Notes on Phosphor

José Cambronero

December 28, 2015

Abstract

This document contains my personal set of notes for building and running examples with Jonathan Bell's Phosphor[1]. 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.

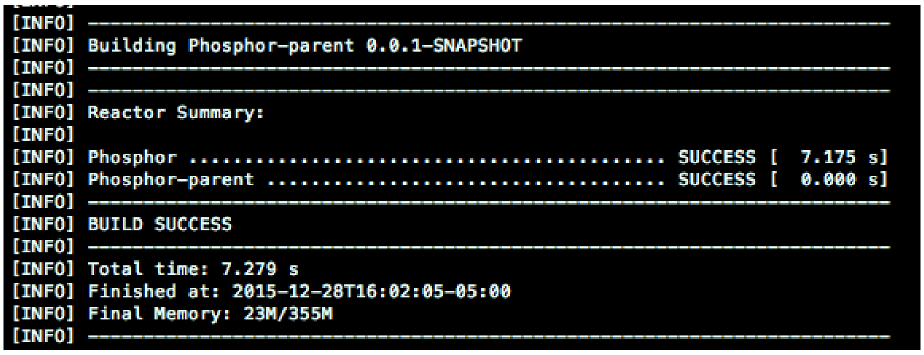
```
$ git clone git@github.com:Programming-Systems-Lab/phosphor.git
```

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



```
[INFO] ------------------------------------------------------------------------
[INFO] Building Phosphor-parent 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Phosphor ........................................... SUCCESS [  7.175 s]
[INFO] Phosphor-parent .................................... SUCCESS [  0.000 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.279 s
[INFO] Finished at: 2015-12-28T16:02:05-05:00
[INFO] Final Memory: 23M/355M
[INFO] ------------------------------------------------------------------------
```

Figure 1: Succesful build message

```
$ 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.

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[2]. 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 `com.josecambronero.IntegerTagExamples`.

3.2.1.1 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);
```

3.2.1.2 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);
```

3.2.1.3 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);
```

3.2.1.4 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.

```
// 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 **??**)

3.2.1.5 Element-level taint tags in arrays

Phosphor maintains tags for each element in an array, rather than a tag for the whole array[1]. 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);
```

3.3 Object taint tags

TODO: experiment with MultiTainter.java

3.4 Implicit flow (control flow taints)

TODO: experiment with implicit flows

References

[1] Jonathan Schaffer Bell and Gail E Kaiser. Phosphor: Illuminating dynamic data flow in the jvm. 2014.

[2] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[3] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic

execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.