Dynamic Taint Tracking for Modern Java Virtual Machines

KATHERINE HOUGH*, Northeastern University, United States
JONATHAN BELL, Northeastern University, United States

Dynamic taint tracking is a program analysis that traces the flow of information through a program. In the Java virtual machine (JVM), there are two prominent approaches for dynamic taint tracking: "shadowing" and "mirroring". Shadowing is able to precisely track information flows, but is also prone to disrupting the semantics of the program under analysis. Mirroring is better able to preserve program semantics, but often inaccurate. The limitations of these approaches are further exacerbated by features introduced in the latest Java versions. In this paper, we propose Galette, an approach for dynamic taint tracking in the JVM that combines aspects of both shadowing and mirroring to provide precise, robust taint tag propagation in modern JVMs. On a benchmark suite of 3,451 synthetic Java programs, we found that Galette was able to propagate taint tags with perfect accuracy while preserving program semantics on all four active long-term support versions of Java. We also found that Galette's runtime and memory overheads were competitive with that of two state-of-the-art dynamic taint tracking systems on a benchmark suite of twenty real-world Java programs.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; • **Security and privacy** → *Information flow control.*

Additional Key Words and Phrases: taint tracking, information flow, dynamic analysis

ACM Reference Format:
Katherine Hough and Jonathan Bell. 2025. Dynamic Taint Tracking for Modern Java Virtual Machines. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE079 (July 2025), 22 pages. https://doi.org/10.1145/3729349

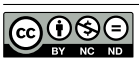
1 Introduction

Dynamic taint tracking traces the flow of information through a program by associating labels, also called "taint tags", with program data and propagating these labels as the program executes. By tracing different types of information, dynamic taint tracking has been used to prevent confidential information from being leaked to public channels [Cox et al. 2014; McCamant and Ernst 2008], detect security vulnerabilities [Halfond et al. 2006; Hough et al. 2020; Kieżun et al. 2009; Qin et al. 2006], assist in automated input generation systems [Ganesh et al. 2009; Kukucka et al. 2022; Rawat et al. 2017; Wang et al. 2010], provide debugging guidance [Attariyan and Flinn 2010; Clause and Orso 2009], and reason about configurable systems [Attariyan and Flinn 2010; Velez et al. 2021].

Java is the second most popular programming language according to the PYPL PopularitY of Programming Language Index as of January 2024 [Carbonnelle 2024] and the fourth most popular programming language according to the 2024 TIOBE Programming Community Index [TIOBE Software BV 2024]. Due to the popularity of Java and other languages that compile to Java bytecode (e.g., Kotlin, Scala, and Clojure), researchers have explored different techniques for performing dynamic taint tracking in the Java Virtual Machine (JVM) [Bell and Kaiser 2014; Chandra and Franz 2007; Franz et al. 2005; Halfond et al. 2006; Nair et al. 2008; Ouyang et al. 2023; Perkins et al. 2020].

*The work described in this manuscript was performed prior to the author's employment with Amazon Web Services.

Authors' Contact Information: Katherine Hough, Northeastern University, Boston, Massachusetts, United States, hough.k@northeastern.edu; Jonathan Bell, Northeastern University, Boston, Massachusetts, United States, j.bell@northeastern.edu.



ACM 2994-970X/2025/7-ARTFSE079
https://doi.org/10.1145/3729349

However, due to changes to the Java platform, these techniques perform poorly on newer Java versions — inducing deviations from an application's normal behavior, inaccurately propagating taint tags, and even causing the JVM to crash in some cases.

As of August 2024, there are four long-term support (LTS) versions of Java that are officially supported by Oracle: 8, 11, 17, and 21 [Oracle Corporation 2023c]. The most recent of these, Java 21, was released in September 2023 [Oracle Corporation 2023c]. The Java community has widely adopted these modern LTS Java versions. New Relic, Inc. [2023] found that 56%, 33%, and 9%, of the applications that reported to them used Java 11, 8, and 21 in production in 2023, respectively.

