Dynamic Taint Tracking for Java with Phosphor (Demo)

Jonathan Bell
Columbia University
500 West 120th St
New York, NY USA
jbell@cs.columbia.edu

Gail Kaiser
Columbia University
500 West 120th St
New York, NY USA
kaiser@cs.columbia.edu

ABSTRACT

Dynamic taint tracking is an information flow analysis that can be applied to many areas of testing. *Phosphor* is the first portable, accurate and performant dynamic taint tracking system for Java. While previous systems for performing general-purpose taint tracking in the JVM required specialized research JVMs, *Phosphor* works with standard off-the-shelf JVMs (such as Oracle's HotSpot and OpenJDK's IcedTea). *Phosphor* also differs from previous portable JVM taint tracking systems that were not general purpose (e.g. tracked only tags on Strings and no other type), in that it tracks tags on all variables. We have also made several enhancements to *Phosphor*, to track taint tags through control flow (in addition to data flow), as well as to track an arbitrary number of relationships between taint tags (rather than be limited to only 32 tags). In this demonstration, we show how developers writing testing tools can benefit from *Phosphor*, and explain briefly how to interact with it.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Taint Tracking, Dataflow Analysis

1. INTRODUCTION

Dynamic taint tracking is a form of information flow analysis that identifies relationships between data during program execution. Inputs to the application being studied are labeled with a marker (are "tainted"), and these markers propagated through data flow. Dynamic taint tracking can be used for detecting brittle tests [11], end user privacy testing [7,14] and debugging [8,12].

While the exact semantics for how labels are propagated may vary with the problem being solved, many parts of the analysis can be reused. Dytan [5] provides a generalized framework for implementing taint tracking analyses for x86
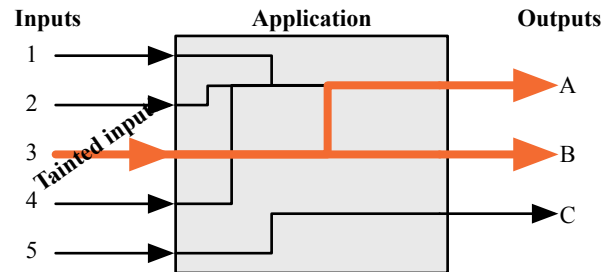
Figure 1: Example of taint tracking. Inputs enter an application, which processes them and provides outputs. Input #3 is marked as tainted, and as outputs A and B are constructed from input 3, they are marked as tainted too.

binaries, but can't be easily leveraged in higher level languages, like those that run within the JVM. By operating within the JVM, taint tracking systems can leverage language semantics that greatly simplify memory organization (such as variables). However, in Java, associating metadata (such as tags) with arbitrary variables is very difficult: previous techniques have relied on customized JVMs or symbolic execution environments to maintain this mapping [3,11,13], limiting their portability and restricting their application to large and complex real-world software.

Without a performant, portable, and accurate tool for performing dynamic taint tracking in Java, testing research can be restricted. For instance, Huo and Clause's *OraclePolish* tool uses the Java PathFinder (JPF) symbolic evaluation runtime to implement taint tracking to detect overly brittle test cases, and due to limitations in JPF, could only be used on 35% of the test cases studied. Other previous general purpose taint tracking systems for the JVM [3, 13] were implemented as modifications to research-oriented JVMs that do not support the full Java specification and are not practical for executing production code. While some portable taint tracking systems exist for the JVM, they support tracking tags through Strings only [4, 9, 10], and can not be used to implement general taint tracking analyses, as they are unable to track data in any other form.

Our dynamic taint tracking system for Java, *Phosphor*, efficiently tracks and propagates taint tags between all types of variables in off-the-shelf production JVMs such as Oracle's HotSpot and OpenJDK's IcedTea [1]. *Phosphor* works entirely through bytecode manipulation, requiring no access to source code or specialized/modified JVMs, and we have shown it to be functional and performant on real-world workloads with software like Apache Tomcat and Lucene [1].

