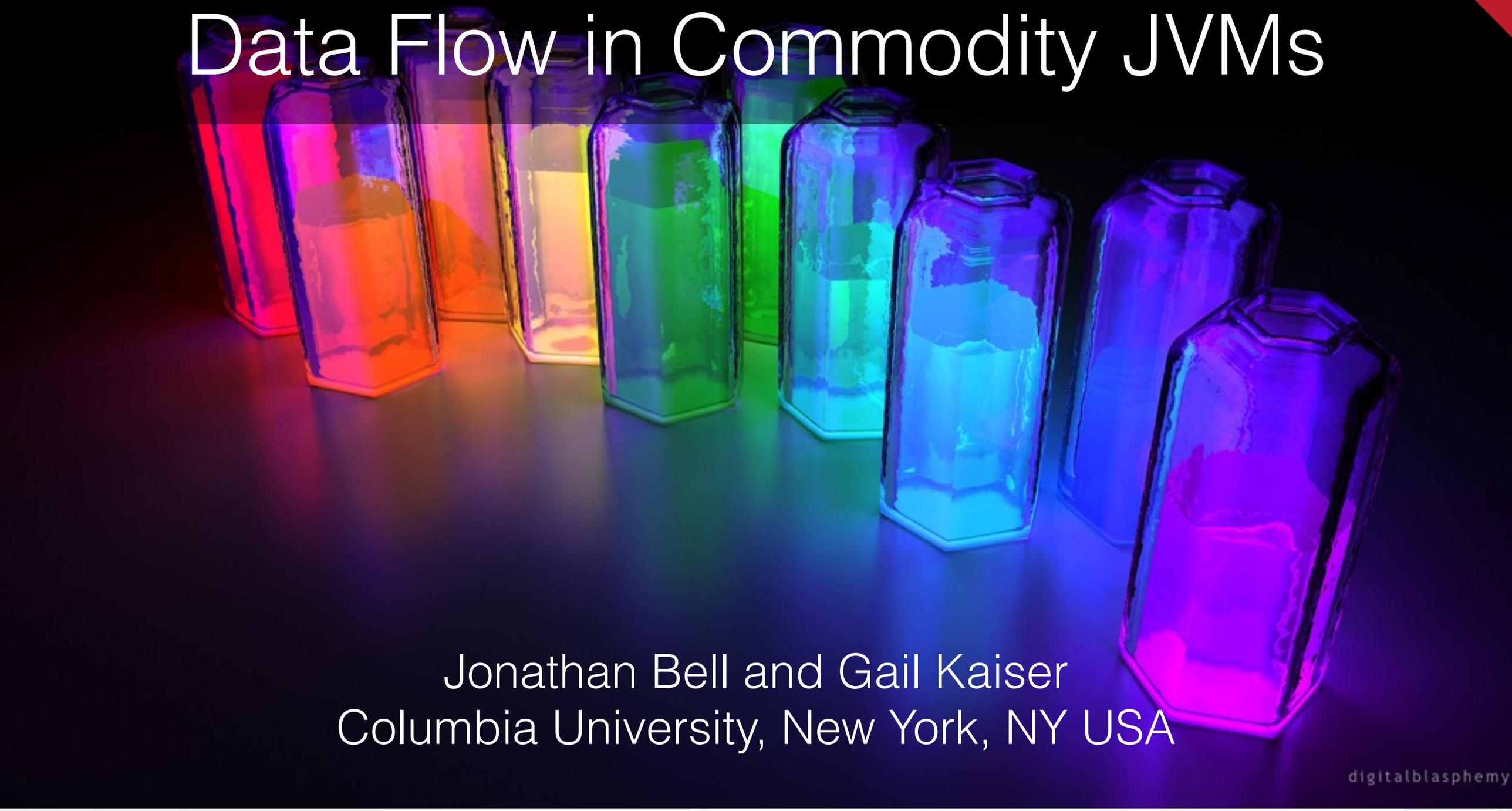
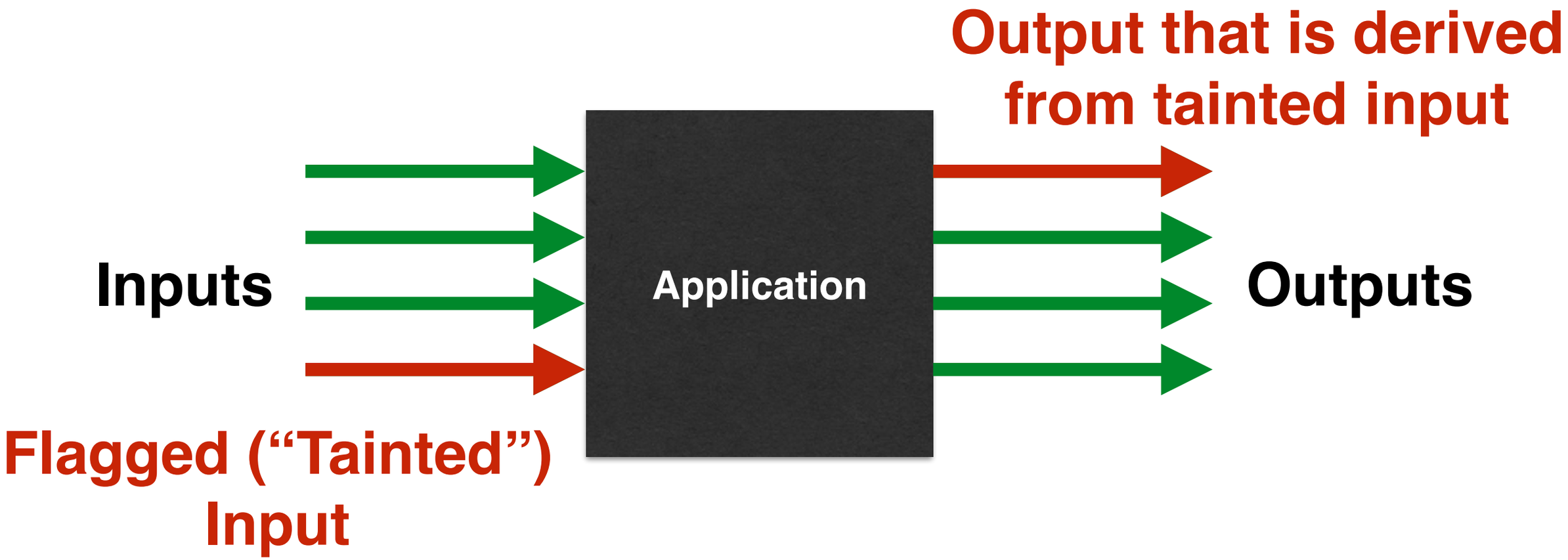
Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs



