A True Positives Theorem for a Static Race Detector

Nikos Gorogiannis, Peter O’Hearn, Ilya Sergey

The purpose of this paper is to state and prove a theorem that has come about by reacting to surprising properties we observed of a static program analysis that has been in production at Facebook for over a year.

The *RacerD* program analyzer searches for data races in Java programs, and it has had significantly more reported industrial impact than any other concurrency analysis that we are aware of. It was released as open source in October of 2017, and the OOPSLA’18 paper by Blackshear et al. (2018) describes its design, and gives more details about its deployment. They report, for example, that over 2,500 concurrent data races found by RacerD have been fixed by Facebook developers, and that it has been used to support the conversion of Facebook’s Android app rendering infrastructure from a single-threaded to a multi-threaded architecture.

The theorem was motivated in the first case by the desire to understand the observation from production that RacerD was providing remarkably accurate signal to developers, and then the theorem guided further analyzer design decisions. Technically, our result can be seen as saying that the analysis computes an under-approximation of an over-approximation, which is the reverse of the more usual (over of under) situation in static analysis. Until now, static analyzers that are effective in practice but unsound have often been regarded as ad hoc; in contrast, we suggest that, in the future, theorems of this variety might be generally useful in understanding, justifying and designing effective static analyses for bug catching.

CCS Concepts:
- Theory of computation → Program analysis
- Software and its engineering → Concurrent programming structures

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom, Abstract Interpretation
Unsound (and incomplete) static analyses can be *principled*, satisfying meaningful theorems that help to understand their behaviour and guide their design.

One can have an unsound but effective static analysis, which has significant industrial impact, and which is supported by a *meaningful theorem*.
Context

1. We had a demonstrably-effective industrial analysis: RacerD (OOPSLA'18); >3k fixes in Facebook Java codebase

2. No soundness theorem
Static Analyses for Bug Detection

Infer
Slither
Eclipse
ESC/Java
Security ErrorProne
FindBugs
Resharper
Context

1. We had a demonstrably-effective industrial analysis: RacerD (OOPSLA'18); >3k fixes in Facebook Java
2. No soundness theorem
3. Architecture: compositional abstract interpreter
4. No heuristic alarm filtering

Just ad hoc?

Our reaction:
Semantics/theory should understand/explain, not lecture.
Conjecture

True Positives Theorem:

Under certain assumptions, the static bug detector reports no false positives.
Static Analyses for Program Validation
The Essence of Static Analysis

“abstraction”

\[ \alpha \]

\[ e \rightarrow p \]

program execution

property of interest
Diagram showing two entities, $e_1$ and $e_2$, connected to a single entity, $p$, with two arrows labeled $\alpha$. The diagram represents a flow or relationship between the entities and the central entity $p$. 
Static Analysis

\[ \text{concreteSem}(c) = e_1 \rightarrow p_1 \quad e_2 \rightarrow p_2 \quad e_3 \rightarrow p_3 \quad e_4 \rightarrow p_4 \]
Static Analysis

concreteSem(c) =

\{ p_1 \}
\{ p_2 \}
\{ p_3 \}
\{ p_4 \}

“has bugs”
“correct”
Verifier or a Bug Detector?
Program Verifier

true positive
false positive
true negative
true negative
Sound Program Verifier

true positive
false positive
true negative
true negative
Sound Program Verifier

abstract over-approximation
Sound Program Verifier

abstract over-approximation

true positive
false positive
true negative
true negative
Sound Program Verifier

Developer:
Go away, that never happens!

```
if (n == VERY_UNLIKELY_VALUE) {
  bug.explode();
} else {
  // do nothing
}
```
Unsound Program “Verifier”

```java
if (n == VERY_UNLIKELY_VALUE) {
    bug.explode();
} else {
    // do nothing
}
```

- true positive: $p_1$
- false positive: $p_2$
- true negative: $p_3$
- false negative: $p_4$
“Sound” Program Verifier

true positive
false positive
true negative
false negative
“Sound” Program Verifier

concrete under-approximation

abstract over-approximation

true positive
false positive
true negative
Sound Static Verifiers

- False negatives (bugs missed) are **bad**

- False positives (non-bugs reported) are **okay**

- Constructed as **over-approximation** (of **under-approximation**)

- **Soundness Theorem:**
  Under certain assumptions about the programs, the analyser has no **false negatives**.
Static Bug Finder

- True positive: $e_1$
- False positive: $e_2$
- True negative: $e_3$
- False negative: $e_4$

$p_1$: true positive
$p_2$: false positive
$p_3$: true negative
$p_4$: false negative
Unsound Static Bug Finder

true positive
false positive
true negative
false negative
Sound (but imprecise) Static Bug Finder

abstract under-approximation

true positive
false negative
true negative
false negative
if (n != VERY_UNLIKELY_VALUE) {
  e2 // bug happens here
} else {
  e3 // normal execution
}

Idea: over-approximate in concrete semantics!
Sound (but Imprecise) Static Bug Finder

Let's merge these executions into one that subsumes both!

