RaceMob: Crowdsourced Data Race Detection

By : Baris Kasikci, Cristian Zamfir, and George Candea

Presentation by: Jeremy Bradford

Background: Data Races

- Data Race – when 2 or more threads in a program access data in an undetermined order and at least one of these accesses is a write

Motivation

- Data races are some of most costly and difficult to find bugs in multithreaded systems
- Exponential number of interleavings means impractical to test them all, so bugs can remain hidden

Solutions?

- Static detectors
  - Problem: many false positives (e.g. RELAY 84%)
  - Cannot accurately infer what is multithreaded
  - Handling of synchronization primitives

- Dynamic detectors
  - False positives are rare
  - Problems: high runtime overhead, false negatives

False Negative – Happens Before

Thread 1	Thread 2	Thread 1	Thread 2
x = 1		x = 1	
Lock(l)			Lock(l)
Unlock(l)			Unlock(l)
	Lock(l)		x = 2
	Unlock(l)	Lock(l)	
	x = 2	Unlock(l)	

RaceMob

- 2-phase data race detector
- Uses static checking and dynamic checking
- Crowdsources validation of statically determined potential data races
  - Why crowdsourced?
  - Reduced overhead and real user execution

Phase 1: Static Detection

- Racemob uses RELAY, a lockset-based detector
- Could use any static detector, preferably complete
  - RELAY complete when no pointers, inline assembly

RaceMob

List of Races

- Unknown
- True Race
- Likely False Positive

"Hive"

- Assignment of Tasks
- Updating List

User Site

- Dynamic Context Inference
- On-demand Detection

Phase 2: Dynamic Validation

- Dynamic Context Inference (DCI) – lightweight initial verification (always on)
- On-demand data race detection
- Schedule Steering

Dynamic Context Inference

- Validates the statically determined races
- Checks for 2 conditions:
  - Concrete instance of aliasing
  - Access from different threads
- Negligible runtime overhead (0.01%)
- Small memory footprint (12 bytes per race)

On-demand Race Detection

- Starts tracking happens-before relationships after first potentially racing access is made
  - No Race: happens-before relationship established between first accessing thread and all other threads
  - Race: Access in another thread before happens-before relationship

Minimal Monitoring in RaceMob

Thread 1	Thread 2	Thread 3
First Access	...	...
Barrier	Barrier	Barrier
...	...	Second Access
...	...	...