Jonathan Bell and Gail Kaiser
Columbia University, New York, NY USA

Dynamic Data Flow Analysis: Taint Tracking



Taint Tracking: Applications

- End-user privacy testing: Does this application send my personal data to remote servers?
- SQL injection attack avoidance: Do SQL queries contain raw user input
- Debugging: Which inputs are relevant to the current (crashed) application state?

Qualities of a Successful Analysis

Soundness

No data leakage!


Precision
Data is tracked with the right tag

Performance

Minimal slowdowns

Portability

No special hardware, OS, or JVM

“Normal” Taint Tracking

- Associate tags with data, then propagate the tags
- Approaches:
  - Operating System modifications [Vandebogart ’07], [Zeldovich ’06]
  - Language interpreter modifications [Chandra ’07], [Enck ’10], [Nair ’07], [Son ’13]
  - Source code modifications [Lam ‘06], [Xu ’06]

  Not portable
  - Binary instrumentation of applications [Clause ’07], [Cheng ’06], [Kemerlis ’12]

Hard to be sound, precise, and performant

Phosphor

- Leverages benefits of interpreter-based approaches (information about variables) but *fully portably*

- Instruments *all* byte code that runs in the JVM (including the JRE API) to track taint tags

  - Add a variable for each variable

  - Adds propagation logic

Key contribution:

How do we efficiently store meta-data for **every** variable **without** modifying the JVM itself?

JVM Type Organization

- Primitive Types
  - int, long, char, byte, etc.
- Reference Types
  - Arrays, instances of classes
  - All reference types are assignable to java.lang.Object

Phosphor's taint tag storage

	Local variable	Method argument	Return value	Operand stack	Field
Object	Stored as a field of the object				
Object array	Stored as a field of each object				
Primitive	Shadow variable	Shadow argument	"Boxed"	Below the value on stack	Shadow field
Primitive array	Shadow array variable	Shadow array argument	"Boxed"	Array below value on stack	Shadow array field

Taint Propagation

- Modify all byte code instructions to be taint-aware by adding extra instructions

- Examples:
  - Arithmetic -> combine tags of inputs
  - Load variable to stack -> Also load taint tag to stack
  - Modify method calls to pass taint tags


