Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs

Jonathan Bell and Gail Kaiser
Columbia University, New York, NY USA
{jbell,kaiser}@cs.columbia.edu

Abstract

Dynamic taint analysis is a well-known information flow analysis problem with many possible applications. Taint tracking allows for analysis of application data flow by assigning labels to data, and then propagating those labels through data flow. Taint tracking systems traditionally compromise among performance, precision, soundness, and portability. Performance can be critical, as these systems are often intended to be deployed to production environments, and hence must have low overhead. To be deployed in security-conscious settings, taint tracking must also be sound and precise. Dynamic taint tracking must be portable in order to be easily deployed and adopted for real world purposes, without requiring recompilation of the operating system or language interpreter, and without requiring access to application source code.

We present PHOSPHOR, a dynamic taint tracking system for the Java Virtual Machine (JVM) that simultaneously achieves our goals of performance, soundness, precision, and portability. Moreover, to our knowledge, it is the first portable general purpose taint tracking system for the JVM. We evaluated PHOSPHOR's performance on two commonly used JVM languages (Java and Scala), on two successive revisions of two commonly used JVMs (Oracle's HotSpot and OpenJDK's IcedTea) and on Android's Dalvik Virtual Machine, finding its performance to be impressive: as low as 3% (53% on average; 220% at worst) using the DaCapo macro benchmark suite. This paper describes our approach toward achieving portable taint tracking in the JVM.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

Keywords Taint Tracking, Dataflow Analysis

To appear at 29th Annual ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '14)

1. Introduction

Dynamic taint analysis (also referred to as dynamic information flow tracking) is a powerful form of information flow analysis useful for identifying the origin of data during execution. Inputs to an application are "tainted," or labeled with a tag. As computations are performed, these labels are propagated through the system such that any new values derived from a tagged value also carry a tag derived from these source input tags. In this way, we can inspect any object and determine if it is derived from a tainted input by inspecting its label. By maintaining a precise mapping from objects to labels, we can enable a broad range of analyses, for such purposes as end-user privacy testing [16], fine-grained data security [3, 10, 27, 31], detection of code-injection attacks [22, 33, 35] and improved debugging [17, 25].

Taint tracking systems typically face challenges in both precision and soundness, in that it is generally difficult to determine the bounds of a variable in memory. Therefore, some parts of a variable may become dissociated with their intended taint tag, or multiple variables may inadvertently be tracked together as a single variable. Taint tracking systems similarly face a challenge of performance: many applications for taint tracking rely on its use in real, deployed systems, demanding an acceptably low run time overhead.

Traditional approaches to building taint tracking systems that address these challenges normally rely on modifications to the operating system [39, 44], modifications to the language interpreter [3, 9, 16, 20, 28, 29], or access to source code [24, 42]. While taint tracking at the interpreter or source code level can improve soundness and precision (by providing variable-level memory semantics), these approaches all introduce a new challenge: portability. Such approaches can have limited applicability due to the need for modifications to client systems (e.g. specialized operating systems), need for specialized interpreters, or need to access source code. For example, in the case of two taint tracking systems that target the JVM [9, 28], they are restricted to support only research-oriented JVMs which do not support the full Java specification. For example, these JVMs are unable to execute the entirety of the popular macro-benchmark suite DaCapo [5], an indicator that they may be impractical to deploy in client environments. We are not aware of any taint tracking systems that target the JVM that are sufficiently portable to function with commonly used JVMs from vendors such as Oracle and the OpenJDK project.

Our key insight is that we can leverage the same benefits (e.g., in soundness and precision) that interpreter-level and source code-level approaches gain *without* requiring any modification to the underlying interpreter by taking advantage of the strong specification for the intermediate language (i.e. byte code) that runs in that interpreter. This approach is a compromise between interpreter level approaches (which do not require access to source code but require modifications to the interpreter) and source level approaches (which require access to source code but do not change the interpreter), functioning instead at the level of byte code. PHOSPHOR provides taint tracking within the Java Virtual Machine (JVM) without requiring any modifications to the language interpreter, VM, or operating system, and without requiring any access to source code. Moreover, PHOSPHOR can be applied to any commodity JVM, and functions with code written in any language targeting the JVM, such as Java and Scala.

PHOSPHOR's approach to tracking variable level taint tags (without modifying the JVM) seems simple at first: we essentially need only instrument all code such that every variable maps to a "shadow" variable, which stores the taint tag for that variable. However, such changes are actually quite invasive, and become complicated as our modified Java code begins to interact with (non-modified) native libraries. In fact, we are unaware of any previous work that makes such invasive changes to the bytecode executed by the JVM: most previous taint tracking systems for the JVM use slower mechanisms to maintain this shadow data [40]. We present a detailed description of the challenges that we faced implementing our instrumentation, and describe how our general approach could be used for other sorts of dynamic data and control flow analyses in Java.

We evaluated PHOSPHOR on a variety of macro and micro benchmarks on several widely-used JVMs from Oracle and the OpenJDK project, finding its overhead to be impressively low: as low as 3.32%, on average 53.31% (and up to 220%) in macro benchmarks. We also compared PHOSPHOR to the popular, state of the art Android-only taint tracking system, TaintDroid [16], finding that our approach is far more portable, is more precise, and is comparable in performance.

The contributions of this paper are:

- A general purpose approach to efficiently storing metadata for variables in the JVM, without requiring any modifications to the JVM.
- A general purpose approach to propagating this shadow information in the form of taint tracking, again, without requiring any modifications to the JVM.
- A description of our open source implementation of this technique: PHOSPHOR (released on GitHub [4]).

2. Motivation

Although several existing systems target Java applications (e.g. [12, 21, 22]) by modifying application or library byte code, these are not general purpose: they can only track data flow of Java Strings (and not of any other type), and therefore are unable to continue tracking those Strings in the event that they are converted by the application to another representation (such as a character array). Moreover, these systems can not track inputs that are not Strings (e.g. integers, or a language-specific version of String in another, non-Java JVM language).

Several existing systems can perform taint tracking on all data types in Java, but are highly restricted in portability, functioning only on research JVMs. The JVMs targeted by [28] (Kaffe [37]) and [9] (Jikes RVM [36]) support only a subset of Java version 6, severely limiting applicability. We will refer to both of these incomplete JVMs as "research JVMs," as they do not implement the complete Java specification, and are principally used within the research community (rather than in production environments).

We also note that while we focus on dynamic taint tracking, static taint analysis is also a topic of interest. However, while static taint analysis for Java [2, 34, 38] can determine a priori where data might leak from a system, it may report false positives from code which can not execute in practice, and as with all static analysis tools for Java, it must model reflective calls, possibly further increasing the likelihood of false positives.

There is a need for a general purpose taint tracking system that is sufficiently decoupled from specific data types to support a wide range of precise and sound analyses (i.e. with no false positives or false negatives) for applications running on any production JVM. We briefly describe work in three broad areas that could benefit from PHOSPHOR.

2.1 Detecting injection attacks

Taint tracking has been widely studied as a mechanism for improving application security. Taint tracking can be used to ensure that untrusted inputs from external sources (such as an end-user) are not used as inputs to critical functions [22, 33, 35]. For instance, consider an application that takes an input string from the user, and then reads a file based on that input, returning the file to the user. An attacker could perhaps craft an input to coerce the application to read and return an arbitrary file, including sensitive files such as `/etc/passwd`. Similar injection attacks can occur when calling external processes, or performing SQL queries. SQL injection attacks are the fifth most prevalent type of attack reported by CVE [14].

Taint tracking has been shown to be effective in detecting these sorts of attacks: all user input is tagged with a taint, and any function that may be an injection point is instrumented to first check its arguments to ensure that there are no taint tags. Trusted input sanitizers that sit between the user's input and the injection point can be used to allow sanitized inputs to flow to possible injection points (with the assumption that they will correctly sanitize the input).

2.2 Privacy testing and fine grained-access control

Taint tracking has also been successfully applied to fine-grained information access control [3, 10, 27, 31], and to end-user privacy testing [16]. In both cases, taint tracking is used to improve the granularity of existing mechanisms for enforcing rules about information flow. For access control, taint tracking is useful as it allows developers or system administrators to specify access rules based on data. For instance, administrators may wish to restrict the operations that users may perform on certain data, without a priori knowledge of where in the application's control flow that data may appear from. As another example, an application may include untrusted libraries during run time, and want to restrict those libraries from accessing sensitive data.

For end-user privacy testing, users specify system-wide taint sources (e.g. on a mobile device, GPS location, personal contacts, etc.), and destinations, where tainted data must never flow to (e.g. system-level functions that send data over the network). In this way, users can determine if their private information is being transferred to remote servers.

Note that both of these applications of taint tracking demand a system that is both performant and portable. For example, an end-user may wish to observe the privacy violations of an application, without the prior planning of the application developers to support taint-tracking, and without requiring specialized hardware or a specialized operating system. Both systems would be challenging to implement in the JVM without a taint tracking system.

2.3 Testing and Debugging

Taint tracking has also been employed to improve the testing and debugging process. For instance, taint tracking can be used to increase test coverage when using automated input generators [25]. In this application, the taint tracking system labels each input, and at each conditional branch, records what label (or set of labels) the jump condition had. This information is then fed back to the input generator to focus input generation on those that are known to be restricting control flow. This approach can also be useful for debugging program failure by using taint tracking to identify which inputs were relevant to the crash [17].