We have extended and generalized *Phosphor* to make it

more easily adoptable by the testing community, and this tool demonstration will show how *Phosphor* can be used to implement analyses for testing. *Phosphor* can now be easily configured to propagate taint tags through data flow only (previously the only option), or through data flow and control flow. *Phosphor* can also be configured to combine tags through bitwise OR'ing (previously the only option), or through arbitrary dynamic means. Finally, *Phosphor* can be used exclusively for applying labels to variables, enabling analyses like dynamic def-use pair detection [6].

Phosphor is implemented entirely atop the ASM byte-code manipulation toolkit [2], and is released under an MIT license on GitHub[1]. *Phosphor* is logically split into two components: the tag storage engine (which associates labels with variables) and the tag propagation engine (which combines tags during program execution). We provide several examples of how *Phosphor* can be used to implement testing analyses, then briefly describe its implementation.

2. USING PHOSPHOR

Phosphor is easy to customize, and exposes several core options that change its behavior. *Phosphor* exposes a simple run-time API that developers can interact with to set or retrieve taint tag markings along with several simple configuration options for determining its taint propagation rules, following the example of past work by Clause et al [5].

Developers using *Phosphor* first have a choice of four taint tag propagation modes:
- Propagate tags through data flow only (The only option in previous versions of *Phosphor*)
- Propagate tags through control flow only
- Propagate tags through both data and control flow
- Do not propagate tags automatically at all

Developers using *Phosphor* also have a choice of using simple integer-based tags (allowing for a maximum of 32 distinct tags, combined through simple bit-wise OR'ing), or of 'multi-tainting,' using arbitrary objects as tags, and allowing *Phosphor* to track all of the distinct relationships between each tag.

Developers can choose to use the simple *Phosphor* API to set and retrieve taint tags directly, or to use the auto-tainting feature of *Phosphor*. In its auto-tainting mode, developers specify a list of "source" methods, and "sink" methods. When a source method is called, any data returned from it is tainted; when a sink method is called, all arguments are checked for taint marks.

These options support a wide diversity of analyses, two of which we describe here.

2.1 Auto-tainting for Security

Alice is testing her application to ensure that it adheres to her company's privacy policies. Her company handles a large amount of sensitive client data, and a key restriction is that none of the data leave the premises. While her team has great confidence that the product that they have developed in house doesn't leak sensitive data to outside servers, they used a number of third party libraries to simplify development, and are unsure if any of these libraries accidentally send their private data to external servers. Simply monitoring for connections to external servers is insufficient, as Alice is aware that various parts of the application intentionally communicate with outside servers.

To help her evaluate whether any of the data flowing out of her application includes sensitive client data, Alice uses *Phosphor*. She configures *Phosphor* to use its auto-tainting feature, specifying as source methods all methods that accept client data, and as sink methods all methods that are used to send data out of the application. Alice uses the default integer taint tags in *Phosphor*, as she finds that she only has a handful of distinct taint sources, and is pleased *Phosphor*'s overhead in this configuration (on average, 52% relative overhead on Dacapo 9.12-bach [1]). Alice realizes that if she needs to track more distinct taint sources than 32, she can simply enable multi-tainting in *Phosphor*. Depending on her needs, she may propagate taint tags through only data flow, or through both data and control flow.

2.2 Control Flow Tainting

Bob is tasked with maintaining the ever-growing regression test suite for his product team. Bob suspects that many of these tests contain "brittle" assertions, as he often finds himself fixing tests that unexpectedly break between versions. Following recent work by Huo and Clause, Bob wants to detect these brittle assertions up front, so that they can be identified and repaired at once [11]. While Huo and Clause have made their tool publicly available, due to limitations in its implementation, it doesn't work on most tests - 35% as evaluated by the tool authors [11] .