HERE BE DRAGONS
MARE GLACIALE
PARS
Two big problems

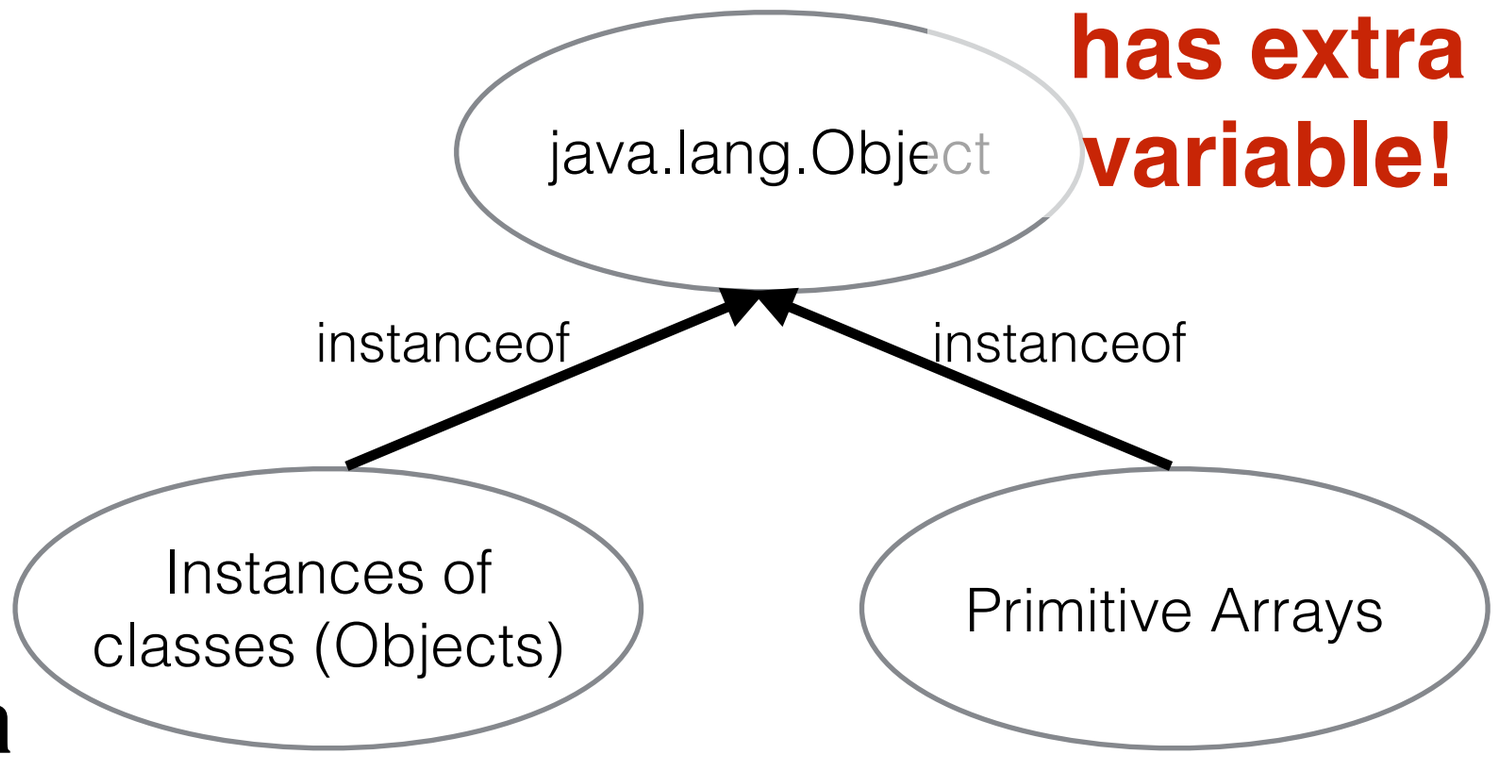
Challenge 1: Type Mayhem

Primitive Types

Always has extra variable!

Never has extra variable!

Sometimes has extra variable!



Always has extra variable

Challenge 1: Type Mayhem

```
byte[] array = new byte[5];
Object ret = array;
return ret;
```

```
int[] array_tag = new int[5];
byte[] array = new byte[5];

Object ret = new TaintedByteArray(array_tag,array);
```

Solution 1: Box taint tag with array when we lose type information

Challenge 2: Native Code

We can’t instrument everything!

Challenge 2: Native Code

```
public int hashCode() {
  return super.hashCode() * field.hashCode();
}
```

```
public native int hashCode();
```

Solution: Wrappers. Rename *every* method, and leave a wrapper behind

Challenge 2: Native Code

```
public int hashCode() {
  return super.hashCode() * field.hashCode();
}
```

```
public native int hashCode();
```

Solution: Wrappers. Rename *every* method, and leave a wrapper behind

```
public TaintedInt hashCode$$wrapper() {
    return new TaintedInt(0, hashCode());
  }
```

Challenge 2: Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
```

Challenge 2: Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
```

```
public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
  //The original method "someMethod", but with taint tracking
}
```

Challenge 2: Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
        {
          return someMethod$$wrapper(0, in).val;
        }

public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
  //The original method "someMethod", but with taint tracking
}
```

Challenge 2: Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
        {
          return someMethod$$wrapper(0, in).val;
        }

public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
  //The original method "someMethod", but with taint tracking
}
```

Design Limitations

- Tracking through native code
  - Return value's tag becomes combination of all parameters (heuristic); not found to be a problem in our evaluation
- Tracks explicit data flow only (not through control flow)