3. Approach

In designing PHOSPHOR, our primary goal was to enable studies and analyses of dynamic data flow in languages that target the JVM, such as Java, Scala and Clojure. While some of theses analyses may be targeted towards researchers running experiments in closed environments (in which case, run time overhead and portability are unlikely to be significant concerns), others may target actual use by end-users (e.g. the privacy study performed in [16]). Hence, a key goal for PHOSPHOR was to ensure that it has both relatively low run time overhead and was portable (i.e. could be used on a variety of JVMs and platforms).

In general, common challenges to building taint tracking systems in support of such analyses include:

1. **Soundness:** When working with native binaries, it can be difficult or impossible to determine the correct level of granularity to assign distinct taint tags. Should each byte be distinctly tagged? Each word? These questions are difficult if not impossible to answer in the general case, and can directly impact the soundness of the tool. If a tool is not sound, then it may incorrectly drop taint information from variables.
2. **Precision:** In the process of improving soundness of a taint tracking system, systems often trade higher accuracy for lower precision, leading to over tainting, where taint tags are propagated between values even when there is no actual connection between them. In some cases, over tainting can lead to significant decreases in precision, with values marked by the wrong tag. If a tool is not precise, it may incorrectly add additional taint information to variables.
3. **Portability:** Most taint tracking systems require access to application source code [24, 42], require modified operating systems [39, 44] or modified language interpreters [3, 9, 20, 28, 29].
4. **Performance:** Taint tracking often adds a very high performance overhead (commonly showing slowdowns of 1x-30x depending on the tool and benchmark), limiting its use in deployment environments.

Our approach to taint tracking uses *variable-level* tracking, inspired by previous work that modified the interpreter to support taint-tracking in Java [9, 16, 28]. A key observation is that when operating within the JVM (e.g. in Java, Scala and others), we can bypass the common challenges related to accuracy and precision: variables are clear units of data, and because code can not access arbitrary memory addresses, we can be certain that if we associate a taint tag with a variable, any access to that variable can be mapped to the taint tag. Therefore, this design choice can eliminate some difficulties associated with maintaining precision in taint tracking that typically affect systems operating at a lower level (e.g. at the OS level [39, 44], or via binary instrumentation [11, 13]).

Most taint tracking systems for other memory managed languages (e.g. targeting JavaScript [20], php [33, 43], Dalvik [16], Java [9, 28] and others), rely on modifications or extensions to the interpreter, which allows taint tracking code access to significantly lower level memory operations than taint tracking code running within a managed environment like the JVM. However, in order to ensure portability,

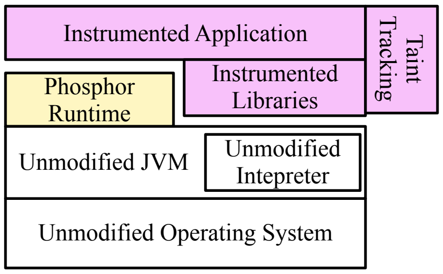

Figure 1: The high level architecture of PHOSPHOR

we designed PHOSPHOR to run entirely within the confines of an unmodified JVM.

The decision to run within the confines of code executing in the JVM (and not inside of the JVM's interpreter) raises several unique challenges because our taint tracking instrumentation is subject to the same memory management restrictions that any other code is. The prime challenge in creating PHOSPHOR (and our key contribution), therefore, is to efficiently maintain a mapping from values to taint tags within the confines of a memory-managed environment.

3.1 JVM Background

Before describing how PHOSPHOR works, we first provide a brief background on data organization within the JVM (based on the JVM specification, version 7 [26]).

There are eight "primitive" types supported by the JVM, all of which are stored and passed by value: boolean, byte, character, integer, short, long, float, and double. In addition to primitive types, the JVM supports two reference types: objects and arrays. Objects are instances of classes, which may contain fields (which are members of each instance) and static fields (which are members of each class). Arrays can be declared to store either reference types (which would include other arrays) or primitive types. Reference types can be cast to a super type, which affects what operations are available on that instance of that type, and are all sub-types of the root type, `java/lang/Object`.

The JVM is a stack machine, with stack memory split into two components: the operand stack and the local variable area. The operand stack is used for passing operands to instructions and can only be manipulated with stack operators, while the local variable area is indexed. Method arguments are passed by placing them on the operand stack, and are accessed by the receiver as local variables. The combination of the operand stack and local variable area make up a JVM frame. When a method is invoked, a new frame is created for that method, and when it returns, the frame is destroyed. It is impossible for code can to access any frame other than the current frame.

3.2 High Level Design

Figure 1 shows a high level overview of our approach to portable taint tracking with PHOSPHOR: we modify all byte code running within the JVM, and then run that code in a completely unmodified JVM, running on an unmodified operating system, with commodity hardware.

PHOSPHOR's taint tracking is based on variable-level tracking, storing a tag for every variable. When operations are performed on these variables, PHOSPHOR combines their taint tags to create the new tag for the resulting combination.

PHOSPHOR modifies byte code to include storage for taint tags and to include instructions to propagate these tags. We use the ASM [6] byte code manipulation library to insert our instrumentation and support all recent versions of the Java byte code specification (up to version 8). This instrumentation normally occurs offline (before execution) but in the event that a class is defined at run time (and hence, wasn't instrumented), PHOSPHOR intercepts all classes as they are loaded, ensuring that every single class is instrumented. The instrumentation process is performed only once per class and is relatively quick, requiring only 1.4 minutes to instrument the entire Java 7 JRE (approximately 19,000 classes). Figure 2 shows an example of the sorts of transformations that are applied to byte code. Note that our example is shown as Java source code for ease of understanding, but in reality, all transformations occur at the level of Java byte code.

At the high level, PHOSPHOR adds a field to every Class to track the tag of instances of that Class, and adds a shadow variable for every variable (be they local variables, method arguments, or fields) that is not an instance of a Class to track that variable's tag. When it's impossible to add such a shadow variable (e.g. to pass the tag of a primitive return value from a method), PHOSPHOR combines the taint tag with the value into a *container class*, which encapsulates both the tag and the value into one reference (which is then the return value). Formally, PHOSPHOR consults the following five properties to determine how to store or retrieve the taint tag for a variable:

Property 3.1. Let R be a reference to an instance of an Object. Then the taint tag of R is stored as a component of the object to which R points.

Property 3.2. Let A be a reference to an array of references. Then the taint tag of array element $A[i]$ is stored as a component of the object to which $A[i]$ points.

Property 3.3. Let V be a primitive value. Then the taint tag of V is stored as a shadow value next to V.

Property 3.4. Let A be a primitive array reference. Then a shadow array A_s is stored next to A, and the taint tag of primitive value $A[i]$ is $A_s[i]$.

Property 3.5. Let A be a primitive array reference and A_s be the reference to its shadow array. If A is stored as the type `Object`, then A and A_s are first boxed into a container, as $C(A, A_s)$.

Note that by these properties, every single variable has its own distinct taint tag: each element in an array is tracked distinctly (unlike in other taint tracking systems, such as

```java
public class Example{ //Original Code

 static int val;
 byte[] bArray;
 String str;


 Example(int n){
  bArray = new byte[n];

 }
 int doMath(int in){
  int ret = in + val;

  return ret;
 }
}
```

(a) The original class

```java
public class Example { //With Tainting
 int phosphorTag;
 static int val;
 static int val_tag;
 byte[] bArray;
 int[] bArray_tag;
 String str;
 Example(int n_tag, int n){
  bArray = new byte[n];
  bArray_tag = new int[n];
 }
 TaintedInt doMath(int in_tag,int in){
  int ret = in + val;
  int ret_tag = in_tag | val_tag;
  return TaintedInt.valueOf(ret_tag, ret);
 }
}
```

(b) The modified class, ready to track taint tags

Figure 2: A basic example of the sort of transformations that PHOSPHOR applies at the byte code level to support taint tracking. Underlined lines call out to changes made by PHOSPHOR. Example shown at the source level, for easier reading.

[9, 16], which sacrifice this precision for added performance by storing only a single taint tag for all of the elements in an array). The implementation and rationale behind each of these properties is described in much greater detail in §4.1.

These properties are also enforced when programs dynamically access fields and invoke methods via Java's reflection interface, which we patch to propagate taint tags.

PHOSPHOR can automatically apply a taint tag to variables that are returned from pre-defined taint "source" methods (for instance, methods that take user input). When applying taint-tracking transformations to byte code, PHOSPHOR consults a configuration file for a list of methods that should result in their return value (or arguments) being tainted. PHOSPHOR also consults the same configuration file for a list of methods that should check their arguments to determine if any of them are tainted (a "taint sink," for instance, a method that executes a SQL command), logging the occurrence or raising an exception in the case that they are tainted.

For more complicated semantics to mark variables with taint tags and respond to variables that are marked, PHOSPHOR provides a simple API, exposing the simple functions `setTaint` and `getTaint`, which respectively set the taint tag of a variable and retrieve the taint tag of a variable. These functions are useful for implementers of analyses that build upon PHOSPHOR, and are not intended to need to be inserted into any target application code directly.

PHOSPHOR represents the taint of a variable as a 32-bit long bit vector, allowing for a total of 32 distinct taints (similar to other systems, such as TaintDroid [16]. When taint tags are combined, they are bit-wise OR'ed. Alternatively, a developer could specify more complex logic for generating and combining taint tags, allowing for 2^{32} possible taint tags, although with perhaps greater overhead (an evaluation which we leave for future work and consider out of scope). We also have anecdotal evidence showing that PHOSPHOR can use any arbitrary type (e.g., objects) to represent the taint tag of a variable — not just a primitive number.