Bob uses *Phosphor* (which runs in his regular, Oracle JVM and supports all sorts of code) to mark each input to each test with a distinct tag (using the multi-tainting option), and the combined data-control flow tag propagation policy. This way, at any point in a test's execution, Bob can see which inputs are relevant to the current test state. Bob instruments his tests to maintain a map between input and assertion: if a test assertion depends (through data or control flow) on a test input, then he knows that that input is relevant to the test. However, if there are inputs that were not relevant to any test assertion, then he can classify these as over constrained.

3. TAG STORAGE IN JAVA

The greatest implementation challenge that *Phosphor* must overcome is how to associate arbitrary metadata (i.e. taint tags) with variables, without requiring modifications to the JVM. Before describing how taint tags are stored and retrieved in *Phosphor*, we quickly review JVM memory organization. The JVM is a stack machine with a managed memory environment, where variables are either pointers to an object, pointers to an array, or a primitive value (which include the basic types boolean, byte, char, double, float, int, long, and short). Variables can be stored directly on the operand stack, as local variables within the stack, or on the heap as static fields of classes or instance fields of objects. Method arguments are passed from the operand stack of the call site to the local variable area of the callee.

Previous systems that performed taint tracking in unmodified JVMs tended to do so by making a simple observation: much data (notably, Strings) in the JVM are represented as objects (instances of classes) [4, 9, 10]. For objects, storing a taint tag is easy: we can simply add an additional field to the definition of every class to store the tag. This approach easily addresses arrays of objects, as each object pointed to by the array still has its own taint tag. However, it does not address primitive values or arrays of primitive values.

```
1 //With Int Tag Tainting
2 TaintedIntWithIntTag doMath$$PHOSPHOR(
       int in_tag , int in ){
3  int ret = in + val ;
4  int ret_tag = in_tag | val_tag ;
5  return TaintedIntWithIntTag .
6    valueOf ( ret_tag , ret ) ;
7 }
```

(a) Modified class, using integer tags for tainting

```
1 //With Object Tag Tainting
2 TaintedIntWithObjTag doMath$$PHOSPHOR(
       Taint in_tag , int in ){
3  int ret = in + val ;
4  Taint ret_tag = in_tag . combine ( in ) ;
5  return TaintedIntWithObjTag .
6    valueOf ( ret_tag , ret ) ;
7 }
```

(b) Modified class, using object tags for multi-tainting

Figure 2: The code shown in Figure 3, as would be modified by *Phosphor* for taint tracking, with changed sections underlined. Example shown at the source level, for easier reading.

Phosphor tracks taint tags for primitive values by adding an additional variable for each primitive variable (and similarly, an additional array for each primitive array) to store the tags. The tag is stored in a location adjacent to the original primitive variable: as another field of the same class, as another local variable within the stack, an adjacent method argument, or directly adjacent on the operand stack. If a method returns a primitive value, *Phosphor* changes its return type to return instead a pre-allocated structure containing the original return and its taint tag; just after invocation at the call site, *Phosphor* inserts instructions to extract both the value and the tag to the stack.

Taint tags for primitive method arguments are always passed just before the argument that is tagged, which simplifies stack shuffling prior to method invocation. *Phosphor* modifies almost all bytecode operations to be aware of these additional variables. For example, instructions that load primitive values to the operand stack are modified to also load the taint tag to the stack; a complete listing of all of the JVM bytecode operators and the modifications made by *Phosphor* appears in the Appendix of [1]. *Phosphor* also wraps all reflection operations to propagate tags through these same semantics as well.

Figures 2 and 3 show at a basic level the sort of changes that *Phosphor* makes to store and propagate taint tags in Java code. While the examples are shown at the level of source code, note that *Phosphor* functions entirely through bytecode instrumentation, requiring no access to source. Although *Phosphor* modifies every call-site of every method to be aware of the potential extra parameters and changed return type (including those called through reflection), it is unable to modify call sites of Java methods made in native code. *Phosphor* uses stubs to wrap calls to methods from native code, allowing it to track tags heuristically, a limitation described in detail in our previous work [1].