Evaluation

- Soundness & Precision
- Performance
- Portability

Soundness & Precision

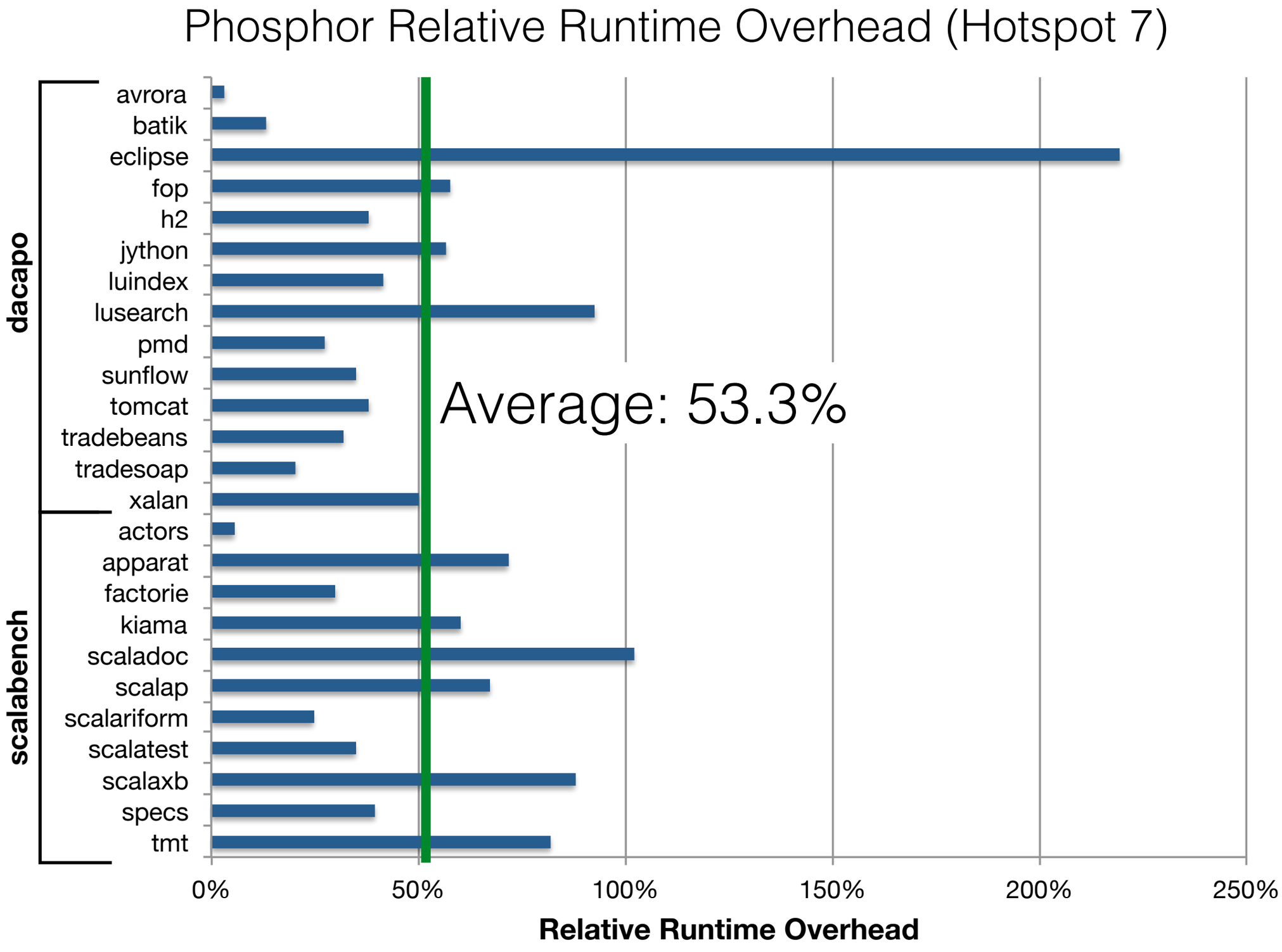
- DroidBench - series of unit tests for Java taint tracking

- Passed all except for implicit flows (intended behavior)