- $e_1$ (true positive)
- $e_2$ (false negative)
- $e_3$ (true negative)
- $e_4$ (false negative)
- $e_5$ (false negative)
- $e_6$ (false negative)
overApproxConcreteSem(c) =

\[
\begin{align*}
&\text{if } (*) \{ \\
&\quad \text{// bug happens here} \\
&\} \text{ else } \{ \\
&\quad \text{// normal execution} \\
&\}
\end{align*}
\]
Sound Static Bug Finder

```java
if (*) {
    // bug happens here
} else {
    // normal execution
}
```

overApproxConcreteSem(c) =

concrete over-approximation

abstract under-approximation

true positive

true negative

false negative
Towards Sound Static Bug Finders

• False negatives (bugs missed) are *okay*

• False positives (non-bugs reported) are *bad*

• Constructed as *under-approximation* of *over-approximation*

• **Soundness (True Positives) Theorem:**
  Under certain assumptions about the programs, the analyser has *no false positives.*
A Recipe for True Positives Theorem

1. **Over-approximate** semantic elements to make up for “difficult” dynamic execution aspects
   
   **Example:** replace **conditions** and **loops** with their **non-deterministic** versions

2. Pick abstraction $\alpha$ for over-approximated executions that **provably identifies** “buggy” behaviours:
   
   \[ \forall e: \text{execution}, \ \text{hasBug}(\alpha(e)) \Rightarrow \text{execution e has a bug} \]

3. Design an abstract semantics $\text{asem}$, so it is **complete** wrt. $\alpha$ and **over-approximated** concrete semantics:
   
   \[ \forall c: \text{program}, \ \text{asem}(c) = \alpha(\text{overApproxConcreteSem}(c)) \]

4. Together, $\text{asem}$ and $\text{hasBug}$ provide a **TP-sound** static bug finder.
Case Study: RacerDX

- A provably TP-Sound version of Facebook’s RacerD concurrency analyser (Blackshear et al., OOPSLA’18)

- **Buggy executions**: data races in lock-based concurrent programs

- **Syntactic assumptions**: Java programs with well-scoped locking (synchronised), no recursion, reflection, dynamic class loading; global variables are ignored.

- **Concrete over-approximation**: Loops and conditionals are non-deterministic.
A True Race

class Bloop {
    public int f = 1;
}

class Burble {
    public void meps(Bloop b) {
        synchronized (this) {
            System.out.println(b.f);
        }
    }

    public void reps(Bloop b) {
        b.f = 42;
    }

    public void beps(Bloop b) {
        b = new Bloop();
        b.f = 239;
    }
}
Path prefix \texttt{b} is "\textit{unstable}" ("\textit{wobbly}") as it’s reassigned, hence race is evaded.
Complete Abstraction for Race Detection

- \( asem(\text{meps}(b)) = (\{b.f\}, 0, \{R(b.f, 1)\}) \)
- \( asem(\text{reps}(b)) = (\{b.f\}, 0, \{W(b.f, 0)\}) \)
- \( asem(\text{beps}(b)) = (\{b, b.f\}, 0, \{W(b, 0), W(b.f, 0)\}) \)

\((W, L, A)\)

“Wobbly” paths, touched during execution

Accesses/locks with formals/fields

Locking level

\[ (\text{beps}(b), \text{reps}(b), \text{meps}(b)) = (\{b, b.f\}, 0, \{W(b, 0), W(b.f, 0)\}) \]

\( \text{Locking level} \)

\( \text{Accesses/locks with formals/fields} \)

\( \text{“Wobbly” paths, touched during execution} \)

\( \text{class Burble} \)

\begin{verbatim}
  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
\end{verbatim}
Analysing Summaries for Races

- \( \text{asem}(\text{meps}(b)) = (\{b.f\}, 0, \{R(b.f, 1)\}) \)
- \( \text{asem}(\text{reps}(b)) = (\{b.f\}, 0, \{W(b.f, 0)\}) \)
- \( \text{asem}(\text{beps}(b)) = (\{b, b.f\}, 0, \{W(b, 0), W(b.f, 0)\}) \)

\( \text{meps}(b) \text{ or } \text{reps}(b) \Rightarrow \text{Can race, report a bug!} \)

class Burble {
    public void meps(Bloop b) {
        synchronized (this) {
            System.out.println(b.f);
        }
    }
    public void reps(Bloop b) {
        b.f = 42;
    }
    public void beps(Bloop b) {
        b = new Bloop();
        b.f = 239;
    }
}
Analysing Summaries for Races