While most traditional taint tracking systems use integers as tags, *Phosphor* can allow arbitrary objects to be used as tags, simplifying development of more complicated analyses. Integer tags allow for very low overhead in taint tag propagation, as combining them can be as simple a single instruction (to bitwise OR them), but such a technique limits the total number of taint marks to only 32. On the other hand, by using Objects as tags, there can be an arbitrary number of tags (2^32), and *Phosphor* maintains the multiple relationships between each tag ("multi-taint tagging"), but tracking these relationships adds an additional runtime overhead.

4. TAINT TAG PROPAGATION

Phosphor can combine taint tags in one of two different

```
1 // Original Code
2 int foo ( int in ){
3  int ret = in + val ;
4
5  return ret ;
6 }
```

Figure 3: Source code for a very simple method. Figure 2 shows the sorts of modifications that *Phosphor* makes to this codes to store taint tags.

ways. In traditional tainting mode, taint tags are 32-bit integers which are combined through bit-wise OR'ing, allowing for a maximum of 32 distinct tags, with fast propagation. In multi-taint mode, taint tags are objects which contain a list of all other tags from which that tag was derived, allowing for an arbitrary number of objects and relationships.

Like most taint tracking systems, *Phosphor* propagates taint tags through data flow operations (e.g. assignment, arithmetic operators, etc.). However, depending on the goals of the analysis, data flow tracking may be insufficient to capture all relationships between variables. Figure 4 shows an example of a short code snippet will return a string identical to the input, but without a taint tag (if tags are tracked only through data flow operations), since there is no data flow relationship between the input and output.

Phosphor now optionally propagates taint tags through control flow dependencies as well ("implicit flow"), which would be necessary in the case of the code in Figure 4 to propagate tags through the method. Our implementation of control flow dependency tracking mirrors that of prior work from Clause et al [5], and leverages a static post-dominator analysis (performed as part of *Phosphor*'s instrumentation) to identify which regions of each method are effected by each branch. Each method is modified to pass and accept an additional parameter that represents the control flow dependencies of the program to the point of that method. Within the method execution, *Phosphor* tracks a stack of dependencies, with one entry for each branch condition that is currently influencing execution. When a given branch no longer controls execution (e.g. at the point where both sides of the branch merge), that taint tag is popped from the control flow stack. Before any assignment, *Phosphor* inserts code to generate a new tag for that variable by merging the current control flow tags with any existing tags on the variable.

5. CONFIGURATION AND USE

Using *Phosphor* is an easy process. First, all bytecode that needs to run (including the JRE's internal classes, the application, and any of its libraries) are instrumented by invoking the *Phosphor* instrumenter, `java -jar phosphor.jar`

```
1 public String leakString(String in){
2    String r = "";
3    for(int i = 0; i < in.length; i++)
4    {
5      switch(in.charAt(i)){
6        case 'a':
7            r+="a'";
8            break;
9        ...
10       case 'z':
11           r+="z";
12           break;
13     }
14   }
15   return r;
16 }
```

Figure 4: Simple code showing the inadequacy of data flow tag propagation: the output will have no taint tag, even if the input did. Control flow propagation, however, will propagate these tags.

`[folder to instrument] [destination]`. Then we run the instrumented application.

When instrumenting code to add taint tracking instructions, *Phosphor* accepts several arguments. By default, *Phosphor* will instrument applications to use integers as taint tags; to enable multi-tainting, users specify the `-multiTaint` flag. *Phosphor* propagates taint tags through data flow only by default: control flow propagation can be enabled with the `-controlTrack` option; data flow propagation can be disabled with the `-withoutDataTrack` option.