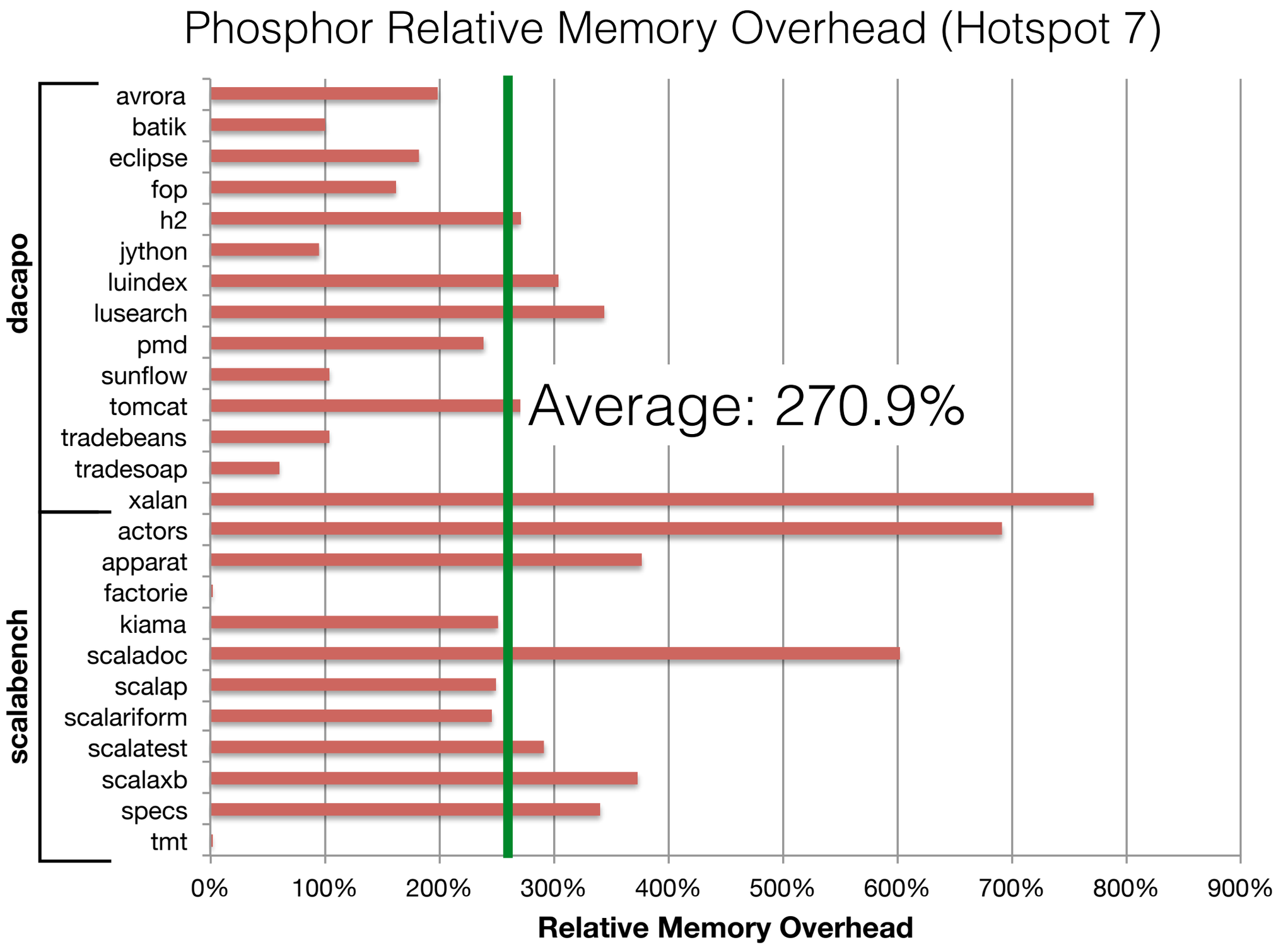
Performance

- Macrobenchmarks (DaCapo, Scalabench)
- Microbenchmarks
  - Versus TaintDroid [Enck, 2010] on CaffeineMark
  - Versus Trishul [Nair, 2008] on JavaGrande