3.3 Approach Limitations

There are several limitations to our approach. First: PHOSPHOR is a system for tracking dynamic data flow through taint analysis, and does not track taint tags through control flow. That is, tags are combined through "explicit" operations (i.e. data flow), and not through "implicit" operations (i.e. control flow). However, existing approaches towards implementing control flow tracking (e.g. [9, 13, 28]) could be combined with our approach for data flow tracking, which we consider outside of the scope of this paper.

Next, as PHOSPHOR functions within the confines of the JVM, it is unable to track data flow through native code executing outside of but interacting with the JVM. We have implemented the current best-practices for handling such flows (i.e., assuming that all native code propagates taints from all inputs to all outputs), discussed further in §4.3. Finally, since our approach requires modifying the byte code of applications, this could modify the behavior of applications that somehow use that byte code as an input, since the byte code will have been modified by PHOSPHOR to include taint propagation instructions. Typically in Java, such inspection is done using the *Reflection* interface, which our implementation patches to hide all traces of PHOSPHOR. PHOSPHOR works with applications that use Java's *Reflection* to read their byte code, but does not work with applications that use other, non-standard approaches to read their code.

4. Implementation

PHOSPHOR consists of an instrumenter that modifies each Java class (either offline, or dynamically at load-time by intercepting classes as they are loaded) to add additional variables and instructions to perform taint tracking, and a small runtime library. The runtime library is very small, and consists only of several helper methods used for ensuring that taint tags are tracked through calls to Java's reflection interface. There are no central data structures that store taint tags: as shown in Figure 2(b), taint tags are stored in variables adjacent to the variables that they are tracking. This lack of centralized structure allows PHOSPHOR to be both performant and thread-safe.

4.1 Taint Tag Storage

Based on the discussion above of memory organization within the JVM, we consider shadow variable storage (for taint tags) in four different areas: as fields, as local variables, on the operand stack, and as method return values. Moreover, based on the discussion of types in the JVM, we consider five broad categories of variables for which we may need different taint tag representations: primitives, primitive arrays, multi-dimensional primitive arrays, arrays of other references, and general references. For each of these types, we will enumerate rules for their taint tag storage.

4.1.1 Reference Types

PHOSPHOR stores one taint tag per-variable, so there is no tag stored for each reference to a variable: the taint tag of a reference is simply the tag of the value that it points to. Storing the taint tag for references that point to instances of classes (i.e. objects) is straightforward: PHOSPHOR adds a new field to that type, such that each instance of the class has an extra field in which we can store the taint tag. This model extends to support arrays of reference types, since the taint tag of each reference type in the array is stored directly as part of the reference type. From these two observations, we can derive Properties 3.1 and 3.2.

However, there are reference types for which PHOSPHOR can not add an extra field to track the taint tag of that type: notably, primitive multi-dimensional arrays. Recall that primitive arrays are reference types, so a multi-dimensional primitive array must be an array of reference types. Since arrays are not objects, we can not simply add a field to that type: instead, we create a new class to *box* the primitive array and its taint tag into a single type. For example, an N-dimension array $char[][][]$, will be mapped to an $(N-1)$-dimension array of $MultiDimensionCharArray[][]$, where $MultiDimensionCharArray$ is a class that has two fields: a $char[]$ field to store the value of the final dimension of the array, and an $int[]$ field to store its taint tags. All references to multi-dimension primitive arrays are remapped to access the array through the container, ensuring that Property 3.2 continues to hold.

4.1.2 Primitives and Primitive Arrays

For variables that are primitives (or primitive arrays), we cannot simply add an extra field to the type to store the tag, since there is no structure exposed within the JVM that represents these types that we could modify. Instead, PHOSPHOR stores the taint tag (or a reference to the taint tag) in a shadow, alongside the actual value (Properties 3.3 and 3.4). This subsection will describe exactly where that shadow is stored.

For variables that are stored as fields in a class, PHOSPHOR creates a shadow field to store the taint tag for that element. For instance, if a class has a member `private int val`, then PHOSPHOR adds another field: `private int val_tag`.

To support primitive values and primitive arrays as local variables, PHOSPHOR creates an additional local variable to store the taint tag, for each local variable that represents a primitive or primitive array. Primitive and primitive array method arguments are supported similarly to local variables: we create shadow arguments to track the taint tag for each primitive and primitive array argument.

Primitive and primitive array return types are supported by boxing the value and its taint tag into a container just before return. PHOSPHOR changes the return type of all such methods to be the appropriate container, and modifies the return instruction to first construct the container, and then return it (instead of just returning the primitive value or primitive array reference). Just after the call site to a method that returns a container type, the container is unboxed, leaving the primitive return value on the stack, with the taint tag just below it. To reduce overhead, each method pre-allocates containers at its entry point for the methods that it will call, passing these containers to each method called. In this way, if a method makes several calls to another method which returns a primitive value, only one container is allocated, and is re-used for each call.

To support primitive values and primitive arrays on the operand stack, PHOSPHOR instruments every stack operator to ensure that before any primitive value or primitive array reference is pushed onto the stack its taint tag is pushed as well, and just after a primitive value or primitive array reference is popped, its taint tag is as well.

PHOSPHOR creates these extra fields and variables as necessary based on the type information for the field or variable. However, note that because primitive arrays are reference types, they are assignable to fields and variables with the generic type `Object` (for which PHOSPHOR would not have a priori created a shadow variable). PHOSPHOR accounts for this situation by automatically boxing primitive arrays with their taint tags before assigning them to the generic type

`Object`, and by automatically unboxing them when casting from the generic type `Object` back to a primitive array.

4.2 Propagating Taint Tags

The remainder of this section will describe the specific changes made to application and library byte code to propagate taint tags. A complete listing of all byte codes available and the modifications that PHOSPHOR makes is available in the appendix to this paper, in Table 4.

Method and Field Declarations: PHOSPHOR rewrites all method declarations to include taint tags for each primitive or primitive array, and to change all primitive and primitive array return types to be container types, which include the taint tag on the primitive value in addition to the actual value. All references to multi-dimension primitive arrays (in both fields and method descriptors) are replaced with container types. PHOSPHOR adds a new instance field to every class, used to track the taint tag of that instance. Finally, for every field that is a primitive or primitive array, PHOSPHOR adds an additional field that stores the taint of that primitive or primitive array.

Array Instructions: For all array load or store instructions, PHOSPHOR must remove the taint tag of the array index from the operand stack before the instruction is executed. For stores to primitive arrays, PHOSPHOR inserts instructions to also store the taint tag of the value being stored into the taint array. For loads from primitive arrays, PHOSPHOR similarly inserts instructions to load the taint tag from the taint array. For stores to reference type arrays, if the item being stored is a primitive array, PHOSPHOR inserts code to box the array and tag into a container before storing it.

PHOSPHOR instruments instructions that create new one-dimension primitive arrays with additional instructions to also create a taint tag array with the same length. For instructions that create multi-dimension primitive arrays, PHOSPHOR modifies them to instead create arrays of our containers (as discussed in §4.1.1).

The last array instruction that PHOSPHOR instruments is `ARRAYLENGTH`, which pops an array off of the operand stack and pushes onto the stack the length of that array. For this instruction, PHOSPHOR adds instructions to pop the taint array from the stack (if the array is a primitive array), and to add an empty taint (i.e. 0) to the returned value (we consider array length to be a control flow operation, and do not propagate any array taints into the taint of the length of each array).

Local Variable Instructions: PHOSPHOR adds an instruction to store a variable's taint tag immediately after each instruction that stores a primitive or primitive array variable. Similarly, for instructions that store object references to local variables, if the variable type is a primitive array, PHOSPHOR also stores the taint tag array for that variable. If the variable type is not a primitive array (i.e. *Object*), but the item being stored is a primitive array, then PHOSPHOR inserts instructions to first box the array into a container, before storing the array. For instructions that load local variables onto the operand stack, if the variable is a primitive or primitive array, then just before the variable is loaded, PHOSPHOR loads the pre-existing shadow variable (containing the taint tag) onto the stack.

Method Calls: PHOSPHOR instruments every method call, first modifying the method descriptor (i.e. the arguments and return type) to pass taint tags. Next, PHOSPHOR ensures that for every parameter of the generic type `Object`, if the parameter being passed is a primitive array, its taint array is boxed with it into a container. If the method is an instance method (i.e. has a receiver instance), PHOSPHOR ensures that if the receiver is a primitive array, its taint tag is dropped from the operand stack before the call. Immediately after the method call, if its return type had been changed to a container type, instructions are inserted to unbox the container, placing on the top of the stack the return value followed by the taint tag.

Method Returns: PHOSPHOR ensures that all return instructions that would otherwise return a primitive value or reference to a primitive array first box the primitive or primitive array with its taint tag(s) before returning.

Arithmetic Instructions: For arithmetic operators that take two operands (e.g. addition, subtraction, multiplication, etc), each operator expects that the top two values on the stack are the operands, yet with PHOSPHOR, the top value will be the first operand, while the second will be the taint tag of the first operand, and the third the second operand, with the fourth its taint tag (as shown in Figure 3). PHOSPHOR prepends each arithmetic operator with instructions to combine the two taint tags (by bitwise ORing them), placing the new taint tag under the two (intended) operands, allowing the arithmetic to complete successfully.

Type Instructions: The JVM provides the `instanceof` instruction, which pops an object reference off of the stack and returns an integer indicating if that reference is an instance of a specified type. For this instruction, PHOSPHOR inserts a null taint tag (i.e. "0") under the return value of the instruction (similar to array length, we consider this to be a control flow operation). Additionally, if the reference type on the operand stack is a primitive array, then its taint tag array is dropped from the stack. If the type argument to `instanceof` is a multidimensional primitive array, then PHOSPHOR changes the argument to instead refer to the appropriate container type (since again, we have eliminated multidimensional primitive arrays).

