A Beginner's Notes on Phosphor

José Cambronero

October 29, 2017

Abstract

This document contains my personal set of notes for building and running examples with Jonathan Bell's Phosphor[**?**]. This should be viewed as a beginners guide (as I am writing it as I learn to use the tool myself.) Any errors or omissions here are my own.

1   Installation/Building

We begin by cloning the github repository for the project.

```
$ git clone git@github.com:Programming-Systems-Lab/phosphor.git
```

Phosphor is a maven project, so to build you can call `mvn package`. As per the documentation Phosphor has been built and tested with both versions 7/8 of Oracle's Hotspot JVM and OpenJDK's IcedTea JVM. If you run into issues with your java version, I suggest installing jenv, which allows you to manage multiple versions of the JVM on your machine cleanly and transparently.

```
$ jenv local 1.8.0.66
$ mvn package
```

You should obtain a message similar to the one in Figure 1 if your build was succesful

Now, we need to instrument our JRE before we can run any of our examples. This can be done as follows

Figure 1: Succesful build message

1

```
$ cd target/
$ java −jar Phosphor−0.0.2−SNAPSHOT.jar \
 /Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/
    jre/ jre−inst/
```

This has created an instrumented JRE suitable for data flow tracking and integer tags combined with OR.

The only thing left to do is make sure we change permissions for all binaries in the instrumented JRE.

```
$ chmod +x jre−inst/bin/*
```

2 Testing

In order to verify that your build works, you can run the `mvn verify` command, as explained in the documentation. You should expect this to take a bit, as it generates different instrumented JREs and tests them. If all is succesful, you will obtain a Maven log message printed to stdout indicating so.

3 Usage

3.1 Automatic

TODO: this remains to be experimented with
 TODO: taintSinks/taintSources usage

3.2 Data flow

In this section we will focus on examples that track taint from data flow operations, usually referred to as explicit flows[?]. Assume x is tainted, an operation such as $y = x * 2$ taints y.

3.2.1 Integer taint tags

Phosphor allows users to track taint tags represented as integers (allowing up to 32 different taint tags), or objects, which allows up to 2^{32} tags. In this section we focus on integer taint tags. Consequently, we also focus our API attention to `edu.columbia.cs.psl.phosphor.runtime.Tainter`. In it, we will primarily use 2 methods, as noted in the original documentantion: `taintedX`, which allows us to mark a primitive source as tainted, and `getTaint` which we can use to check for taint.

We show a series of simple examples using this across different datatypes. Note that the types can be easily replaced. Note that for each example we only show part of the relevant code, the interested reader can check the source code associated with this guide in `java.com.josecambronero.IntegerTagExamples`.

Simple binary operation

In this simple example, we create a source for the taint (variable x), and want to check if there is taint flow from our source to a sink (variable y). In this case, y is the result of an assignment operation involving x, and thus should be tainted.

We can use `Tainter.taintedInt` to taint x, and `Tainter.getTaint` to check the taint tag for y.

```
// source
int x = 0;
// sink
int y = 0;
// primitive and taint tag
x = Tainter.taintedInt(x, 1);
y = x * 3;
// check sink for taint
assert (Tainter.getTaint(y) != 0);
```

Nullifying taint

Note that the taint for a sink can change throughout the program. For example, if we define y in terms of a tainted x and then redefine y as a constant, the final y is not tainted. The excerpt from `IntegerTagExamples.testExample2` below shows this behavior.

```
// source
int x = 0;
// sink
int y = 0;
x = Tainter.taintedInt(x, 1);
// tainted
y = x * 3;
assert (Tainter.getTaint(y) != 0);
// not tainted, as zero returns constant (not function of x)
y = zero(x);
assert (Tainter.getTaint(y) == 0);
```

Arithmetic identity

Note that some arithmetic identities, which would allow a human to easily determine that no taint exists, as the result is always constant, will yield a tainted tag with taint analysis. So for example, in the excerpt below y is tainted despite the fact that it assigned 0, regardles of the tainted value of x.

```
// source
int x = 1;
x = Tainter.taintedInt(x, 1);
// sink
int y = 0;
y = 2 * x - (x + x);
assert (Tainter.getTaint(y) != 0);
```

Tracking Source of Taint

Phosphor allows the user to track the source of the taint in a sink by exploring the values of the taint tag. A similar functionality is available for the Multi-Tainter by exploring the dependencies of the tag.