Macrobenchmarks

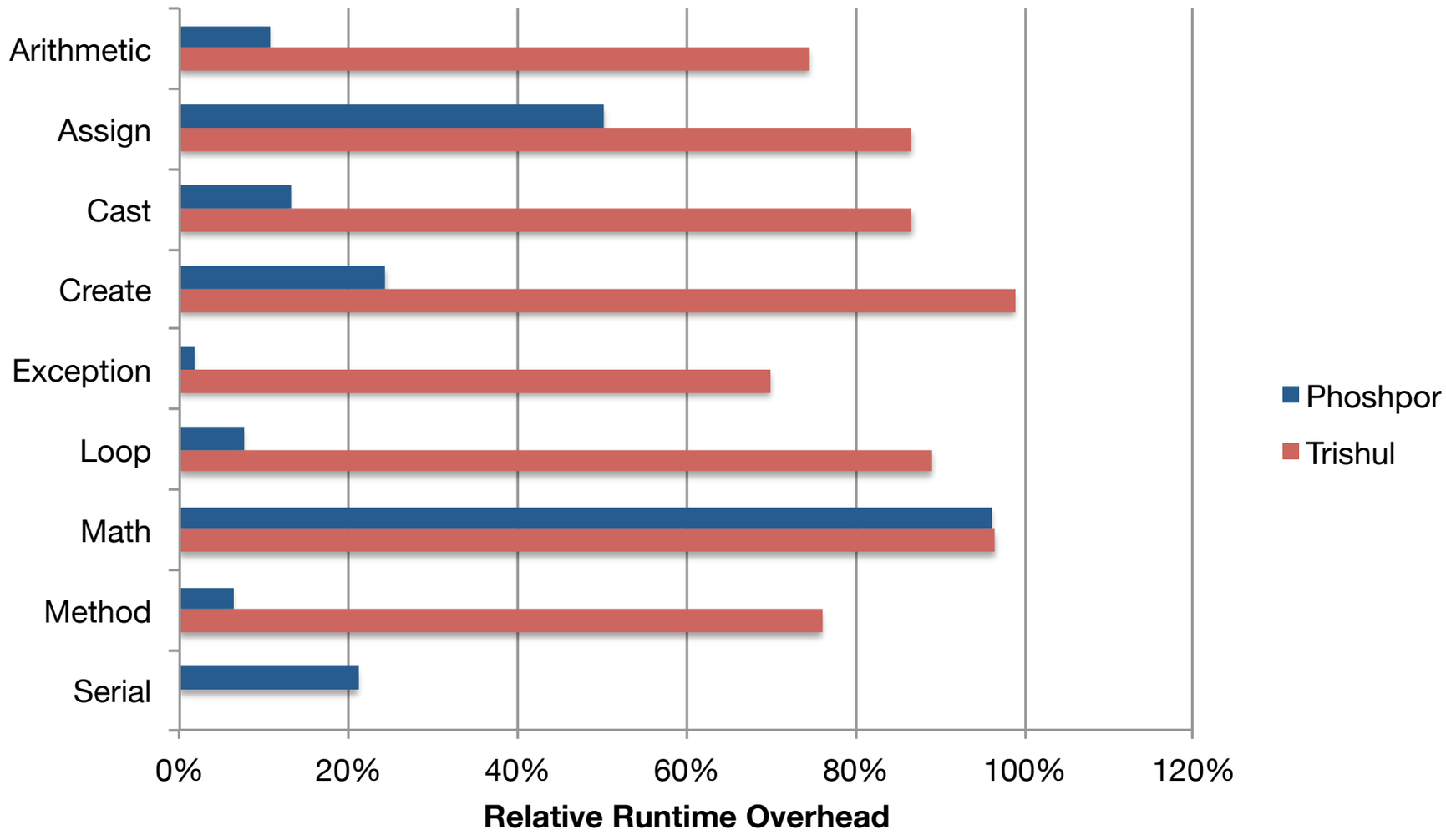


Macrobenchmarks



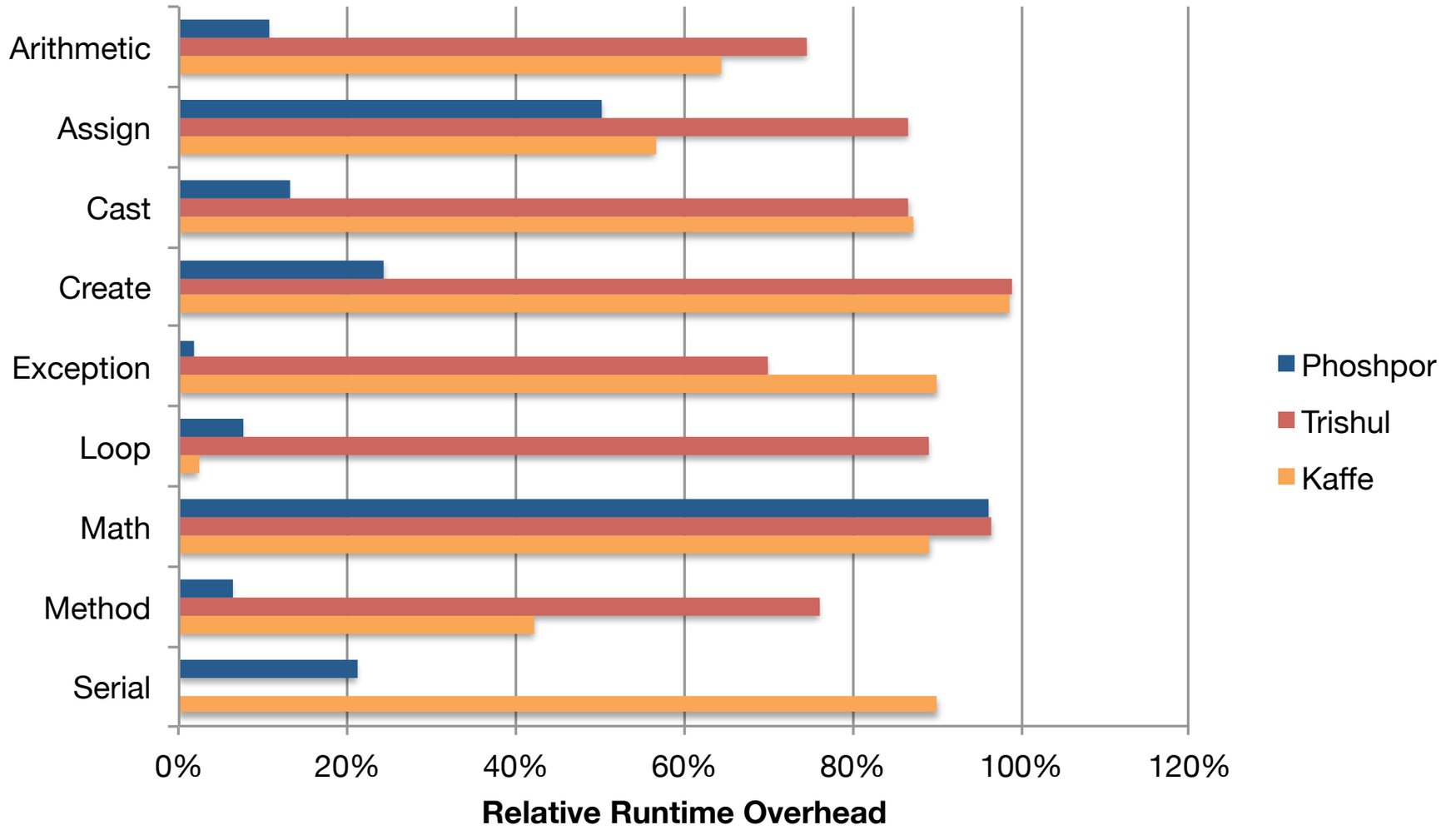
Microbenchmarks