The other type instruction that PHOSPHOR instruments is the `checkcast` instruction, which ensures that the object reference at the top of the stack is an instance of a specified type, throwing an exception if not. PHOSPHOR rewrites this instruction to be aware of our boxed container types: if the cast is to a one-dimension primitive array type and the operand is a container, PHOSPHOR first unboxes the array and its taint tag array. If the cast is to a multi-dimension primitive array, then PHOSPHOR changes the type cast to be to the appropriate container type (since PHOSPHOR eliminates multi-dimensional primitive arrays), leaving it boxed.

Stack Manipulators: There are several instructions that directly manipulate the order of elements on the operand stack, for instance, swapping the top two values. In all cases, PHOSPHOR modifies each instruction based on the contents of the operand stack just before execution. For instance, if an instruction will swap the top two elements on the stack, and the top element is a primitive value (with a taint tag stored beneath it), but the element below that is an object reference (hence, with no taint tag beneath it on the stack), PHOSPHOR removes the swap instruction, replacing it with instructions to place the top two elements beneath the third.

Locking Instructions: There are two instructions in Java byte code related to locking, one to procure a lock on an object reference, and one to release a lock already held on an object reference. In both cases, PHOSPHOR checks the top stack value, and if it is a one dimensional primitive array (which implies that there is a taint tag array on the stack beneath it), PHOSPHOR pops the taint tag array after the lock is acquired or released.

Jump Instructions: The JVM provides several jump instructions, jumping on either one or two object references or primitive values. For those that jump based on primitive values, in all cases PHOSPHOR first removes the taint tag from the value(s) being checked before the jump. For those that jump based on object references, PHOSPHOR removes the taint array tag, if the value(s) being checked before the jump are references to one dimensional primitive arrays.

4.3 Native Code and Reflection

As PHOSPHOR is implemented within the JVM, it is restricted from propagating taint tags in code that executes outside of the JVM. The JVM allows for "native" methods, which are implemented in native machine code, and can be called by normal code running inside of the JVM. We follow the same approach used by TaintDroid [16] for patching taint flow through these methods: we surround each with a wrapper that can propagate taint tags from the arguments of the method into the return value. As with TaintDroid, our implementation currently assigns the taint tag of the return type to be the union of the taint tags of all primitive, primitive array and String parameters. The wrapper is also necessary to wrap and unwrap values from their container types. For example, if a native method returns a primitive integer, the calling code will expect that the return value will actually be a BoxedTaintedInteger (rather than the primitive integer that it would normally return).

Java supports reflection, a feature that allows code to dynamically access and invoke classes and methods. PHOSPHOR patches all reflective calls to propagate taint tags as necessary, following the exact same semantics used for regular method calls and field accesses. PHOSPHOR also patches calls that inspect the fields and methods that exist in classes to hide any artifacts of the taint tracking process, removing additional fields and arguments as applicable.

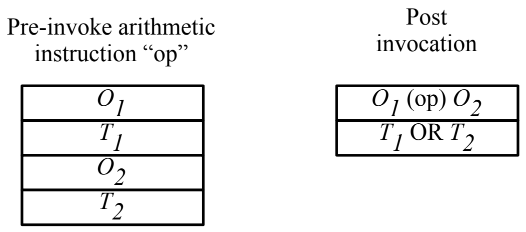

Figure 3: Operand stack before and after performing two-operand arithmetic. The actual operands are shown as O, and their taint tags as T.

```
public static Integer valueOf(int i) {
  assert IntegerCache.high >= 127;
  if (i >= IntegerCache.low && i <=
     IntegerCache.high)
    return IntegerCache.cache[i + (-
     IntegerCache.low)];
  return new Integer(i);
}
```

Figure 4: Java's Integer.valueOf method, a very commonly used method with an indirect data flow caused by caching. If the input is between IntegerCache.low and IntegerCache.high, the output will have no taint tag, even if the input did. PHOSPHOR uses a special case to patch it.

4.4 Java-Specific Features

While our taint tracking process is generic to any language running in the JVM, we found that its support of Java could be significantly enhanced with several optimizations and modifications. For instance, both JVMs that we evaluated (OpenJDK and Oracle's HotSpot JVM) make implicit assumptions about the internal structure of several classes (notably the super-type: `java.lang.Object`, and several of the classes internally used as containers for primitive types: `java.lang.Character`, `java.lang.Byte`, `java.lang.Boolean`, and `java.lang.Short`), which would prevent PHOSPHOR from adding taint storage fields to these classes. PHOSPHOR does not track taint tags on raw instances of the class `java.lang.Object`, which has no fields itself, and therefore, we do not believe is relevant in data flow analyses. For the four restricted primitive container types, PHOSPHOR instead stores the taint tag for instances of these types in a HashMap (similar to the technique used by [40]), hence avoiding the need to modify the internal structure of the class. Storing taint tags in a HashMap is much slower than as individual variables (when using it to store all taint tags, [40] showed a slowdown of up to 526x).

We also make a small modification to support a very commonly used indirect data flow in Java. Primitive container types can be very frequently used in Java, and are used within the JVM when necessary to represent a primitive value as an instance of a reference type. For efficiency, for each primitive type there is a cache of instances of the container class for all low values of that type. Listing 4 reproduces the code used to fetch an instance of class `Integer`. Due to the implicit flow in lines 3-4, if an integer is found in the cache, then its taint tag is dropped. If the integer does not exist in the cache, then the taint tag will be propagated into the new instance of `Integer` in line 5. PHOSPHOR modifies the code that calls the `valueOf` method for each of the primitive container types to ensure that if the primitive argument has a non-zero taint tag, a new instance of the container is created with the tag, hence continuing to propagate taints.

4.5 Optimizations

The entire instrumentation process is implemented in a stream-processing manner: for each byte code instruction, PHOSPHOR outputs new instructions, without context of instructions that previously were output, or those that will be output next. After the instrumentation process, we add several short optimization passes to provide a small amount of context to PHOSPHOR, greatly reducing the size of outputted methods.

First, PHOSPHOR detects instances where taint tags may be loaded to the stack, then immediately popped: for instance, variables loaded to the operand stack and used as operands for jump conditions. PHOSPHOR simply ignores loading the taint tags in these places.

Next, PHOSPHOR detects large methods that perform no instructions other than to load constants into arrays. Rather than initialize the taint tag for each constant as each constant is loaded, PHOSPHOR instead reasons that all tags will be 0, and can instead rapidly initialize them all at once, rather than initializing them one-by-one. This optimization was necessary in several cases in order to ensure that the generated methods remained within the maximum method size (64 kilobytes; this limitation is based on the size of the JVM's internal program counter).

Finally, after all instrumentation has been completed, PHOSPHOR scans each generated method for simplifications. For example, given our rules outlined in the previous section, for any method that returns a primitive value, instructions are inserted after its call site to unbox the taint tag and return value from the return container. However, if both of those values will be immediately discarded from the stack (i.e. pop'ed), then we can simplify the instructions that load and then discard the return value and return taint tag to simply not load the value or tag.

To some extent, these optimizations can also be achieved by the JIT compiler as it compiles the byte code, but we have found that performing them in advance still improves run time (and in some cases, is necessary to ensure that the generated code fits within the maximum method size).

4.6 Application to Android and Dalvik

Although we designed PHOSPHOR for the JVM, we recognized that it could also be applicable to the language virtual machine used by Android, the Dalvik Virtual Machine (DVM). Nearly all applications for Android devices are written in Java, which is then compiled to Java byte code and translated into the DVM's form of byte code, called dex.

Because it executes a translated form of Java byte code, and PHOSPHOR operates at the byte code level, we can apply PHOSPHOR to Android and the DVM by inserting taint propagation logic in the intermediate Java byte code before it is translated to dex. PHOSPHOR could even be applied without needing this intermediate Java byte code, by using a tool such as [15], which translates dex byte code back into Java byte code. Note that although it runs a translated form of Java byte code, the DVM should not be confused with a JVM; our primary target remains the JVM, and any modifications to the DVM or access to intermediate compiled code described in this subsection are unnecessary for JVM taint tracking.

There are many optimizations that the DVM performs beyond those of the JVM, perhaps due to the tight vertical integration of Android devices (from operating system to interpreter to language to APIs and applications). Several of these optimizations pose significant challenges for PHOSPHOR, as they significantly increase coupling between the interpreter and other classes, beyond those discussed in §4.4. Notably, the DVM provides very efficient native implementations of the `java.lang.String` methods `charAt`, `compareTo`, `equals`, `fastIndexOf`, `isEmpty` and `length`. These implementations rely on compile-time knowledge of the run-time organization of the class `java.lang.String` (i.e. the byte-level offsets of each field). Further, the DVM assumes in several cases that all internal primitive container types (not just the several assumed by the JVMs evaluated) contain only a single field containing the primitive value, and no other fields. While we could in principle support taint tracking instances of these classes by storing their taint tag in a HashMap (as for the several classes similarly restricted in the JVMs evaluated), doing so for all of the tightly coupled classes would have posed a prohibitive overhead.

Instead, we made several very small modifications to the Dalvik VM to decouple the VM from the implementation of these classes. Note that although we chose to modify the DVM in this case, the number of changes is significantly smaller than those necessary for TaintDroid, as we are not modifying the interpreter to perform taint tracking, but only to decouple it. These changes required modifying seven constants defined in header files, and modifying six lines of native code that handle reflection. In comparison, the most recent version of TaintDroid (4.3.1) contains a total of over 32,000 lines of new code in the Dalivk VM (as reported by executing a diff of the repository), of which over 18,000 are in assembly code files, and 10,661 in C source code files.

