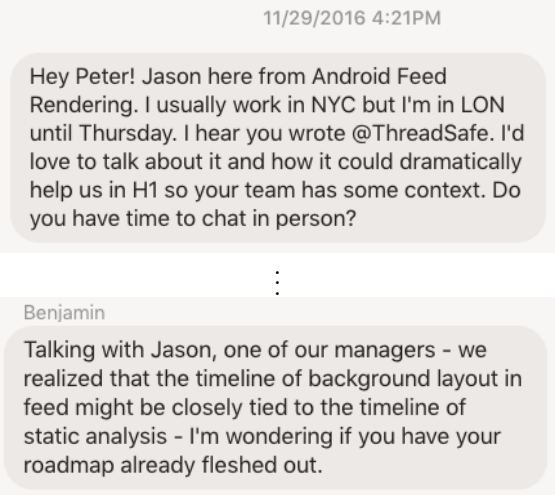
The story is that a team in NYC was embarking on a new venture, converting part of the News Feed in Facebook's Android app from a sequential to a multi-threaded architecture. This was challenging because hundreds of classes written for a single-threaded architecture had to be used now in a concurrent context: the transformation could introduce concurrency errors. They asked for inter-procedural capabilities because Android UI is arranged in such a way that tree structures like



are represented with one class per node. Races could happen via inter-procedural call chains sometimes spanning several classes. Furthermore, the Litho library for UI being developed at Facebook was arranged in such a way that mutations

almost never happened at the top level: *intra*-procedural race detection would miss most races.

I was naturally pleased to have engineers express interest in the concurrency project, doubly-pleased in fact because inter-procedural reasoning is one of Infer's strong points. But, I kept my head down working to my planned timeline, which would see release a year later. Then, in November the level of interest ramped up considerably, for example when I received these messages:



At this point I had a decision to make. I could seek to damp down expectations and continue with my plan of releasing my prover in a year, or I could pivot to serve the engineers as well as I could, as soon as possible. As a scientist I would perhaps have regretted giving up on my "beautiful" prover, but after moving from university to work in an engineering org I found it useful to frequently remind myself of the difference between science and engineering.

> "Scientists seek perfection and are **idealists**. ... . An engineer's task is to **not be idealistic**. You need to be realistic as you *have* to compromise between conflicting interests." Sir Tony Hoare[6]

These words of Tony helped me to stay grounded as I contemplated the compromises that would inevitably be needed were I to choose the second course.

It was in truth an easy decision to make. The job of the Infer team is to help Facebook's programmers: it is only right to respond and help them, and we place this duty above (say) abstract aims concerning the exact shape of an eventual tool. Furthermore, when you try to deploy a static analysis it can sometimes seem like an uphill struggle to convince people that heeding the warnings will be worthwhile. Their time is valuable and one needs to be very careful not to inadvertently waste it. Here, though, we had a team *asking* for static analysis support from a project we were already working on, for an important project

[6] https://reinout.vanrees.org/weblog/2009/07/01/ep-keynote.html

they were planning, and where the potential near-term impact of doing so far outstripped anything I had thought about. This seemed like a golden opportunity for someone working in static analysis.

So, I made the decision to pivot to meet the Android engineers' needs as well as we could in a timeframe that would fit their goals. I asked Sam to stop (only) being my mentor and to join the project immediately as a full contributor himself. He agreed, and this would considerably expand our capabilities.

6 Understanding the Engineering Problem, Part 2: The People

Sam and I met with two Android engineers, Benjamin Jaeger and Jingbo Yang from Facebook's NYC office, and agreed that we should spec out a *minimum viable product* (MVP) for serving them. After discussing what we thought might and might not be possible technically in a three-month timespan, Ben came back to us with a first draft of an MVP, which we iterated on, arriving at something that included the following (in essence).