Phosphor (Hotspot 7) and Trishul Relative Overhead



Microbenchmarks

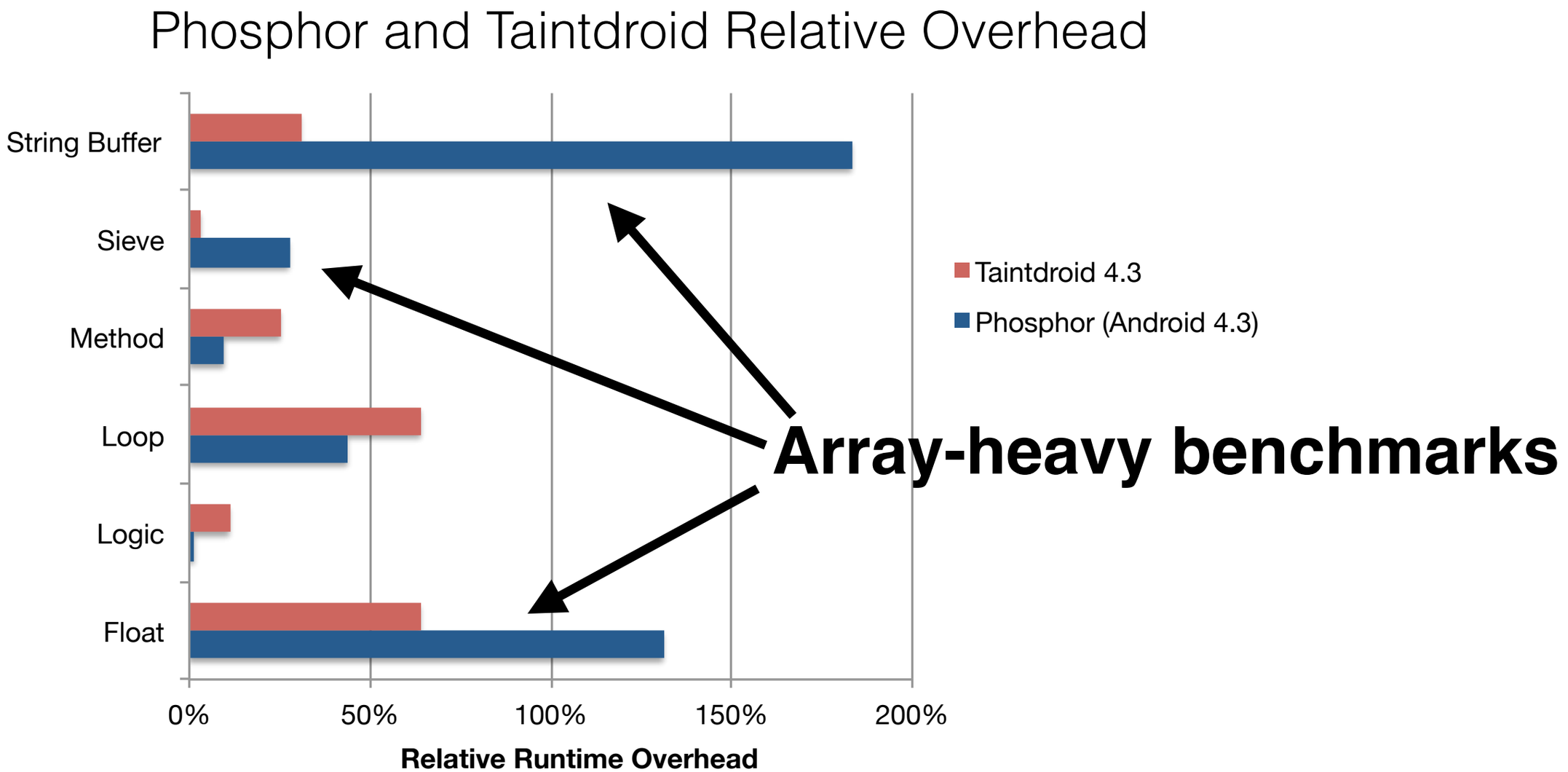
Phosphor (Hotspot 7) and Trishul Relative Overhead