In the example below, we can see that y has taint stemming solely from x, while z has taint stemming from both x and w, and nothing stemmed from r. Note that `getTaintSource` is a simple helper that checks the bits in an integer tag to find the sources, it is not part of Phosphor.

```
// sources
int x = 0, w = 0, r = 0;
x = Tainter.taintedInt(x, 2);
w = Tainter.taintedInt(x, 16);
r = Tainter.taintedInt(x, 4);
// sinks
int y = 0, z = 0;
y = x * 3;
z = x + w;

assert (Tainter.getTaint(y) != 0);
// get list of sources (recall taint tags are OR'ed)
List<Integer> zSources = getTaintSources(Tainter.getTaint(z));
assert (zSources.contains(Tainter.getTaint(x)));
assert (zSources.contains(Tainter.getTaint(w)));
assert (!zSources.contains(Tainter.getTaint(r)));
```

Note that we picked tag numbers that allow us to uniquely identify the sources (i.e. powers of 2), given that the tags are combined with logical OR. If you need more than 32 tags, you should look into using the MultiTaint functionality (see Section 3.2.2)

Element-level taint tags in arrays

Phosphor maintains tags for each element in an array, rather than a tag for the whole array[**?**]. This behavior can be seen in the excerpt below

```
// source
int x = 0;
x = Tainter.taintedInt(x, 1);
// (potential) sinks
int[] y = new int[4];
y[0] = x * 2;
assert (Tainter.getTaint(y[0]) != 0);
assert (Tainter.getTaint(y[1]) == 0);
```

3.2.2 Object taint tags

We will now experiment with object taint tags, which allow us to create arbitrary numbers of distinct tags (2^{32}, which is arbitrarily large for our purposes). In order to creat object taint tags we turn to `edu.columbia.cs.psl.phosphor.runtime.MultiTainter`, which similarly to the integer Tainter, has tainted$X$, where X is one of the various possible datatypes. As per documentation, we ignore methods with `$$PHOSPHORTAGGED` suffixes as these are for internal use.

Creating a simple object taint tag

This example creates 2 tainted integers (x and y), with two different ways of creating taint labels, and performs an addition to create a third tainted integer z (our sink). Note that we can then check the taint tag for z and also print out a string representation.

```
// Source
int x;
x = MultiTainter.taintedInt(100, "source_1");
// Source
int y;
y = MultiTainter.taintedInt(100, new Taint("source_2"));
// Sink
int z = x + y;
Taint tz = MultiTainter.getTaint(z);
assert (tz != null);
System.out.println(tz.toString());
```

Creating simple object taint tags: Object variable

Note that if we create a tainted object variable, using `MultiTainter.taintedObject`, the label provided is shown as that taint tag's label, rather than simply as a dependency. In primitives, the label is null and instead appears in the dependency graph. Note that in contrast to the other `taintedX` methods, `taintedObject` has void return. Also note that the taint tag here must be an instance of `Taint`, not any object as was possible for primitives. TODO: Verify if this is always the case...if anyone knows better, please correct

```
Object s = "My_string";
MultiTainter.taintedObject(s, new Taint("source_x"));
Object s2 = identity(s);
Taint tag = MultiTainter.getTaint(s2);
assert (tag != null);
System.out.println((String) tag.getLabel());
System.out.println(tag.toString());
```

More than 32 distinct tags

We start with a simple example that shows how we can assign more than 32 distinct tags (in this case 33).

```
// Creating more than 32 distinct tags (the advantage of object
    tags)
// Array of 33 different sources
double[] mydata = new double[33];
for(int i = 0; i < mydata.length; i++) {
    mydata[i] = MultiTainter.taintedDouble((float) i, "Source_" + i
    );
    }

// Sink
double result = 0.0;
for (double elem : mydata) {
    result += elem;
    }
```

```
Taint tag = MultiTainter.getTaint(result);
assert (tag != null);
System.out.println(tag.toString());
```

The print out of the tag in Figure 2 shows that we have 33 different taint tags as dependencies for this sink.

