RacerD

RacerD is a data race detector for Java code. It focuses a high-confidence data races, meaning it is willing to miss certain races in exchange for a reduction in false positives. It is a static tool, so it offers good code coverage. The tool emphasizes speed, scaling to millions of lines of code projects. It supports compositional analysis, so it only needs to evaluate against incremental changes to the code, reducing analysis runtime significantly.

RacerD is one element of Facebook's Infer static analysis suite. Thus, it runs like any other Infer analysis. Infer performs its static analysis during the build process. You simply prepend "`infer --`" to an existing project's build command. You can add the `--racerd-only` flag to run only the RacerD portion of the analysis.

Experimental Setup

We evaluated RacerD using a Facebook-provided, precompiled build of Infer v0.17.0. All testing was conducted in an Ubuntu 18.04.4 LTS 64-bit virtual machine with access to all 8 threads on an Intel Core i7-6700K@4.4GHz. We performed testing on 4 existing open-source Android apps in addition to a variety of multithreaded Java programs sourced from the companion materials of *The Art of Multiprocessor Programming*. Lines of code were evaluated using CLOC.

Infer was designed to integrate into existing build systems, including Gradle, the standard Android app build system. As a result, running analysis on Android apps is straightforward. For each app, we imported the code from the source repository, opened it in Android Studio where dependencies would automatically be installed and resolved, then checked the default build command. That command could then be run from the terminal to build without Infer. From there, we prepend our Infer command. Typically, this looks something like "`infer --racerd-only -- ./gradlew clean build`" to clean, build, and statically analyze the Android app. We analyzed Telegram, a secure messaging app; Open Camera, an open-source camera app; Firefox Focus, a privacy-oriented web browser; and VLC, a media player.

The companion materials from *The Art of Multiprocessor Programming* are small, functional examples of multithreaded programming in Java. They were created in Netbeans and thus use the Ant build system, which is supported by Infer. Each example was built using the command "`infer --racerd-only – ant jar`".

Results

While the original RacerD research paper evaluated the tool's performance on the Facebook app, we extend the analysis to additional full-scale Android apps. Measuring the number of lines of code, the scale of the evaluated programs varies widely, from thousand of lines of code to millions. In Figure 1, we evaluate the number of lines of code, both Java specific and overall; the build time with and without Infer's analysis, to evaluate the overhead of race detection; and the number of reported bugs.

The overhead added by running Infer on the code was between 1% and 120%. Comparing against the number of lines of code, the size of the project has little influence on the overhead. This is likely because RacerD only analyzes code that it believes can be executed in a concurrent context.

Firefox Focus identifies no bugs. This is likely because a large portion of the codebase has been converted to Kotlin, which RacerD cannot analyze. Furthermore, because RacerD infers what code can be run in a concurrent context, it is possible that Java code called concurrently from Kotlin will not be identified or analyzed by RacerD. Finally, Mozilla is listed as a partner company that uses Infer, so it is likely that the development team there has already used Infer to find and fix data races.

Telegram fails to compile when building with RacerD. Infer throws an error internally during analysis. While this means the tool could not generate a bug report, it also illustrates that there are circumstances where RacerD is not a simple drop-in analysis tool. Telegram is the largest, most complex app we tested on Infer, and it is not clear what causes the incompatibility.

	Open Camera	Firefox Focus	VLC	Telegram
Lines of Java code	53,222	9,905	11,176	533,399
Total lines of code	78,056	312,565	125,470	1,659,957
% Java	68%	3%	9%	32%
Build time (seconds)	20	46	137	423
Infer build time (seconds)	44	84	139	Error
Overhead (%)	120%	83%	1%	Error
Reported Bugs	239	0	670	Error

Figure 1: Android app analysis

Interestingly, when evaluating the companion material, several data races were identified. The results of the analysis are included in Figure 2. Queue contains a simple bug in the enqueuer code where it unlocks but never acquires the lock. Priority contains several data races in the sanityCheck() function, which accesses shared memory without locks, suggesting it was meant to run sequentially after a concurrent execution, where locks would no longer be required. TinyTM also contains potential data races in an atomic array class, as several shared variables are modified without any explicit synchronization. Monitor accesses a map data structure without any synchronization. RacerD has no explicit support for monitor semantics, but instead of ignoring the monitors as expected, it instead handles them incorrectly and reports races in the enqueue and dequeue operations, a rare case of false positives. All other identified races in the companion material were true data races. Note that several of these projects use atomic operations, which RacerD will not analyze.

	Mutex	Register	Spin	Queue	Combine	Steal	Priority	TinyTM	Monitor
Lines of Java code	632	769	1,287	1,526	522	1,313	1,186	2,469	261
Total lines of code	1,175	1,312	1,830	2,070	726	1,843	1,714	3,091	792
% Java	54%	59%	70%	74%	72%	71%	69%	80%	33%
Reported Bugs (actual)	0 (0)	0 (0)	0 (0)	5 (5)	0 (0)	0 (0)	2 (2)	4 (4)	7 (5)

Figure 2: Companion material analysis