- \( \text{asem}(meps(b)) = (\{b.f\}, 0, \{R(b.f, 1)\}) \)
- \( \text{asem}(reps(b)) = (\{b.f\}, 0, \{W(b.f, 0)\}) \)
- \( \text{asem}(beps(b)) = (\{b, b.f\}, 0, \{W(b, 0), W(b.f, 0)\}) \)

\[ \text{meps}(b) \ || \ | \text{beps}(b) \implies \text{Maybe don't race, don't report a bug} \]
Formal Result

RacerDX enjoys the True Positives Theorem \textit{wrt.} Data Race Detection

(Details in the paper)
Evaluation

What is the price to pay for having the TP Theorem?

(Reporting no bugs whatsoever is TP-Sound)
RacerD vs RacerDX

<table>
<thead>
<tr>
<th>Target</th>
<th>LOC</th>
<th>D CPU</th>
<th>DX CPU</th>
<th>CPU ±%</th>
<th>D Reps</th>
<th>DX Reps</th>
<th>Reps ±%</th>
</tr>
</thead>
<tbody>
<tr>
<td>avroroa</td>
<td>76k</td>
<td>103</td>
<td>102</td>
<td>0.4%</td>
<td>143</td>
<td>92</td>
<td>36%</td>
</tr>
<tr>
<td>Chronicle-Map</td>
<td>45k</td>
<td>196</td>
<td>196</td>
<td>0.1%</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>jvm-tools</td>
<td>33k</td>
<td>106</td>
<td>109</td>
<td>-3.6%</td>
<td>30</td>
<td>26</td>
<td>13%</td>
</tr>
<tr>
<td>RxJava</td>
<td>273k</td>
<td>76</td>
<td>69</td>
<td>9.2%</td>
<td>166</td>
<td>134</td>
<td>19%</td>
</tr>
<tr>
<td>sunflow</td>
<td>25k</td>
<td>44</td>
<td>44</td>
<td>-1.4%</td>
<td>97</td>
<td>42</td>
<td>57%</td>
</tr>
<tr>
<td>xalan-j</td>
<td>175k</td>
<td>144</td>
<td>137</td>
<td>5.0%</td>
<td>326</td>
<td>295</td>
<td>10%</td>
</tr>
</tbody>
</table>

We can make the following observations:

- The difference between runtimes is largely within the noise margins, especially given that a large percentage of these runtimes is spent compiling Java source into bytecode, as Infer extracts an AST from the compiled artefact.
- The loss in terms of number of reports ranges between 10% and 57% (we exclude Chronicle-Map as there are too few reports to start with), and the loss in terms of number of distinct access paths ranges from 21% to 67%.

### 7.2.2 The Causes for Deterioration of Reporting Rate.

We triaged a sample of reports that RacerD made but RacerDX didn’t. We discerned two main classes of reports:

- In a call `this.this.f()`, the check whether one of the actuals is a proper prefix of another (Figure 7, method call case) fails because `this ≺ this.f`. Thus, this is marked as unstable (is added to the `W` component of the abstract state). But this means that all accesses in the caller method will not be considered in data race reports, leading to potential false negatives. A potential solution here may be to use the fact that the first argument of a non-static Java method cannot be reassigned, and thus may be left out of the check above, but we have not at present assessed how this might affect the status of our theorems.

- Inner (nested) classes are common in Java, and allow methods of an inner class object to reference fields and methods of the containing class. To achieve this, the compiler inserts in the inner class a hidden reference to the outer class object, and initialises this appropriately at construction. Unfortunately, this also means that the `this` reference of the outer class is marked as unstable whenever an inner class object is constructed, thus precluding accesses occurring in the enclosing method from being reported.

Remarkably, all the reports we triaged were true positives. Both of these classes of missing reports may benefit from elaborating the stability abstract domain to track escaping references, i.e., when a path is read it is not immediately marked as unstable, but only when the address read ends up being stored somewhere. This is something we will investigate in further work.
## RacerD vs RacerDX