1. High signal: detect actionable races that developers find useful and respond to. Find many races that would be encountered in practice, but no need to (provably) find them all.
2. Inter-procedural: ability to track data races involving many nested procedure calls.
3. Low friction: no reliance on manual annotations to specify which locks protect what 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, as used in most of product code, but no need for precise analysis of intricate fine-grained synchronization (the minority, changing rarely, found in infrastructure code).

The requirement for high signal balances false positives and negatives rather than seeking idealistic perfection in either direction. It is common in engineering to accept imperfection but to measure and improve metrics over time. The engineers were comfortable with this requirement; in fact, they suggested it when we said we could not produce a prover with few false positives in three months.

The requirement for low friction was particularly important. Hundreds of classes needed to be placed into a multi-threaded context. If humans were required to write annotations about which locks protected what data, the kind of thing one finds in `@GuardedBy` type systems for Java, then it would take too much time. Inference would help our engineers to move fast.

It is important to note that the specification of the MVP was not arrived at by abstract reasoning alone. It was conceived of by balancing what was technically possible (in our opinion) with the specific needs of people facing a specific problem (convert News Feed), as well as other people in the general engineering population who we needed to help prevent concurrency regressions. By aiming

to strike this balance we were able to convert a seemingly intractable problem into a one where an initial success that we could then build on was in sight.

7 Development and Deployment of RacerD

As it happens my pre-α prototype provided a good starting point. It was recording information about accesses and whether any lock was held at a program point. Sam and I set about completing the MVP for RacerD. It was implemented as a special abstract interpreter within the Infer.AI framework. Our design included principles guiding the initial and further developments, specific abstract domains, and a means of determining potential races given what the abstract domains compute. You can see the details in the companion paper [1].

Apart from the specifics, a general design goal of RacerD was simplicity: we were determined not to introduce technical innovations unless they were absolutely necessary. As technical program analysis people it is always tempting to try this or that technique. But we employed occam's razor rigorously, guided by experiment as well as logic, and this was important for how the tool would evolve. If we were to introduce additional complex techniques without extremely good reasons it would have slowed us down, affecting the turnaround time for responding to engineer feedback, and could even have imperiled the project.

After Sam and I completed the MVP, Ben and Jingbo began applying RacerD to classes in our Android codebase. They encountered many false positives, which often had to do with either access to newly allocated entities that had not escaped, and that therefore could not be interfered with. Sam had worked out an ownership analysis for tracking these entities; it was the most challenging part of getting a high-signal analysis, and it was refined over several months. Other problems came up from threading assumptions. Sam and Ben designed an annotation system for expressing assumptions about ownership and threading to help the anlayzer along, and which reflected the way our engineers thought about the code. By this point Sam was doing most of the implementation work responding to false positives and other feedback.

Meanwhile Jingbo was converting vast quantities of Android code to a concurrent context, reporting false alarms to us, and running experiments to keep the conversion moving in the right direction. By October of 2017 it was clear that both projects, RacerD and the conversion of News Feed in Android, were successes. Jingbo announced a performance win for the multi-threaded News Feed, and RacerD had caught thousands of potential data races both in Jingbo's conversions and in automatic comments on code modifications from hundreds of other engineers. I won't belabour the point that the projects went well: you can read more about multi-threading in a blog post[7], and more about RacerD in its own blog[8] and in the companion paper [1]. And there is more to come in

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

[8] https://code.facebook.com/posts/293371094514305/open-sourcing-racerd-fast-static-race-detection-at-scale/

the future: already RacerD has enabled further work beyond races, on deadlock detection and starvation, and a modified version of it is in production for C++ as well as for Java. Perhaps the CSL prover will even make an appearance.

Instead of talking more about what was achieved, I want instead to recount some of the further hiccups we had while doing the work, and how we tried to react to them.

8 Hiccups, Theorems, Science and Engineering

We had used reasoning intuition in building an analyzer that quickly searches for likely races, but we did not formalize what we were doing before we did it. However, in parallel, theoretical work was happening, driven mostly by Nikos and Ilya, with Sam and I participating but concentrating more on the in-prod analyzer.