Given the prevalence of newer Java versions, it is critical that dynamic taint tracking systems support modern Java features. However, existing taint tracking tools fail to address two major features introduced in newer Java versions: *signature polymorphism* and *modules*. The challenges introduced by these features impact other dynamic analyses, e.g., dynamic invariant detectors [Ernst et al. 2007], and have yet to be addressed by existing work. In our empirical evaluation (Section 5), we found that state-of-the-art dynamic taint tracking systems were unreliable and inaccurate when analyzing newer Java applications. To support dynamic taint tracking in modern JVMs, this work makes the following contributions:

- A technique for instrumenting Java classes used in the initialization of the JVM without violating constraints imposed by Java's module system.
- A technique for reasoning about the flow of information through signature polymorphic methods.
- A precise, robust technique for performing full-system dynamic taint tracking that minimizes deviations from an application's normal behavior.
- An open-source dynamic taint tracking system for modern JVMs that uses these techniques: Galette.
- An empirical evaluation of Galette's accuracy, performance, and ability to preserve program semantics compared against two state-of-the-art dynamic taint tracking systems for the JVM: Phosphor [Bell and Kaiser 2014] and MirrorTaint [Ouyang et al. 2023].

In our empirical evaluation (Section 5), we found that Galette was able to propagate taint tags with perfect accuracy while preserving program semantics on all four active LTS versions of Java. By contrast, we observed several cases in which both MirrorTaint and Phosphor exhibited incorrect taint tag propagation. We also observed several instances in which Phosphor introduced deviations from an application's normal behavior. Our analysis of the design of these systems reveals that these flaws are due to fundamental limitations of prior approaches. We found that Galette's runtime and memory overheads were competitive with that of Phosphor and better than that of MirrorTaint. Overall, the mean execution time overheads for Galette, Phosphor, and MirrorTaint across all benchmarks were 7.393, 8.420, and 5593.692, respectively. The mean memory overheads for Galette, Phosphor, and MirrorTaint across all benchmarks were 1.536, 1.379, and 9.971, respectively.

2 Background

2.1 The Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract computing machine that runs Java bytecode programs [Lindholm et al. 2013]. The JVM supports multiple, concurrent threads of execution. Each thread of execution has its own program counter and JVM stack. The JVM stack consists of JVM frames. Each of these frames contains an array of local variables and an operand stack which are used to store values for a particular method invocation. There is one active or "current" frame for each thread; this frame stores values for the method invocation that is currently executing on that

thread [Lindholm et al. 2013]. Java bytecode instructions can manipulate the current frame's local variables and operand stack to store, load, and perform operations on values.

When a method is invoked, a new frame is created for the callee method. The JVM passes any parameters to the callee method by loading them from the top elements of the caller's operand stack and storing them into the new frame's local variables. Then, the new frame becomes the current frame and control is transferred from the call site in the caller method to the entry point of the callee method. When the callee method completes, the current frame passes the return value of the method invocation, if any, to the top of the previous frame's operand stack. The current frame is then destroyed, the previous frame is made current, and control is transferred back to the caller.

In addition to local variables and the operand stack, the "heap" is also used to store values. The heap is shared between all threads and contains objects and arrays. Java bytecode instructions can manipulate heap data by loading and storing elements from arrays, loading and storing values from objects' fields, or by accessing the length of an array.

2.2 Dynamic Taint Tracking for the JVM

Dynamic taint tracking systems have three primary responsibilities: associating taint tags with values, passing taint tags between procedure calls, and propagating taint tags in response to operations. Instrumentation-based taint tracking systems fulfill these responsibilities by transforming Java bytecode. There are two main approaches to instrumentation-based taint tracking in the JVM: "shadowing" [Bell and Kaiser 2014; Perkins et al. 2020] and "mirroring" [Ouyang et al. 2023]. As we describe in this section, neither approach on its own is sufficient to construct a precise, robust system for performing full-system dynamic taint tracking in modern JVMs. Section 3 describes our novel approach for combining aspects of each approach.

2.2.1 Shadowing. Shadowing was proposed by Bell and Kaiser [2014] and Perkins et al. [2020]. Shadowing stores taint tags alongside original values. The taint tag associated with a local variable is stored in a new "shadow local variable" in the same JVM stack frame as the original local variable. The operand stack is modified to store the taint tag of each operand, its corresponding "shadow operand", immediately next to the original operand in the same operand stack. Classes are instrumented to add a "shadow field" to store the tag associated with each original field. To store the taint tags associated with the lengths and elements of arrays, each array is replaced with a wrapper object which stores the original array and its taint tags.

Shadowing uses "shadow parameters" to pass taint tags between method calls. For each original method, a corresponding "shadow method" is created. The shadow method's signature contains the original method's parameters plus a corresponding shadow parameter for each original parameter to store the taint tag associated with the original parameter. An extra shadow parameter is added to the method's signature to store the return value's taint tag if the method has a return value. Each original method call is replaced with a call to the corresponding shadow method.