Microbenchmarks: Taintdroid

- Taintdroid: Taint tracking for Android's Dalvik VM [Enck, 2010]
  - Not very precise: one tag per array (not per array element!)
  - Applied Phosphor to Android!

Microbenchmarks



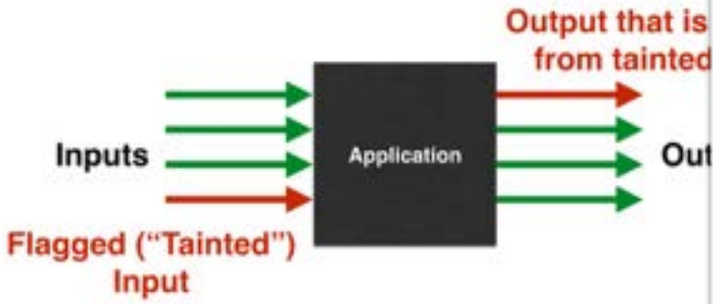
Portability

JVM	Version(s)	Success?
Oracle (Hotspot)	1.7.0_45, 1.8.0_0	**Yes**
OpenJDK	1.7.0_45, 1.8.0_0	**Yes**
Android Dalvik	4.3.1	**Yes**
Apache Harmony	6.0M3	**Yes**
Kaffe VM	1.1.9	**Yes**
Jikes RVM	3.1.3	No, but may be possible with more work

Future Work & Extension

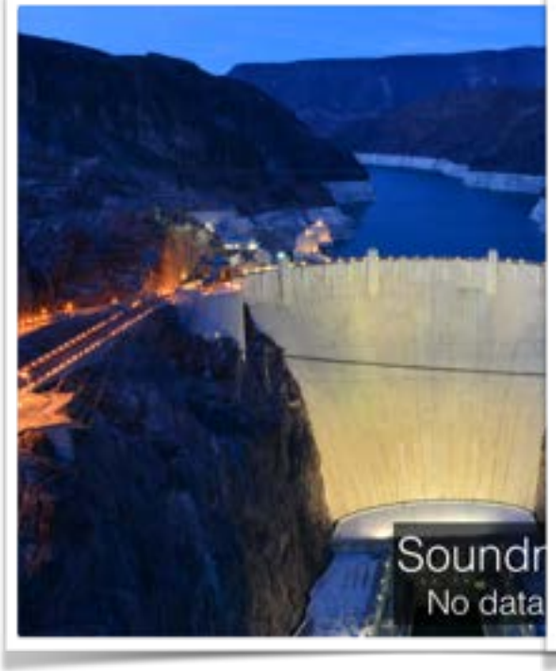
- This is a general approach for tracking metadata with variables in unmodified JVMs
- Could track any sort of data in principle
- We have already extended this approach to track path constraints on inputs

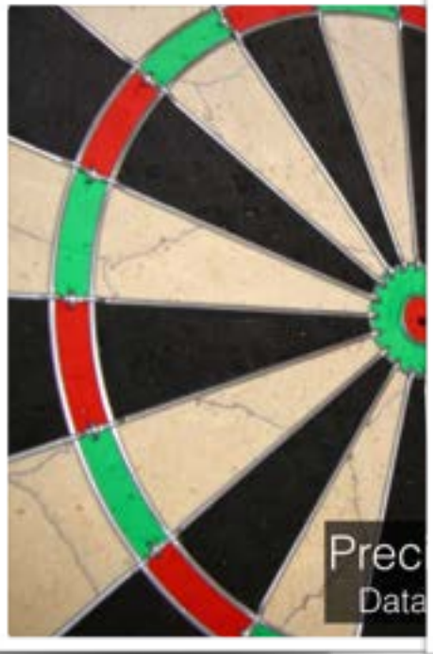
Dynamic Data Flow Analys
Taint Tracking



Phosphor's taint tag stor

	Local variable	Method argument	Return value	Operand stack	Fiel
Object	Stored as a field of the object				
Object array	Stored as a field of each object				
Primitive	Shadow variable	Shadow argument	"Boxed"	Below the value on stack	Shad fiel
Primitive array	Shadow array variable	Shadow array argument	"Boxed"	Array below value on stack	Shac array





Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs

Jonathan Bell and Gail Kaiser
Columbia University
jbell@cs.columbia.edu @_jon_bell_

https://github.com/Programming-Systems-Lab/Phosphor