5. Related Work

Dynamic taint analysis is a problem widely studied, with many different systems tailored to specific purposes and languages. For instance, there are several system-wide tainting approaches based on modifications to the operating system ([35] and others). However, PHOSPHOR tracks taint tags by instrumenting application byte code. This general approach is most similar to other approaches that track taint tags by instrumenting application binaries. When available, we compared the Java-based systems directly to PHOSPHOR (an evaluation presented in Section 6), but please note that the performance overheads reported in this section are to provide ballpark information only — the selection of benchmarks used varies greatly from system-to-system (the slowdowns reported here are provided by the original authors).

DyTan is a general purpose taint tracking system targeting x86 binaries that supports implicit (control) flow tainting, in addition to data flow tainting, with runtime slowdown ranging from 30x-50x [13] (where a slowdown of 1x means that the system now takes twice as much time to run). TaintTrace only performs data flow tainting (like PHOSPHOR), and achieves an average slowdown of 5.53x [11]. Libdft, another binary taint tracking tool, shows overheads between 1.14x-6x, thanks to optimizations largely based on assumptions that data (overall) will be infrequently tainted [23]. In contrast, PHOSPHOR does not assume that variables are mostly not tainted (and hence does not make such optimizations, although they mostly are still applicable to the JVM), and therefore its performance will remain constant regardless of the frequency of tainting.

Another general class of taint tracking systems target interpreted languages and make modifications to the language interpreter, targeting, for example, JavaScript [41], Python [43], PHP [29, 33, 43], Dalvik [16] and the JVM [9, 28]. In general, interpreter level approaches can benefit from additional information available in the context of the language that defines the exact boundary of each object in memory (so soundness and precision can be improved over binary-level approaches). The portability of these systems is often restricted, as they require modifications to the language interpreter and/or modifications to application source code.

Of these interpreter-based taint tracking systems, the most relevant to PHOSPHOR are Trishul [28], an approach by Chandra et al. [9], and TaintDroid [16]. Trishul performs data and control flow taint tracking by modifying the Kaffe interpreted JVM, an open source JVM implementation (in a purely interpreted mode, with no JIT compilation — adding an inherent slowdown of several orders of magnitude). Chandra et al. modifies the Jikes Research Virtual Machine to perform data and control flow taint tracking, showing slowdowns of up to 2x on micro-benchmarks, but its implementation depends on the usage of the research VM, rather than a more popularly deployed JVM [9]. Neither the Jikes nor the Kaffe JVM support the complete Java language specification. TaintDroid is a popular taint tracking system for Android's Dalvik Virtual Machine (DVM), implemented by modifying the Dalvik interpreter [16]. TaintDroid only maintains a single taint tag for every element in an array (unlike PHOSPHOR, which maintains a tag for each element), allowing TaintDroid to perform more favorably on array-based benchmarks, but at the cost of precision.

While all of these approaches employ variable-level tracking, like PHOSPHOR, the key difference that sets PHOSPHOR apart is its portability: each of the above systems requires modifications to the language interpreter. For example, TaintDroid's most recent version (version 4.3 at time of publication) adds over 32,000 lines of code to the VM (as measured by lines of code in the TaintDroid patch to Android 4.3.1). For any new release of the VM, the changes must be ported into the new version and if a researcher or user wished to use a different VM (or perhaps a different architecture), they would need to port the tracking code to that VM. PHOSPHOR, on the other hand, is designed with portability in mind: PHOSPHOR runs within the JVM without requiring any modifications to the interpreter (and we show its applicability to the popular Oracle HotSpot and OpenJDK IcedTea JVMs). This design choice also allows us to support Android's Dalvik Virtual Machine with only minor modifications, as discussed in §4.6.

There have been several recent works in dynamic taint tracking for Java that operate by modifying core Java libraries to track taint tags. Without requiring interpreter modification, WASP detects and prevents SQL injection attacks in Java by using taint tracking with low overhead (1-19%), but is restricted to only track taint tags on Strings [22], much like the earlier Java tainting system by Haldar et al. [21], and Chin et al's optimized version of the same technique [12]. These systems simply provide a replacement for the class, `java.lang.String` that is manually modified to perform taint tracking for those objects, and the approach is therefore unsuited to general purpose taint tracking (aside from Strings). PHOSPHOR differs from all of these approaches in that it tracks taints on all forms of data within the JVM: not just Strings.

Vitasek et al. propose a solution to a problem related to taint tracking: in addition to assigning labels to each object in the JVM, their ShadowData system can also enumerate all such labels [40]. Vitasek et al. evaluated several approaches to this, finding the most efficient to be storing the mapping from object to label in a HashMap, showing slowdown ranging from 4.8x-185.5x, largely due to contention in accessing that HashMap, a drawback that PHOSPHOR's decentralized taint tag storage avoids (but note that PHOSPHOR does not provide the ability to enumerate all data that is tagged).

6. Evaluation

We evaluated PHOSPHOR in the dimensions of performance (as measured by runtime overhead and memory overhead) and in soundness and precision. We have also compared the performance of PHOSPHOR with that of TaintDroid, when running within the Dalivk VM on an Android device. We were restricted from comparing against other taint tracking systems, as many were unavailable for download and did not utilize standardized benchmarks in their evaluations. All of our JVM experiments were performed on an Apple Macbook Pro (2013) running Mac OS 10.9.1 with a 2.6Ghz Intel Core i7 processor and 16 GB of RAM. We used four JVMs: Oracle's "HotSpot" JVM, version 1.7.0_45 and 1.8.0 and the OpenJDK "IcedTea" JVM, of the same two versions. All instrumentation was performed ahead of time and the dynamic instrumenter therefore only needed to instrument classes that were dynamically generated (for example, by the Tomcat benchmark, which compiles JSP code into Java and runs it).

For all experiments, no other applications were running and the system was otherwise at rest. All of our Android experiments were performed on a Nexus 10, running Android version 4.3.1, built from the Android Open Source Project repository. No other applications were running on the Android device during our experiments.

6.1 Performance: Macro benchmarks

Our first performance evaluation focused on macro benchmarks, from the DaCapo [5] benchmark suite (9.12 "bach"), and the Scalabench [32] benchmark suite (0.1.0-20120216). The DaCapo benchmark suite contains 14 benchmarks that exercise popular open source applications with workloads designed to be representative of real-world usage. Several of these workloads are highly relevant to taint tracking applications, as they benchmark web servers: the "tomcat," "tradebeans" and "tradesoap" workloads. The Scalabench suite contains 12 benchmarks written in Scala that are also broad in scope. In all cases, we used the "default" size workload.

First, we ran the benchmarks using both the Oracle "HotSpot" JVM and the OpenJDK "IcedTea" JVM in our test environment to measure baseline execution time. Then, we instrumented both JVMs and all of the benchmarks to perform taint tracking, and measured the resulting execution time and the maximum heap usage reported by the JVM. To control for JIT and other factors, we executed each benchmark multiple times in the same JVM until the coefficient of variation (a normalized measure of deviation: the ratio of the standard deviation of a sample to its mean) dropped to at most 3 over a window of the 3 last runs (a technique recommended in [18]). Our measurements were then taken in the next execution of the benchmark in that JVM. This process was repeated 10 times, starting a new JVM to run each experiment, and we then averaged these results.

We include results for all benchmarks except for the "scalac" benchmark from the scalabench workloads, a benchmark that exercises the Scala compiler. The Scala compiler has certain expectations about the structure and contents of class files that it compiles, so injecting taint tracking code into the compiler itself (plus the intermediate code that is being compiled) causes runtime errors. A general limitation of our approach is that applications that inspect their own byte code directly (rather than that code being read and interpreted by the JVM, and rather than using Java's reflection interface to inspect it) may not function correctly, as we have changed that byte code (a limitation discussed in §3.3).

Table 1 presents the results of this study, showing detailed results for Oracle's HotSpot JVM (version 7), and summary results for HotSpot 8, and OpenJDK's IcedTea JVMs (versions 7 and 8). We focus on the results for HotSpot 7, as it is far more widely adopted than version 8 (at time of submission, Java 7 was approximately three years old, and Java 8 was approximately one week old). Using Oracle's HotSpot JVM 7, for the DaCapo suite, the average runtime overhead was 51.9%, and across the Scalabench suite, the average runtime overhead was 55.1% (runtime overhead for other JVMs is shown in Table 1). The average heap overhead was 239.1% for DaCapo, and 311.5% for Scalabench (heap usage in the other JVMs was similar). This heap overhead is unsurprising: in addition to requiring additional memory to store the taint tags, PHOSPHOR also increases memory usage by its need to allocate containers to box and unbox primitives and primitive arrays for return values, and primitive arrays when casting them to the generic type `java.lang.Object` (as discussed in §4.1.2).

There are several interesting factors that can contribute to the heap overhead growing to be more then twice as large. First, note that a Java `integer` is four bytes, while a `byte` is 1 byte, and `char`s and `short`s are both two bytes. Therefore, the space overhead to store the taint tag for a variable can be as high as 4x.