Phosphor exposes a simple interface for setting taint marks on variables. Developers can make calls to the methods `Tainter.taintedXXX(XXX input,int tag)` or `MultiTainter.taintedXXX(XXX input,Object tag)`, replacing `XXX` with appropriate type (e.g. `int`, `long`, `Object` etc.). These methods return a copy of the input with the `tag` transparently applied (an integer tag in the case of `Tainter` or an arbitrary object in the case of `MultiTainter`). Developers can retrieve the tags on variables through the `Tainter.getTaint(...)` and the `MultiTainter.getTaint(...)` functions.

Object tags are wrapped in instances of *Phosphor*'s `Taint` class, which contains the tag specified when creating the taint marking, along with a list of dependencies on all other `Taint` types with which the tag has been combined.

Developers insert calls to these functions in their programs as they write them, compile them, use *Phosphor* to instrument them, then run them. As *Phosphor* is instrumenting applications it recognizes calls to these internal functions and inserts implementations.

Alternatively, users may use the auto-tainting features of *Phosphor*, enabled by specifying the `-taintSources` and `-taintSinks` flags when running the *Phosphor* instrumenter, pointing them to files with a list of methods to automatically mark the return values of as tainted, or to automatically check the arguments for taint tags. In the case of integer taint tags, *Phosphor* assigns each taint source a tag by the order of the taint source as appeared in the taint source file (up to 31 unique sources). In the case of object taint tags, *Phosphor* assigns the taint tags from taint sources to be the name of the method returning the tainted data. By default, *Phosphor* combines taint tags through control flow

6. CONCLUSION

Phosphor is a dynamic taint tracking system for the JVM

that can be easily applied to different program analysis problems. *Phosphor* does not require a specialized JVM, specialized operating system, or access to source code, and our previous studies have shown it to be sufficiently performant to use in testing environments (showing on average a 52% overhead on the DaCapo benchmark suite when using integer tags and data flow tag propagation). In this demonstration, we have shown how developers can interact with *Phosphor* to build their own analyses. *Phosphor* is released under an MIT license and is publicly available on github, at: `https://github.com/Programming-Systems-Lab/phosphor`.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *OOPSLA*, 2014.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[3] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC*, 2007.

[4] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *ACM Workshop on Secure Web Services*, 2009.

[5] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, 2007.

[6] G. Denaro, A. Margara, M. Pezze, and M. Vivanti. Dynamic data flow testing of object oriented systems. In *ICSE*, 2015.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[8] M. Ganai, D. Lee, and A. Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *FSE*, 2012.

[9] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *ACSAC*, 2005.

[10] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *FSE*, 2006.

[11] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, 2014.

[12] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. P. Lippmann. Coverage maximization using dynamic taint tracing. Technical Report TR-1112, MIT Lincoln Lab, 2007.

[13] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, Feb. 2008.

[14] R. Spahn, J. Bell, M. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *OSDI*, 2014.

A.   USER GUIDE

Phosphor is available for download (both source code and pre-compiled binaries) on GitHub, at `https://github.com/Programming-Systems-Lab/phosphor`. The GitHub page contains further links to the artifact that passed the OOPSLA 2014 artifact evaluation process, which consists of a VirtualBox VM image that contains all of the java experiments performed in the original OOPSLA 2014 paper on *Phosphor*.

We describe here a brief getting started guide (which is also available on the *Phosphor* website), as well as a listing of the options available when using *Phosphor* and the key API methods exposed by *Phosphor*.

A.1   Getting Started

Phosphor works by modifying your application's bytecode to perform data and control flow tracking. To be complete, *Phosphor* also modifies the bytecode of JRE-provided classes, too. The first step to using *Phosphor* is generating an instrumented version of your runtime environment. We have tested *Phosphor* with versions 7 and 8 of both Oracle's HotSpot JVM and OpenJDK's IcedTea JVM.

We'll assume that in all of the code examples below, we're in the same directory (which has a copy of phosphor.jar), and that the JRE is located here: `UNINST_JAVA` (modify this path in the commands below to match your environment).

