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

- Inputs
- Flagged ("Tainted") Input
- Application
- Outputs
- Output that is derived from tainted input
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 [Vandeboogart ’07], [Zeldovich ’06]
  • Language interpreter modifications [Chandra ’07], [Enck ’10], [Nair ’07], [Son ’13]
  • Source code modifications [Lam ’06], [Xu ’06]
  • 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

<table>
<thead>
<tr>
<th></th>
<th>Local variable</th>
<th>Method argument</th>
<th>Return value</th>
<th>Operand stack</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stored as a field of the object</td>
</tr>
<tr>
<td><strong>Object array</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stored as a field of each object</td>
</tr>
<tr>
<td><strong>Primitive</strong></td>
<td>Shadow variable</td>
<td>Shadow argument</td>
<td>&quot;Boxed&quot;</td>
<td>Below the value on stack</td>
<td>Shadow field</td>
</tr>
<tr>
<td><strong>Primitive array</strong></td>
<td>Shadow array variable</td>
<td>Shadow array argument</td>
<td>&quot;Boxed&quot;</td>
<td>Array below value on stack</td>
<td>Shadow array field</td>
</tr>
</tbody>
</table>
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
Two big problems

HERE BE DRAGONS

Two big problems
Challenge 1: Type Mayhem

- **Primitive Types**
  - Always has extra variable!

- **java.lang.Object**
  - Sometimes has extra variable!

- **Instances of classes (Objects)**
  - Never has extra variable!

- **Primitive Arrays**
  - Always has extra variable!
Challenge 1: Type Mayhem

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

```java
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();
}

Solution: Wrappers. Rename every method, and leave a wrapper behind
Challenge 2: Native Code

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

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

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

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

```java
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

```java
public int[] someMethod(byte in)
```
Challenge 2: Native Code

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

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

```java
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

```java
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

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

```java
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

Phosphor Relative Runtime Overhead (Hotspot 7)

Average: 53.3%
Relative Memory Overhead

Phosphor Relative Memory Overhead (Hotspot 7)

Average: 270.9%
Microbenchmarks

Phosphor (Hotspot 7) and Trishul Relative Overhead

Relative Runtime Overhead

Arithmetic
Assign
Cast
Create
Exception
Loop
Math
Method
Serial

Phosphor
Trishul
Microbenchmarks

Phosphor (Hotspot 7) and Trishul Relative Overhead

Relative Runtime Overhead

- Arithmetic
- Assign
- Cast
- Create
- Exception
- Loop
- Math
- Method
- Serial

Phosphor
Trishul
Kaffe
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

Phosphor and Taintdroid Relative Overhead

Array-heavy benchmarks
## Portability

<table>
<thead>
<tr>
<th>JVM</th>
<th>Version(s)</th>
<th>Success?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (Hotspot)</td>
<td>1.7.0_45, 1.8.0_0</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>1.7.0_45, 1.8.0_0</td>
<td>Yes</td>
</tr>
<tr>
<td>Android Dalvik</td>
<td>4.3.1</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache Harmony</td>
<td>6.0M3</td>
<td>Yes</td>
</tr>
<tr>
<td>Kaffe VM</td>
<td>1.1.9</td>
<td>Yes</td>
</tr>
<tr>
<td>Jikes RVM</td>
<td>3.1.3</td>
<td>No, but may be possible with more work</td>
</tr>
</tbody>
</table>
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 Analysis
Taint Tracking

Inputs
Flagged ("Tainted") Input

Application

Output that is not from tainted

Phosphor's taint tag store

<table>
<thead>
<tr>
<th>Local variable</th>
<th>Method argument</th>
<th>Return value</th>
<th>Operand stack</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Stored as a field of the object</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object array</td>
<td>Stored as a field of each object</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primitive</td>
<td>Shadow variable</td>
<td>Shadow argument</td>
<td>&quot;Boxed&quot;</td>
<td>Below the value on stack</td>
</tr>
<tr>
<td>Primitive array</td>
<td>Shadow array variable</td>
<td>Shadow array argument</td>
<td>&quot;Boxed&quot;</td>
<td>Array below value on stack</td>
</tr>
</tbody>
</table>

Challenges

Sound
No data

Precis
No data

Portability
No special hardware, OS, or JVM

OOPSLA 2014

@_jon_bell_

October 22, 2014
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