The second factor that can adversely impact heap overhead comes from our container types. For every method that returns a primitive type, we replace its primitive return type with an object that wraps the primitive value with its taint tag. Although we pre-allocate these return types and attempt to reuse them, our implementation will only allow for reuse when (1) a method calls multiple other methods that return the same primitive type, or (2) a method calls other methods that return the same primitive type as the caller. These allocations are relatively cheap in terms of execution time (and are represented in our overall execution overhead measures), but can put significant pressure on the garbage collector that wouldn't exist without PHOSPHOR, as primitive values are not reference-tracked. We saw a particularly heavy allocation pattern in the *xalan* benchmark, where approximately 36 million instances of `TaintedInt` and 35 million

		Oracle Hotspot 7						Other JVMs		
		Runtime (ms)			Heap Size (MB)			Runtime Overhead		
	Benchmark	T_b	T_p	Overhead	M_b	M_p	Overhead	HotSpot 8	IcedTea 7	IcedTea 8
DaCapo 9.12-bach [5]	avrora	2333 ± 53	2410 ± 27	3.3%	75	223	198.8%	.7%	3.8%	3.6%
	batik	903 ± 15	1024 ± 15	13.5%	105	211	100.2%	12.1%	N/A*	N/A*
	eclipse	15305 ± 702	48907 ± 1885	219.6%	1026	2901	182.7%	138.8%	209.8%	124.0%
	fop	203 ± 6	320 ± 7	57.7%	100	261	162.0%	63.3%	57.4%	49.8%
	h2	3718 ± 136	5137 ± 138	38.2%	739	2738	270.5%	34.0%	34.7%	35.2%
	jython	1343 ± 19	2107 ± 47	56.9%	412	805	95.1%	25.7%	59.4%	26.8%
	luindex	454 ± 50	642 ± 44	41.6%	39	157	303.6%	52.9%	44.4%	53.2%
	lusearch	584 ± 65	1126 ± 73	92.8%	619	2750	344.2%	86.6%	102.0%	92.6%
	pmd	1336 ± 20	1705 ± 56	27.6%	172	583	239.5%	26.8%	29.8%	23.5%
	sunflow	1616 ± 76	2182 ± 231	35.0%	532	1086	104.3%	28.8%	28.2%	29.1%
	tomcat	1364 ± 35	1885 ± 41	38.2%	173	881	410.7%	33.4%	30.0%	36.8%
	tradebeans	3175 ± 94	4189 ± 136	31.9%	1093	2225	103.6%	33.3%	41.4%	34.3%
	tradesoap	12159 ± 2416	14657 ± 2470	20.6%	1910	3058	60.1%	17.5%	14.1%	3.6%
	xalan	498 ± 40	748 ± 102	50.2%	91	790	771.9%	49.2%	38.5%	75.7%
	Average	3214	6217	51.9%	506	1334	239.1%	43.1%	53.4%	45.2%
Scalabench 0.1.0-20120216 [32]	actors	2523 ± 103	2663 ± 130	5.5%	90	716	692.0%	4.0%	.6%	3.5%
	apparat	7874 ± 640	13516 ± 1102	71.7%	509	2430	377.0%	102.6%	66.7%	92.8%
	factorie	19262 ± 1812	25063 ± 781	30.1%	2769	2791	.8%	38.7%	32.1%	35.5%
	kiama	238 ± 5	381 ± 11	60.3%	151	529	250.7%	51.1%	52.9%	59.7%
	scaladoc	1092 ± 25	2206 ± 85	102.1%	174	1225	602.7%	98.0%	94.0%	94.0%
	scalap	136 ± 6	227 ± 7	67.3%	86	298	248.8%	82.1%	61.6%	82.3%
	scalariform	419 ± 15	523 ± 8	24.6%	88	304	246.1%	28.9%	21.8%	23.5%
	scalatest	840 ± 55	1133 ± 73	34.9%	153	599	292.1%	45.4%	32.8%	41.9%
	scalaxb	288 ± 6	540 ± 38	87.8%	87	413	373.6%	218.8%	79.9%	219.2%
	specs	1268 ± 44	1770 ± 21	39.6%	162	714	340.8%	24.6%	36.5%	40.6%
	tmt	3755 ± 65	6834 ± 33	82.0%	2733	2777	1.6%	95.0%	81.5%	93.5%
	Average	3427	4987	55.1%	637	1163	311.5%	71.7%	50.9%	71.5%
All Average		3307	5676	53.3%	563	1259	270.9%	55.7%	52.2%	57.3%

Table 1: Runtime duration for macro benchmarks, showing baseline time (T_b), PHOSPHOR time (T_p) and relative overhead for Oracle's HotSpot JVM version 1.7.0_45, indicating standard deviation of measurements with $\pm$. We also show heap size measurements for the baseline execution (M_b) and PHOSPHOR execution (M_p), as well as the percent overhead for heap size. For HotSpot 8, IcedTea 7 and IcedTea 8, we show only runtime overhead. *The "batik" benchmark depends on Oracle-proprietary classes, and therefore does not execute on the OpenJDK IcedTea JVM.

instances of `TaintedBoolean` were allocated to encapsulate return types.

In terms of runtime overhead, we saw the best performance from PHOSPHOR in the "avrora" benchmark, and worst performance in the "eclipse" benchmark. The "avrora" benchmark runs a simulator of an AVR micro controller, and from our inspection, contains many primitive-value operations. We believe that it was a prime target for optimization by the JIT compiler; indeed, when disabling the JIT compiler and running the benchmark in a purely interpreted mode, we saw an 87% overhead, much more in line with the average performance of PHOSPHOR. "Eclipse" represents a greater mix of operations that are more complicated and computationally expensive for PHOSPHOR to implement. For instance, many parts of the Eclipse JDT Java compiler (a component of the benchmark) store primitive arrays into fields declared with the generic type, `java.lang.Object`. For every access to these fields, PHOSPHOR must insert several instructions to box or unbox the array, which requires allocating a new container each time, and hence, adding significantly to the overhead.

To compare broadly to other binary-instrumentation based taint tracking systems, DyTan [13] shows a performance overhead of 30x in a macro benchmark, with a memory overhead of 240x. LibDFT [23] shows a performance overhead of 1.14-6x on macro benchmarks. PHOSPHOR showed an average overhead of 1.5x, ranging overall from 1.03x to 3.19x. Again, it is impossible to compare directly to these

Benchmark	Oracle - HotSpot 7			Rel. Overhead (Other JVMs)			Rel. Overhead (DVM)	
	T_b (ns)	T_p (ns)	Rel. Overhead	Hotspot 8	IcedTea 7	IcedTea 8	PHOSPHOR	TaintDroid
Float	5620 ± 25	7765 ± 47	38.2%	108%	37.7%	114.2%	131.2%	63.9%
Logic	1338 ± 4	1341 ± 5	0.2%	-0.3%	-0.6%	0.2%	1.5%	11.2%
Loop	3283 ± 59	4060 ± 38	23.7%	23.4%	22.2%	23.3%	43.9%	64%
Method	266 ± 5	642 ± 4	141.3%	140.1%	132.4%	137.6%	9.8%	25.3%
Sieve	6128 ± 42	7062 ± 87	15.2%	12.8%	14.3%	13.6%	27.7%	3.2%
String Buffer	1081 ± 7	3396 ± 43	214.2%	212.9%	215.9%	208.4%	183.3%	30.8%
Average	2953	4044	72.1%	82.8%	70.3%	82.9%	66.2%	33.1%

Table 2: Runtime duration (in nanoseconds) and overhead for micro benchmarks, showing baseline time (T_b), PHOSPHOR time (T_p) with standard deviation as $\pm$, and relative overhead for Oracle's HotSpot JVM version 1.7.0_45 and 1.8.0, OpenJDK's IcedTea JVM version 1.7.0_45 and 1.8.0, and Android's DVM version 4.3.1. For the DVM, we also show TaintDroid's overhead (relative to the same baseline Android configuration).

systems, as they target different platforms (i.e., not the JVM) and there was no standard benchmark that we could use for the purpose.

The most applicable systems to compare PHOSPHOR to are TaintDroid [16], Trishul [28] and Chandra et al's approach [9]. Of these, we were able to obtain TaintDroid and Trishul (the authors of [9] were unable to find their implementation [8]), but were unable to use our macro benchmarks to compare to these systems as the benchmarks are not supported by Dalvik and Kaffe respectively (the VMs used by Trishul and TaintDroid). The authors of TaintDroid used the CaffeineMark [30] benchmark in their evaluation, and the authors of Trishul used the jMocha benchmark [19] in their evaluation. We compare PHOSPHOR's performance directly to TaintDroid and Trishul in the following section.

6.2 Performance: Micro Benchmarks

We performed a series of micro benchmarks to further analyze PHOSPHOR's runtime performance overhead. Our micro benchmarks are based on the CaffeineMark [30] suite of micro benchmarks, commonly used by Android developers – including by the authors of TaintDroid [16]. We modified these benchmarks to run under Google's Caliper micro benchmark tool, so that they could benefit from the framework's warmup, timing, and validation features (the original CaffeineMark benchmarks do not contain any warmup phase and therefore the results can be skewed by JIT compilation). The "embedded" suite (used in the TaintDroid study) consists of six benchmarks: "Float" (simulates 3D rotation of objects around a point; uses arrays), "Logic" (contains many simple branch conditions), "Loop" (contains sorting and sequence generation; uses arrays), "Sieve" (uses the sieve of eratosthenes to find primes; uses arrays), "Method" (features many recursive method calls) and "String" (performs string concatenation; uses arrays). Each benchmark was executed several times in the same JVM over the course of 3 seconds to warm up, and then executed for a period of 1 second. For that last second, we measure the amount of time in nanoseconds that each benchmark took (by running it many times and averaging). We did this entire process 10 times, and averaged the results of each trial.