Then, to instrument the JRE we'll run: `java -jar phosphor.jar UNINST_JAVA jre-inst`.

The instrumenter takes two primary arguments: first a path containing the classes to instrument, and then a destination for the instrumented classes. Full use including all options is detailed in Figure 5.

After instrumenting the JRE, make sure to chmod +x the binaries in the new folder, e.g. `chmod +x jre-inst/bin/*`.

The next step is to instrument the code which you would like to track. We'll start off by instrumenting the demo suite provided under the PhosphorTests project. This suite includes a slightly modified version of DroidBench, a test suite that simulates application data leaks (modified to remove Android-specific tests that are not applicable to a desktop JVM). We'll instrument the phosphortests.jar file: `java -jar phosphor.jar phosphortests.jar inst`.

This will create the folder inst, and place in it the instrumented version of the demo suite jar.

We can now run the instrumented demo suite using our instrumented JRE using the command:
`jre-inst/bin/java -Xbootclasspath/a:phosphor.jar -cp inst/phosphortests.jar -ea phosphor.test.DroidBenchTest`. The result should be a list of test cases, with assertion errors for each "testImplicitFlow" test case (assuming you did not enable control flow tracking).

A.2   Interacting with Phosphor

Phosphor exposes a simple API to allow marking data with tags, and to retrieve those tags, shown in Figure 6. Key functionality is implemented in two different classes, one for interacting with integer taint tags, and one for interacting with object tags (used for the multi-taint mode). To get or set the taint tag of a primitive type, developers call the `taintedX` or getTaint(X) method (replacing `X` with each of the primitive types). To get or set the taint tag of an object, developers first cast that object to the inter-

```
Usage: java -jar phosphor.jar [OPTIONS] [in] [out]
 -controlTrack              Enable taint tracking
                            through control flow
 -help                      print this message
 -multiTaint                Support 2^32 tags
                            instead of just 32
 -taintSinks <taintsinks>   File with listing of
                            taint sinks to use to
                            check for auto-taints
 -taintSources <taintSources>File with listing of
                            sources to auto-taint
 -withoutDataTrack          Disable taint
                            tracking through data
                            flow (on by default)
```

Figure 5: Arguments accepted by *Phosphor*

```
//Integer−taint related API
//Class: edu.columbia.cs.psl.phosphor.
    runtime.Tainter
int getTaint(<primitive type>);
<primitive type> taintedPrimitiveType(<
    primitive type> val , int tag);
//Interface: edu.columbia.cs.psl.phosphor.
    struct.TaintedWithIntTag
int getPHOSPHOR_TAG();
void setPHOSPHOR_TAG(int tag);

//Multi−taint related API
//Class: edu.columbia.cs.psl.phosphor.
    runtime.MultiTainter
Taint getTaint(<primitive type>);
<primitive type> taintedPrimitiveType(<
    primitive type> val , Object tag);
//Interface: edu.columbia.cs.psl.phosphor.
    struct.TaintedWithObjTag
Taint getPHOSPHOR_TAG();
void setPHOSPHOR_TAG(Taint tag);
//Class: edu.columbia.cs.psl.phosphor.
    runtime.Taint
Taint(Object label);
LinkedList<Taint> getDependencies();
Object getLabel();
```

Figure 6: Key API methods, classes and interfaces exposed by *Phosphor*

face `TaintedWithIntTag` or `TaintedWithObjTag` (*Phosphor* changes all classes to implement this interface), and use the get and set methods.

In the case of integer tags, developers can determine if a variable is derived from a particular tainted source by checking the bit mask of that variable's tag (since tags are combined by bitwise OR'ing them). In the case of multi-tainting, developers can determine if a variable is derived from a particular tainted source by examining the dependencies of that variable's tag.

A.3   Extending Phosphor

We have released *Phosphor* under an MIT license, and encourage its use and extensions of it. We would very much welcome any feedback regarding *Phosphor*.