RaceMob: Crowdsourced Data Race Detection

Baris Kasikci, Cristian Zamfir, and George Candea

School of Computer & Communication Sciences
Data Races

- Accesses to shared memory location
  - By multiple threads
  - At least one of the accesses is a write
  - Synchronization operations do not enforce an order among the accesses
<table>
<thead>
<tr>
<th>Kept for performance</th>
<th>???</th>
<th>Caused massive losses</th>
</tr>
</thead>
</table>

Spectrum of Data Races

**Kept for performance**

- Kept for performance

**Caused massive losses**

- Caused massive losses

**2003 Blackout**

**An Investigation of the Therac-25 Accidents**

- 2003 Blackout

- **An Investigation of the Therac-25 Accidents**

**Kept for performance**

- Kept for performance

- **Kept for performance**

**Caused massive losses**

- Caused massive losses
Pitfalls

• Programs with data races are incorrect according to POSIX & C/C++ standards

• Compilers can break correctness of programs with data races [HotPar’11]

  • Harmless data races can become harmful

Developers need to know every true data race
How to Find All Data Races?

- **Static race detectors**
  - Full path analysis ✔
  - Cheap (0 runtime overhead) ✔
  - Few false negatives ✔
  - Many false positives (≈80%) ✗

- **Dynamic race detectors**
  - Per-run analysis ✗
  - Expensive (≤ 200x) ✗
  - Many false negatives ✗
  - Few false positives (≈0%) ✔

Existing detectors are not practical
How to Find All Data Races?

- Full path analysis ✔
- Cheap (0 runtime overhead) ✔
- Few false negatives ✔
- Few false positives (~0%) ✔
RaceMob

- Full path analysis ✔
- Cheap (0 runtime overhead) ✔
- Few false negatives ✔
- Few false positives (~0%) ✔

All Memory Accesses

Static Detection ➔ Potentially Racing Accesses ➔ Likely Racing Accesses ➔ True Races

Dynamic Context Inference

On-demand Validation

Crowdsourcing

RaceMob:

- Race & Hang
- Race & Crash

Tuesday 12 November 13
Usage Model

RaceMob

Issues

Application Home
(e.g., AppStore)

program

Users run instrumented programs

Users
Insights

• Use static race detection to prune memory accesses that need not be monitored

• Cost of dynamic detection can be amortized across many users

• Using the crowd, we can detect races that ”matter”
RaceMob

All Memory Accesses

Static Detection → Potentially Racing Accesses → Dynamic Context Inference → Likely Racing Accesses → On-demand Validation → True Races

Crowdsourcing

Detected 106 real races in 10 programs with 2.3% average overhead
Static Data Race Detection

- We use RELAY [FSE’07]
  - Analyzes entire program paths at once
  - Computes & composes per-function summaries to scale
  - Tracks locks

Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1 (\rightarrow) LS(_1) = {}</td>
<td>lock(l)</td>
</tr>
<tr>
<td>lock(l)</td>
<td>(\rightarrow)</td>
</tr>
<tr>
<td></td>
<td>unlock(l)</td>
</tr>
<tr>
<td>x = 2 (\rightarrow) LS(_2) = {}</td>
<td></td>
</tr>
</tbody>
</table>

\(x = 1\) and \(x = 2\) are potentially racing
RaceMob

All Memory Accesses

Static Detection

Potentially Racing Accesses

Dynamic Context Inference

Likely Racing Accesses

On-demand Validation

True Races

Crowdsourcing

Race & Hang

Race & Crash

X

X

X

X
Dynamic Context Inference (DCI)

- Inaccuracy in static data race detection
  - Pointer alias analysis errors
  - Inability to infer multithreaded program context
- DCI checks at runtime:
  - If accesses are from different threads
  - If accesses alias

Dynamic context inference reduced the set of accesses to monitor by 53%
**DCI Example**

Alice

Thread 1

- $x = 1$
- lock(l)
- ...
- unlock(l)

Thread 2

- Address = 0xBEEF
- ThreadID = 1
- lock(l)
- ...
- unlock(l)
- $x = 2$

Address = 0xBEEF
ThreadID = 2

Proceed to on-demand data race validation
RaceMob

All Memory Accesses

Static Detection

Potentially Racing Accesses

Dynamic Context Inference

Likely Racing Accesses

On-demand Validation

True Races

Crowdsourcing

Race & Hang

Race & Crash
On-demand Data Race Validation

- Happens-before based
  - Track synchronization
  - Few false positives
- Minimal tracking
  - Only memory accesses of the target data race
  - Synchronization in between these accesses
    - Until enough happens-before edges form
- Steers thread schedule to expose races

```
Thread 1
x=1
unlock(1)
```

```
Thread 2
lock(1)
x=2
```

Tuesday 12 November 13
On-demand Validation Example

Alice

Thread 1

x = 1
lock(1)
...
unlock(1)

Thread 2

HB

lock(1)
...
unlock(1)

x = 2

No Race

Bob

Thread 1

x = 1
lock(1)
...
unlock(1)

x = 2

No HB

Thread 2

No HB

lock(l)
...
unlock(l)

Race
Detection Results

<table>
<thead>
<tr>
<th>Detection Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>RACE</td>
</tr>
<tr>
<td>RACE &amp; CRASH</td>
</tr>
<tr>
<td>RACE &amp; HANG</td>
</tr>
<tr>
<td>NO RACE</td>
</tr>
<tr>
<td>NOT ALIASING</td>
</tr>
<tr>
<td>NOT MULTITHREADED</td>
</tr>
<tr>
<td>NOT SEEN</td>
</tr>
</tbody>
</table>

Application Home (AppStore, Play)

RaceMob

program

Tuesday 12 November 13
RaceMob

All Memory Accesses

Static Detection

Potentially Racing Accesses

Dynamic Context Inference

Likely Racing Accesses

On-demand Validation

True Races

Crowdsourcing

Race & Hang

Race & Crash

X

X

X

X

Tuesday 12 November 13
Evaluation

- Detection effectiveness
- Contribution of techniques to reducing overhead
- Breakdown of overhead
- Comparison to other detectors
- Comparison to concurrency testing tools
- Scalability analysis
Evaluation

- Detection effectiveness
- Contribution of techniques to reducing overhead
- Breakdown of overhead
- Comparison to other detectors
- Comparison to concurrency testing tools
- Scalability analysis
RaceMob detected and confirmed 106 data races
How Does Each Technique Lower the Overhead?

- Dynamic detection
- Static + dynamic detection
- RaceMob (static + dynamic + DCI + on-demand validation) aggregate
- RaceMob per-user

All techniques are required for low overhead.
Breakdown of overhead

2.3% average runtime overhead per-user
RaceMob

Detected 106 real races in 10 programs with 2.3% average overhead