Table 2 shows the results of this study, showing the runtime for PHOSPHOR for Oracle's HotSpot 7 JVM (being the most popular JVM at time of publication), and the runtime overhead for all of the subject JVMs, plus the Android DVM. We also show our measured overhead of TaintDroid, relative to the same baseline Android DVM. PHOSPHOR's fine-grained array taint tag tracking (i.e. that it stores a taint tag per-element, rather than a single tag per-array) caused it to perform somewhat poorer than TaintDroid in the benchmarks that relied heavily on arrays. Recall that this optimization will result in a loss in precision for TaintDroid, which does not affect PHOSPHOR.

However, in the benchmarks that did not involve significant array usage (e.g. "Logic," "Loop," and "Method"), PHOSPHOR outperformed TaintDroid. It would be interesting to perform a followup study by modifying TaintDroid to also track taint tags per-element, to see which approach is faster in that case. Another interesting observation from the micro benchmarks is that the average overhead across these micro benchmarks for PHOSPHOR (72.13%), is somewhat higher than its average overhead across the macro benchmarks (52.06%). Perhaps these less than optimal cases occur less in practice than those cases wherein PHOSPHOR is faster. Unfortunately we are severely restricted in availability of macro benchmarks for Android (DaCapo is not easily ported to Android as many of its benchmarks rely on Java APIs that are not included in the Android Dalvik VM), and therefore could not perform a macro benchmark study comparing TaintDroid with PHOSPHOR.

To compare to Trishul [28], we hoped to use the same suite of micro-benchmarks (the suite used by the authors of Trishul, jMocha, is no longer available) used above. However, we found that the benchmark framework that we used to collect timing information (Google Caliper version 0.5) was incompatible with Kaffe, the JVM that Trishul is built

	Relative Overhead to HotSpot 7		
Benchmark Group	Kaffe	Trishul	PHOSPHOR
Arithmetic	64.4%	74.5%	10.7%
Assign	56.5%	86.6%	50.2%
Cast	87.1%	86.7%	13%
Create	98.5%	98.8%	24.4%
Exception	90.0%	69.9%	1.7%
Loop	2.3%	89.0%	7.6%
Math	89.1%	96.5%	96.0%
Method	42.2%	76.0%	6.3%
Serial	90.04%	N/A*	21.14%
Average	68.9%	84.7%	25.7%

Table 3: Runtime overhead of PHOSPHOR, Kaffe 1.1.7 [37] and Trishul [28] compared to Oracle HotSpot 1.7.0_55 on the JavaGrande benchmark [7]. *Threw exception

upon. Therefore, we selected another suite of micro benchmarks to run for this purpose: JavaGrande [7], a micro benchmark suite from 2000, which was popular at the time (and worked with Kaffe). We performed these experiments in an Ubuntu 6.10 VirtualBox VM with 3.5GB of RAM (running on the same MacBook Pro 2.6Ghz Intel Core i7, 16 GB of RAM) that was provided by the Trishul authors. We measured the performance of Oracle's HotSpot 7 running within this VM as a baseline, and then also measured the performance of Kaffe 1.1.7 (which Trishul is based on), Trishul, and a PHOSPHOR-instrumented HotSpot 7. Again, we executed each benchmark 10 times (each time in a separate JVM), but here, with no warmup phase, as JavaGrande's benchmark runner does not support a warmup phase (and we were unable to use Google Caliper to control for warmup as it was not supported by Kaffe VM).

Table 3 presents the results of this evaluation (overheads presented are relative to HotSpot 7). Note that in most cases, Kaffe itself (the VM that Trishul is built on) is significantly slower relative to HotSpot, and hence, perhaps some large amount of the overhead imposed by Trishul can be attributed to the underlying VM. In all cases, PHOSPHOR had significantly lower overhead than Trishul (in the case of the "Serial" benchmark, Trishul threw an exception and was unable to execute the benchmark). Trishul's performance on the loop and method benchmarks was particularly poor (even relative to an unmodified Kaffe VM), likely due to the fact that it performs control flow tainting, and not just data flow tainting. Adding control flow tainting to PHOSPHOR would likely also increase its overhead in these two benchmarks.

6.3 Soundness and Precision

We evaluated the soundness and precision of PHOSPHOR using two benchmark suites. First, we wrote our own suite of unit tests, testing that each of our taint tracking properties (as described in §4.1) are not violated, for each primitive and primitive array type, as well as for reference types. PHOSPHOR passed all of these tests. These unit tests are included in our GitHub repository [4].

To add additional validity to our claim that PHOSPHOR is sound and precise, we also implemented the DroidBench [2] taint tracking benchmark, removing the components that were Android specific so that it would run on a desktop JVM. DroidBench consists of 64 test cases for taint tracking systems, of which, we found 35 to be Android-specific (testing taint propagation through Android-specific callbacks and life-cycle events), leaving 29 tests. These tests are designed to test both soundness (that variables that should be tainted are indeed tainted with the correct taint) and precision (that variables that should not be tainted are not tainted) of taint tracking. Four of the tests are designed to test taint tracking through implicit flows. PHOSPHOR passed all data flow tests and failed on the four implicit flow tests as expected.

6.4 Portability

We further studied the portability of PHOSPHOR by attempting to apply it to three completely different JVMs (in addition to the two versions of Oracle's HotSpot and OpenJDK's IcedTea, plus the Dalvik DVM). We downloaded the most recent versions of the Apache Harmony JVM (version 6.0M3) [1], Kaffe VM (version 1.1.9) [37] and Jikes RVM (version 3.1.3) [36]. For each VM, we attempted to execute our soundness and precision tests as a basic indicator of whether PHOSPHOR would work.

While PHOSPHOR did not work immediately with Harmony or Kaffe, after approximately 30 minutes of debugging, we identified several additional classes that were tightly coupled between the class library and the interpreter. For instance, no JVM that we tested allowed for unrestricted modifications of the class `java.lang.Object`; Harmony and Kaffe similarly would not allow for modifications of the class `java.lang.VMObject` (which does not exist in Oracle or OpenJDK's class library). We patched around these classes, and can confirm that PHOSPHOR works with Harmony and Kaffe.

However, we were unable to successfully apply PHOSPHOR to the Jikes RVM, which is a JVM implemented in Java. We believe that this is due to our inherent design limitation (discussed further in the following section), that should an application try to read its own byte code, it will see unexpected entires (namely, everything added by PHOSPHOR). Jikes uses its own internal implementation of Java's reflection library for configuring its bootstrap class image, and PHOSPHOR does not currently patch this to hide its modifications, causing it to fail. We believe that it would be possible to modify PHOSPHOR to be compatible with Jikes, but have not investigated this further.

6.5 Threats to Validity

The main threats to validity to our experiments are related to our claims of portability. We claim that PHOSPHOR is portable to any JVM that fulfills the official JVM specifications versions 7 and 8, as it only requires modifications to application byte code and library byte code. We evaluated this claim on four JVMs, including two versions of two very widely used JVMs (Oracle HotSpot and OpenJDK IcedTea), and two much less frequently used JVMs (Kaffe and Harmony). Just as these JVMs had tight coupling for several classes, preventing PHOSPHOR from adding fields to them to track taint tags, it is certainly possible that other JVMs have even more constraints on more classes (such coupling between class libraries and interpreter are not discussed in the JVM specification). However, we are confident that if such cases arose, PHOSPHOR would still be applicable, falling back to storing taint tags for instances of such classes with a HashMap, an approach that would still work, though perhaps with somewhat higher overhead (such changes would need to be manually implemented). We believe that PHOSPHOR's incompatibility with the Jikes Research Virtual Machine is an exceptional case in that (1) it is intended specifically for research purposes and not production purposes, (2) it is written in Java itself and is self-hosted (i.e. its Java code runs on itself). Moreover, although we were unable to find any usage statistics, we believe that Oracle HotSpot and OpenJDK IcedTea dominate the JVM market by far.

Although we selected popular, well-accepted macro benchmarks for evaluating PHOSPHOR, it is possible that the selected benchmarks are not representative of the sorts of workloads that would normally are targets for taint tracking. However, because three of these benchmarks involve workloads on web servers, and taint tracking has been shown to be highly applicable to detecting and preventing code injection attacks in web servers, we believe that the benchmarks are sufficient.

There are several key limitations to our approach, as discussed previously in §3.3, most notably that PHOSPHOR only tracks data flows, and not control flows ("implicit flows"), much like other well known taint tracking systems [10, 16, 23]. Note that implicit flow tracking primarily requires static analysis, and its implementation should be unaffected by PHOSPHOR's approach to data flow tracking. Support for implicit flows would be interesting to add as an optional feature to PHOSPHOR (e.g. DyTan [13] supports both sorts of tracking), but we consider this to be future work, outside of the scope of this paper.

Java provides a simple reflection API (also used by many Scala applications) to access information about class files, such as the list of methods available in a class. PHOSPHOR patches this API to hide all of its changes from applications, however, if an application directly reads in the byte stream of a Class file (without using this API) and parses its structure, that application will find potentially unexpected artifacts of PHOSPHOR in the Class. This scenario arose in our macro benchmark study exactly once: in the case of the Scala compiler ("scalac"), which does not use the reflection API. We do not believe that this is a common occurrence outside of the scope of compilers, as Java's reflection API is widely used for this purpose.

7. Conclusions

Due to difficulties simultaneously achieving precision, soundness, and performance, all previous implementations of dynamic taint analysis for JVM based languages have been restricted, functioning only within specialized research-oriented JVMs, making their deployment difficult. We presented PHOSPHOR, our approach to providing accurate, precise, and performant taint tracking within the JVM without requiring any modifications to it, demonstrating its applicability to two very popular JVMs: Oracle's HotSpot and OpenJDK's IcedTea, each for the two most recent versions: 1.7 and 1.8. Moreover, PHOSPHOR does not require any specialized operating system or specialized hardware or access to application source code. PHOSPHOR is released with an open-source license via GitHub [4], and we hope that it can be used by other researchers to further their work in fields such as security, debugging and testing.