Tracing dependencies

So far we have been simply printing out the taint tag, in which you can see the dependencies. However, we might find it useful to actually iterate over these dependencies. In order to do this (and to more broadly see what we can do with a Taint object), we explore `edu.columbia.cs.psl.phosphor.runtime.Taint`. In this example, we'll use `getDependencies`, which returns a `edu.columbia.cs.psl.phosphor.struct.LinkedList` of Taint objects. In order to access the nodes in the linked list, you need to make sure you also import the class `edu.columbia.cs.psl.phosphor.struct.LinkedList.Node`.

```
// Iterating over dependencies in an object taint tag
int x = MultiTainter.taintedInt(100, "x");
int y = MultiTainter.taintedInt(100, "y");
int z = x + y;
double w = MultiTainter.taintedDouble(100.0, "w");
int n = 0;

int result = z + (int) w + n;
Taint tag = MultiTainter.getTaint(result);
assert (tag != null);

LinkedList<Object> dependencies = tag.getDependencies();
System.out.println(dependencies.toString());

// iterate over the linked list
Node currNode = (Node) dependencies.getFirst();
while (currNode != null) {
    String currDepName = (String) currNode.entry;
    System.out.println(currDepName);
    currNode = currNode.next;
    }
```

3.3   Implicit flows (control flow taints)

In this section we'll go over a series of examples showing how to use the control flow tracking functionality of Phosphor. Tracking tags through control flow allows us to keep track of implicit flows, meaning, information flows from one variable to another that do not take place directly through assignments (explicit flows) but rather as a consequence of control flow conditions/tests. For purposes of this section, we use a JRE that has been instrumented to use object taint tags and control flow tracking.

Boolean negation

We initially had some issues tracking tags when we performed boolean negation. However, after speaking with Jon, it was made clear to us that boolean negation is translated into a jump instruction, so tracking tags under such an operation

requires that we enable control flow tracking. This first example shows how the behavior is as expected once this has been enabled.

```
// Source
boolean x;
x = MultiTainter.taintedBoolean(false, "source1");
boolean y;
y = !x;
Taint t = MultiTainter.getTaint(y);
// we can now track unary negation, by using control flow tracking
assert(t != null);
```

If-Else statement

Our following example shows a classic case of implicit flow, where the definition of a variable, y, is not a function of a tainted variable, x, directly, but rather the tainted variable affects the branch taken and thus the value assigned to y.

```
// Source
double x = MultiTainter.taintedDouble(100.0, "x");
double y = 0.0;

// information flows from x to y through control flow
if (x >= 90.0) {
    y = 20.0;
} else {
    y = 30.0;
}

Taint t = MultiTainter.getTaint(y);
assert(t != null);
```

Ternary operator

This example is exactly the same idea as the prior example, but we now show that tracking is succesfully performed with the ternary operator as well.

```
// source
boolean x = MultiTainter.taintedBoolean(true, "x");
// sink
double y = (x) ? 1.0 : 2.0;
Taint t  = MultiTainter.getTaint(y);
assert (t != null);
```

For-loop

At risk of beating a dead horse, we introduce a simple example showing the succesful tracking of taint in a for-loop where the tainted variable is involved in the termination condition.

```
// source
int x = MultiTainter.taintedInt(100, "x");
int y = 0;

// loop termination condition allows information to flow
for(int i = 0; i < x; i++) {
    y += i;
```

```
    }
Taint  t  =  MultiTainter . getTaint ( y ) ;
assert  ( t  !=  null ) ;
```