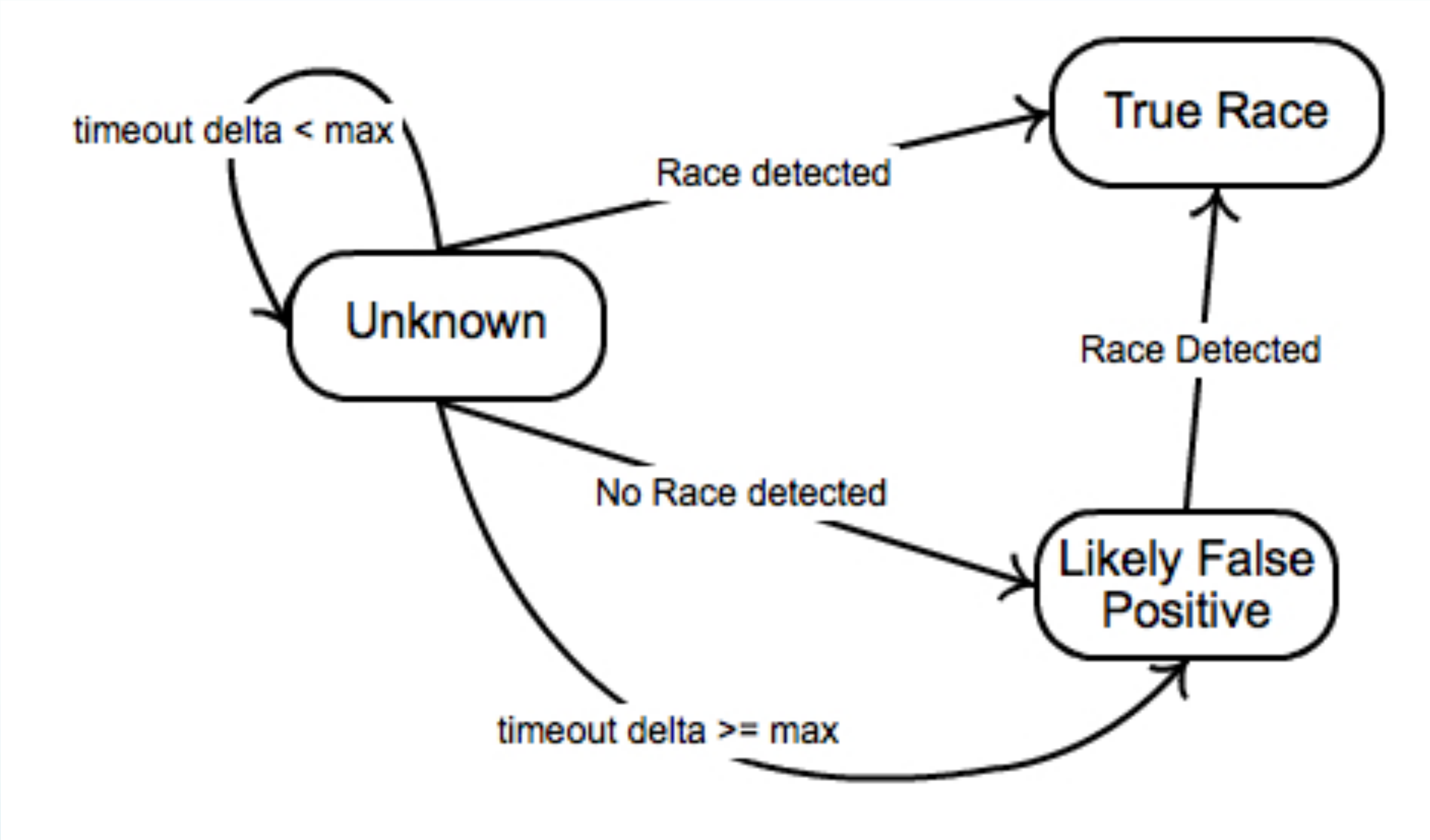
Minimal Monitoring in RaceMob

Thread 1	Thread 2	Thread 3
First Access	...	...
...	...	Second Access
Barrier	Barrier	Barrier
...	...	...

Schedule Steering

- Tries to force different orders of execution for greater coverage
  - Pauses thread that is about to access data if not "scheduled thread"
  - If incorrect order, reports a timeout to the Hive, which may increase pause time up to a maximum
- Timeout generally kept small for low overhead
- Successful: found races otherwise undetected

Dynamic Validation



Results

- 106 total data races in 10 programs
- 0% false positive for detected races
- Efficiency
  - Runtime overhead average 2.32%, maximum 4.54%
- Found 2 previously undiscovered hangs in SQLite

Comparison: Reported Races

Program	Apache	SQLite	Fmm	Aget	Pfscan
RaceMob	8	3	58	4	2
TSAN	8	3	58	2	1
RELAY	118	88	176	256	17

Comparison: Total Overhead

Program	Apache	SQLite	Fmm	Aget	Pfscan
RaceMob Aggregate Overhead	339%	282%	1598%	144%	103%
TSAN Average Overhead	25,208%	1429%	47888%	184%	13402%

Issues with RaceMob

- Additional overhead for client
- Less in-house testing/releasing buggy software?
- Privacy implications
- Crowdsourcing with dishonest or malicious users

Final Thoughts

- Innovative combination of static and dynamic methods
- Much more accurate and with lower overhead than many of today's standard tools
- Questions concerning privacy with crowdsourcing