8. Acknowledgments

The authors would like to thank Riley Spahn, Roxana Geambasu and Kangkook Jee for their valuable feedback and advice regarding PHOSPHOR. The authors would also like to thank Srijith Nair for his assistance with running Trishul, and the anonymous reviewers for their valuable feedback. The authors are members of the Programming Systems Laboratory, which is funded in part by NSF CCF-1302269, CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

References

[1] Apache Software Foundation. Apache harmony - open source java platform. `http://harmony.apache.org`.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[3] M. R. Azadmanesh and M. Sharifi. Towards a system-wide and transparent security mechanism using language-level information flow control. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[4] J. Bell and G. Kaiser. Phosphor: Dynamic taint tracking for the jvm. `https://github.com/Programming-Systems-Lab/phosphor`.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks:

Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, New York, NY, USA, 2006. ACM.

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

[7] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking java grande applications. In *in Proceedings of ACM 1999 Java Grande Conference*, pages 81–88. ACM Press, 1999.

[8] D. Chandra. Personal Communication (Email). July 10, 2014.

[9] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 463–475, Dec 2007.

[10] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS '08*, pages 39–50, New York, NY, USA, 2008. ACM.

[11] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, Washington, DC, USA, 2006. IEEE.

[12] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09. ACM, 2009.

[13] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07*. ACM, 2007.

[14] CVE Details. Vulnerability distribution of cve security vulnerabilities by types. `http://www.cvedetails.com/vulnerabilities-by-types.php`.

[15] Dex2Jar Project. dex2jar - tools to work with android .dex and java .class files - google project hosting. `https://code.google.com/p/dex2jar/`.

[16] 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'10*, Berkeley, CA, USA, 2010. USENIX Association.

[17] M. Ganai, D. Lee, and A. Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *FSE '12*, pages 46:1–46:11, New York, NY, USA, 2012. ACM.

[18] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[19] E. Gluzberg, E. Gluzberg, S. Fink, and S. Fink. An evaluation of java system services with microbenchmarks. Technical report, 2000.

[20] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *ISSTA '11*, New York, NY, USA, 2011. ACM.

[21] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[22] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14*, pages 175–185, New York, NY, USA, 2006. ACM.

[23] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[24] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, Washington, DC, USA, 2006. IEEE.

[25] 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.

[26] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*, Java SE 7 edition, Feb 2013.

[27] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. Defcon: High-performance event processing with information security. In *Proceedings of the 2010 USENIX ATC*, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[28] 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.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.

[30] Pendragon Software Corporation. Caffeinemark 3.0. `http://www.benchmarkhq.ru/cm30/`, 1997.

[31] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI '09*, pages 63–74, New York, NY, USA, 2009. ACM.

[32] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *OOPSLA '11*, pages 657–676, New York, NY, USA, 2011. ACM.

[33] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *CCS '13*, New York, NY, USA, 2013. ACM.

[34] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA '11*. ACM, 2011.

[35] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, pages 85–96, New York, NY, USA, 2004. ACM.

[36] The Jikes RVM Project. Jikes rvm - project status. `http://jikesrvm.org/Project+Status`.

[37] The Kaffe Team. Kaffe vm. `https://github.com/kaffe/kaffe`.

[38] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. In *PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.

[39] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4), Dec. 2007.

[40] M. Vitásek, W. Binder, and M. Hauswirth. Shadowdata: Shadowing heap objects in java. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 17–24, New York, NY, USA, 2013. ACM.

[41] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*. ACM, 2013.

[42] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[43] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP '09*, pages 291–304, New York, NY, USA, 2009. ACM.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI '06*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

A. Appendix: JVM Byte Code Opcode Reference

PHOSPHOR modifies the operation of many byte code instructions by inserting additional instructions around them. This table lists all byte code instructions supported by the Java Virtual machine, and for each one, a brief description of the change(s) that PHOSPHOR makes.

Table 4: All JVM byte codes, annotated with descriptive transformation information

Opcode(s)	**Brief Description**	**PHOSPHOR Modifications**
aastore	Stores reference to array	Removes the taint tag for the index before storing; if the ref. is to a primitive array, boxes before storing
aaload	Loads reference to array	Removes the taint tag for the index to load
anewarray	Allocates new array for references	If the array type is a mutli-d primitive array, change to a container type
arraylength	Returns length of array as integer	Place the tag "0" just below return val on the operand stack after execution
areturn	Exit a method, returning the object reference at the top of the stack	If the top of the stack is a primitive array, boxes the array and its taint tags before return
astore	Store an object to a local variable	If the variable type is a primitive array, store the taint tags also to their variable. If the variable type is "Object" and the item being stored is a primitive array, box it.
baload, caload, daload, iaload, faload, laload, saload	Loads a value from a primitive array	Removes the taint tag for the index to load; loads the taint tag for the corresponding element too
bastore, castore, dastore, iastore, fastore, lastore, sastore	Stores a value to a primitive array	Removes the taint tag for the index to store to; stores the taint tag for the corresponding element too
bipush, sipush, iconst, lconst, dconst, fconst	Loads a constant to the stack	Loads the taint tag "0" before loading the constant requested
checkcast	Casts the top Object ref	If casting to a primitive array, unbox the boxed primitive array
*X*add, *X*mul, *X*div, *X*rem, *X*sub, *X*and, *X*or, *X*shl, *X*shr, *X*ushr, *X*xor, lcmp, dcmpl, dcmpg	Performs binary-operand math on top two stack elements	Moves taint tags of operands out of way and ORs them, placing new tag just below the result
dload, fload, iload, lload	Load a primitive local variable	Load the taint tag, just before loading the requested variable
dstore, fstore, istore, lstore	Store a primitive local variable	After storing the requested variable, store the taint tag
dup, dup2, dup2_x1, dup2_x2, dup_x1, dup_x2	Duplicates the top N words on operand stack, possibly placing under the third or fourth word	Also duplicates the taint tag (if there is one) and if placing under other elements, places under their taint tag (if present)
dreturn, ireturn, freturn, lreturn	Exit a method, returning the primitive value at the top of the stack	Boxes the primitive into a container, then executes ARETURN instead
getfield, getstatic	Retrieves the value of an instance field of an object	If applicable, also retrieves the taint tag just before performing the getfield/getstatic
if_acmpeq, if_acmpne	Jump if the top two object references on stack are/aren't equal	If either operand is a primitive array, pops the taint tag before executing
if_icmplt, if_icmpge, if_icmple, if_icmple, if_icmpeq, if_icmpne	Compare top 2 ints and jumps	Pops the taint tag for both integers before executing

Table 4: All JVM byte codes, annotated with descriptive transformation information

Opcode(s)	**Brief Description**	**PHOSPHOR Modifications**
ifeq, ifne, ifgt, ifge, ifle, iflt	Compares top 1 int and jumps	Pops the taint tag before executing
ifnonnull, ifnull	Jump if top reference is/isn't null	If operand is a primitive array, pops taint tag before executing
instanceof	Return 0/1 if the top reference is (or isn't) the instance of a requested type	If the operand is a primitive array, pops the taint tag before executing. Inserts the taint tag ”0” just under the result.
invokespecial, invokevirtual, invokeinterface, invokestatic	Invoke a method, popping the arguments from the stack and placing on top the return value	If the callee is a primitive array, pops the taint tag (all cases but invokestatic); Remaps the method descriptor to include taint tags as necessary; If any parameter is of type ”Object” but the type being passed is a primitive array, box it into a container. After return, if return was a container, then unbox it
ldc, ldcw, ldc2_w	Loads a constant onto the stack	If loading a primitive type, load taint tag ”0” on stack first
lookup/table switch	Computed jump	Pops the taint tag of the operand before executing
monitorenter	Obtain lock on the ref. on stack	If the ref. is a primitive array, pops the taint tag before executing
monitorexit	Release lock on the ref. on stack	If the ref. is a primitive array, pops the taint tag before executing
newarray	Create a new 1D primitive array of a given length	Remove the taint for the length of the array; Create a 1D int array of same length to store taint tags before executing.
pop, pop2	Removes the top 1 or 2 words from the stack	If a word being popped is a primitive or primitive array, also remove its taint tag
putfield, putstatic	Stores a value to a field	If the value being stored is a primitive or primitive array, also store taint tag. If storing primitive array to a field of type ”Object” then box it first
swap	Swaps the top two words on the stack	If either operand has a taint tag, then ensure that the tags are swapped with the values
multianewarray	Create (and possibly initializes) a multidimensional array	Removes the taint tag of all operands. If element type is primitive, then changes to a container type, and initializes the last dimension if it would have been otherwise
aconst_null	Loads the constant “null” onto the stack	No modification necessary
athrow	Pops an exception off of the top of the stack and throws it	No modification necessary
d2f, d2i, d2l, f2d, f2i, f2l, i2b, i2c, i2d, i2f, i2l, i2s, l2d, l2f, l2i	Casts primitive types	No modification necessary
dneg, fneg, ineg, lneg	Negates a primitive type	No modification necessary
goto, jsr, ret	Unconditional jump	No modification necessary
new	Creates a new uninitialized object	No modification necessary
return	Returns “void” from a method	No modification necessary
iinc	Increments a local variable	No modification necessary
wide	Indicates that the next instruction accesses a local variable with an index greater than 255	No modification necessary