Nikos and Ilya created an idealized analyzer, written in LaTeX, based on CSL with fractional permissions. They then constructed a small prototype, a "baby prover", written in OCaml. There was a gap between the baby prover and the in-prod RacerD. Our original plan was to close the gap, but we found it widening over time because the in-prod analyzer was mutating at a fast rate in response to developer feedback. There was not unanimity of opinion on what to do about this gap.

A non-starter would have been to pause the development of the in-prod analyzer, repeat its work to grow the baby prover, wait until the prover had matured enough to have a low enough false positive rate, and then do the switch-over. The in-prod analyzer was serving our programmers and improving steadily; we did not want to sacrifice this for idealistic purposes.

We wondered if we could instead modify the in-prod analyzer to be sound for bug prevention (over-approximation). I suggested to identify a subset of analyzer runs which corresponded to proofs. We added soundness flags to the analyzer and tried to keep track of sound moves, but this ultimately did not work out. Next, Sam implemented an escape analysis to find race bugs due to locally declared references escaping their defining scope, to reduce the false negatives. The escape analysis led to too many false positives to put into production and we abandoned it. Nikos tried another tack to reduce false negatives: a simple alias analysis, to find races between distinct syntactic expressions that denote the same lvalue. Again the attempt caused too many false positives to put into prod.

The above should not be taken as a commentary on soundness versus unsoundness in general, or as saying unsoundness is necessary or desirable in industrial program analysis. I see no reason why the baby prover could not be grown to the point of being good enough for production. But, having a very effective analyzer in place meant that the priority for doing this became less than if we had no analyzer at all in place.

So we had engineered RacerD to search for probable race bugs rather than to exclude races, we viewed this as a compromise, and we were responding to this compromise by trying to change the analyzer to fit a standard soundness

theorem. And we were not succeeding. Then, one day in Sept 2017 I thought to myself: this analysis is actually very effective, the signal is surprisingly good, few false positives are being reported. Those false positives that were turning up often resulted from missing assumptions (e.g., about threading or ownership) rather than problems in the analyzer design. So I thought, rather than try to change the analyzer so it fit a pre-conceived soundness notion, let's try to understand why the already-effective analyzer is effective and formalize a portion of that understanding. I proposed a theorem (or, hypothesis) as follows.

> *TP (True Positives) Theorem:* Under certain assumptions (assumptions reflecting product code), the analyzer reports no false positives.

An initial version of the assumptions was described as follows. Consider an idealized language, IL, in which programs have only non-deterministic choice in conditionals, and where there is no recursion. The absence of booleans in conditionals reflects the idea that in product code very coarse grained locking is typically used. We don't often, for example, select a lock conditionally based on the value of a field, like we do in ownership transfer or fine-grained concurrency examples. The no recursion condition is there because we want to say that the analyzer gets the races right except for when divergence make a piece of code impossible to reach. Now IL does not perfectly represent product code (nothing does as far as we know), but it serves as a useful working hypothesis for theory.

[**Aside**. One can see the TP Theorem as establishing an under-approximation of an over-approximation. Start with a program without recursion. Replace booleans by non-determinism (that's over-approximation). Then no false positives (under-approx). The more usual position in program analysis is to go for the reverse decompostion, an over-approximation of an under-approximation. The "under" comes about from ignored features, and the "over" from a usual sound-for-bug-prevention analysis. In contrast, under-of-over seems like a good way to think about static analysis for bug catching. (And yes, I know the above can even be thought of as under of over of under, where the first "under" ignores recursion.)]