### Evaluation: target projects and statistics.

<table>
<thead>
<tr>
<th>Target</th>
<th>LOC</th>
<th>D CPU</th>
<th>DX CPU</th>
<th>CPU ±%</th>
<th>D Reps</th>
<th>DX Reps</th>
<th>Reps ±%</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>76k</td>
<td>103</td>
<td>102</td>
<td>0.4%</td>
<td>143</td>
<td>92</td>
<td>36%</td>
</tr>
<tr>
<td>Chronicle-Map</td>
<td>45k</td>
<td>196</td>
<td>196</td>
<td>0.1%</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>jvm-tools</td>
<td>33k</td>
<td>106</td>
<td>109</td>
<td>-3.6%</td>
<td>30</td>
<td>26</td>
<td>13%</td>
</tr>
<tr>
<td>RxJava</td>
<td>273k</td>
<td>76</td>
<td>69</td>
<td>9.2%</td>
<td>166</td>
<td>134</td>
<td>19%</td>
</tr>
<tr>
<td>sunflow</td>
<td>25k</td>
<td>44</td>
<td>44</td>
<td>-1.4%</td>
<td>97</td>
<td>42</td>
<td>57%</td>
</tr>
<tr>
<td>xalan-j</td>
<td>175k</td>
<td>144</td>
<td>137</td>
<td>5.0%</td>
<td>326</td>
<td>295</td>
<td>10%</td>
</tr>
</tbody>
</table>
# RacerD vs RacerDX

## Evaluation: target projects and statistics

<table>
<thead>
<tr>
<th>Target</th>
<th>LOC</th>
<th>D CPU</th>
<th>DX CPU</th>
<th>CPU ±%</th>
<th>D Reps</th>
<th>DX Reps</th>
<th>Reps ±%</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>76k</td>
<td>103</td>
<td>102</td>
<td>0.4%</td>
<td>143</td>
<td>92</td>
<td>36%</td>
</tr>
<tr>
<td>Chronicle-Map</td>
<td>45k</td>
<td>196</td>
<td>196</td>
<td>0.1%</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>jvm-tools</td>
<td>33k</td>
<td>106</td>
<td>109</td>
<td>-3.6%</td>
<td>30</td>
<td>26</td>
<td>13%</td>
</tr>
<tr>
<td>RxJava</td>
<td>273k</td>
<td>76</td>
<td>69</td>
<td>9.2%</td>
<td>166</td>
<td>134</td>
<td>19%</td>
</tr>
<tr>
<td>sunflow</td>
<td>25k</td>
<td>44</td>
<td>44</td>
<td>-1.4%</td>
<td>97</td>
<td>42</td>
<td>57%</td>
</tr>
<tr>
<td>xalan-j</td>
<td>175k</td>
<td>144</td>
<td>137</td>
<td>5.0%</td>
<td>326</td>
<td>295</td>
<td>10%</td>
</tr>
</tbody>
</table>

## Evaluation results

CPU columns are in seconds; Reps are distinct reports; π are distinct paths.

We can make the following observations:

- The difference between runtimes is largely within the noise margins, especially given that a large percentage of these runtimes is spent compiling Java source into bytecode, as Infer extracts an AST from the compiled artefact.
- The loss in terms of number of reports ranges between 10% and 57% (we exclude Chronicle-Map as there are too few reports to start with), and the loss in terms of number of distinct access paths ranges from 21% to 67%.

### 7.2.2 The Causes for Deterioration of Reporting Rate

We triaged a sample of reports that RacerD made but RacerDX didn’t. We discerned two main classes of reports:

- In a call `this`foo(`this`.f), the check whether one of the actuals is a proper prefix of another fails because `this` ≺ `this`.f. Thus, `this` is marked as unstable (is added to the W component of the abstract state). But this means that all accesses in the caller method will not be considered in data race reports, leading to potential false negatives. A potential solution here may be to use the fact that the first argument of a non-static Java method cannot be reassigned, and thus may be left out of the check above, but we have not at present assessed how this might affect the status of our theorems.
- Inner (nested) classes are common in Java, and allow methods of an inner class object to reference fields and methods of the containing class. To achieve this, the compiler inserts in the inner class a hidden reference to the outer class object, and initialises this appropriately at construction. Unfortunately, this also means that the `this` reference of the outer class is marked as unstable whenever an inner class object is constructed, thus precluding accesses occurring in the enclosing method from being reported. Remarkably, all the reports we triaged were true positives. Both of these classes of missing reports may benefit from elaborating the stability abstract domain to track escaping references, i.e., when a path is read it is not immediately marked as unstable, but only when the address read ends up being stored somewhere. This is something we will investigate in further work.
To Take Away: Theory

- A True Positive-Sound static bug finder never reports false positives. It can be designed as an under-approximation of an over-approximation.

- An abstraction \( \alpha \) for TP-Sound static bug detection can be very simple, but it has to be complete (i.e., sufficient) to report bugs.
To Take Away: Practice

• **RacerDX** is **TP-Sound race detector**, whose precision and performance are comparable with Facebook’s RacerD (Blackshear et al., OOPSLA’18)

• If **RacerDX** had been deployed initially rather than **RacerD**, it would have found 1000s of bugs, far outstripping all *reported impact* in previous concurrency analyses (counterfactual reasoning)

• Until now, static analysers for bug catching that are effective in practice but unsound have often been regarded as *ad hoc*; in the future, they can be *principled, satisfying theorems* to inform and guide their designs.