Now, the True Positives Theorem was not actually true of the in-prod analyzer when we formulated it, and it is still not. But we took it as a guiding principle. We would subsequently, when faced with a design choice, take an alternative consistent with the theorem. For example, we implemented a "deep ownership" assumption: If an access path, say `x.f`, is owned, then so are all extensions, e.g. `x.f.g`. The analyzer would never report a race for an owned access. The deep ownership assumption goes against soundness for bug prevention (over-approximation), but is compatible with soundness for bug finding (under-approximation). The TP Theorem also provided rationalization for our earlier decisions not to deploy alias or escape analysis.

Since then, Nikos and Ilya have proven a version of the TP Theorem for a version 2 of RacerD, a modified version of the analyzer which is not too far removed from the in-production version. While we had difficulty seeing how to modify RacerD to be sound for bug prevention, we were able to modify it to be consistent with the TP theorem (sound for bug catching, under assumptions). We

are in the process of measuring and evaluating the differences between versions 1 and 2. I won't say more on the status of that work as it is in progress, but I would like to talk about science and engineering in program analysis in light of the above.

First, about the role of the TP theorem. We did not think of the theorem before we started, and then go on to implement an analyzer to satisfy the theorem. Rather, the theorem came up later, in response to the surprising behaviour we observed. If you think about the way software is developed in an iterative way then this seems natural: analyzers do not need to be developed according to a (straw man) waterfall model, any more than general software does. And as an aside, if you think about the way research is developed, neither does it usually flow in a neat waterfall fashion.

Second, the role of the theorem is not to provide iron-clad guarantees to rely on. Rather, it has been used to help understand the analyzer and guide the design while it's in flight. This reminds me of something my PhD supervisor Bob Tennent emphasized to me many years ago, which I did not fully appreciate at the time. Bob said that, yes, semantics can be used to show what is true about a programming language after it is set in stone, but it is (arguably) even more valuable in helping inform design when a language is being created [10]. The same is true for program analyzers: RacerD is but one illustration.

Finally, you can see that both science and engineering have played important roles in the development of RacerD. Science gave rise to original ideas on sequential reasoning about concurrent programs, it gave us compositionality, abstraction, and much more. It gave us the basic ideas to get started at all in a way that scales. Then contact with Android engineers led to compromises to get an analyzer that is effective in practice. Trying to understand why the analyzer was effective caused us to go back to science to formulate the TP theorem, and this in turn influenced further development of the analyzer.

This way of having science and engineering playing off one another in a tight feedback loop is possible, even advantageous, when practicing static analysis in industry at present. The subject is not so over developed that exclusively-engineering work is typically the best route to good results, and not so under developed that exclusively-science is the only thing feasible to do. I hasten to stress that exclusively-engineering and exclusively-science remain valuable, things that folks should feel comfortable and well justified in doing. My main point is rather that the current state of the subject, with the possibility of tight science-engineering feedback, makes for a rich variety of ideas and problems to explore.

Acknowledgements. Thanks first of all to Jingbo Yang and Ben Jaeger for being so giving of their time, experience and insight when working with Sam Blackshear and I. And thanks to Sam for taking the engineering of RacerD to another level. I am fortunate to have participated in this collaboration with Ben, Jingbo and Sam. I'm grateful to Nikos Gorogiannis and Ilya Sergey for joining and contributing greatly to the project. Finally, special thanks to Bryan

O'Sullivan for consistently and enthusiastically supporting this work, and encouraging me to proceed and be bold.

References

1. S. Blackshear, N. Gorogiannis, I. Sergey, and P. O'Hearn. Racerd: Compositional static race detection. In *OOPSLA*, 2018.
2. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pp59–70, 2005.
3. J. Boyland. Checking interference with fractional permissions. In *10th SAS Symposium*, volume 2694 of *LNCS*, pages 55–72, 2003.
4. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.
5. C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P.W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11, 2015.
6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
7. M. Harman and P. O'Hearn. 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*, 2018.
8. P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, Oxford, July 2018.
9. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.
10. R. D. Tennent. Language design methods based on semantic principles. *Acta Inf.*, 8:97–112, 1977.