Shadowing adds logic to propagate taint tags directly into the body of methods. Operations on taint tags happen in the same method call as the original program logic using the same stack frame.

Because shadows are stored in the same fashion as original values and tag propagation occurs in the original method call, shadowing is able to match the runtime semantics of the JVM resulting in precise taint tag propagation. However, the invasiveness of the changes made by shadowing may cause divergences between an application's normal behavior and its behavior when performing the taint analysis including unexpected crashes [Ouyang et al. 2023]. We observed these divergences in our empirical evaluation of dynamic taint tracking systems for the JVM (discussed in Section 5).

2.2.2 Mirroring. Mirroring was proposed by Ouyang et al. [2023]. Mirroring stores the taint tags associated with values in "mirrored JVM stack frames" and a "mirrored heap". The taint tag

associated with a local variable is stored in the corresponding local variable slot in the mirrored frame for a method invocation. Similarly, the taint tags associated with the operand stack are stored in an operand stack within the mirrored frame. The mirrored heap uses a mapping to associate an object with a mapping from the names of its fields to taint tags. This mapping is also used to store the taint tags associated with arrays.

Mirroring uses a "mirrored JVM stack" to pass taint tags between method calls. Before a method call, mirroring creates a mirrored frame for that call. Next, the taint tags associated with the arguments passed to the method call are stored in the created frame. The frame is then pushed to a thread-local mirrored JVM stack. When control is transferred to the callee, this frame is accessed from the stack. The frame is also used to pass the taint tag of the return value of the method invocation, if any, to the caller method.

Mirroring does not add logic to propagate taint tags directly into method bodies. Instead, executed instructions are forwarded to the active mirrored frame which manipulates the mirrored heap and frame to model the effects of the operation.

Mirroring is less invasive than shadowing; it does not require changes to be made to classes' field, methods' signatures, or the class hierarchy. However, mirroring does not correctly model the runtime semantics of the JVM. As shown in our empirical evaluation (discussed in Section 5), incorrect modeling of JVM semantics can result in inaccurate taint tag propagation. Furthermore, mirroring may associate mirrored frames with the wrong method call. JVM implementations are allowed to make "up-calls" into the Java runtime between the execution of Java bytecode instructions [Lindholm et al. 2013]. In some cases, these calls are required by the specification, for example, to initialize a class, load a class, or create an implicitly thrown exception [Lindholm et al. 2013]. These calls can occur between the execution of an instruction inserted by the instrumentation to push a mirrored frame and the execution of the instruction inserted by the instrumentation to retrieve the pushed frame in the callee method. This can result in a mirrored frame being associated with the wrong method call causing incorrect taint tag propagation.

2.3 Signature Polymorphism

Signature polymorphism was introduced to support dynamic language features in the JVM [Lindholm et al. 2013; Oracle Corporation 2024c]. When linking and invoking signature polymorphic method calls, the JVM follows alternative semantics to allow for greater flexibility in how these methods can be used [Lindholm et al. 2013]. As of Java 21, there are two classes that can contain signature polymorphic methods: `java.lang.invoke.MethodHandle` (released in Java 7) and `java.lang.invoke.VarHandle` (released in Java 9) [Lindholm et al. 2013; Oracle Corporation 2023b]. Method handles provide a type-safe, performant way to dynamically access methods, constructors, and fields [Oracle Corporation 2023b]. Method handles are used to implement the bytecode instruction `invokedynamic` which was added in Java 7 and can be used to invoke a dynamically computed call site [Lindholm et al. 2013; Oracle Corporation 2023b, 2024c]. The `invokedynamic` instruction has been used to facilitate the implementation of dynamically typed languages for the JVM (Java 7), to support lambda expressions (Java 8), to improve string concatenation (Java 9), and to implement record types (Java 16). Variable handles provide type-safe, reflective access to variables using a variety of access-consistency policies [Oracle Corporation 2023b]. The alternative semantics used by the JVM to link and invoke signature polymorphic methods are what enables method and variable handles to support these use cases.

Before a Java bytecode instruction used to call an instance method (`invokevirtual`) can be executed, it must be linked by the JVM to resolve the symbolic method reference used by the instruction [Lindholm et al. 2013]. Normally, this resolution succeeds only if the JVM is able to find a method with the name and type descriptor, a string indicating a method's parameters and return

```
1 import java.lang.invoke.*;
2
3 public class Example {
4   public static void main(String[] args)
        throws Throwable {
5     MethodType mt = MethodType.methodType(int.
        class, int.class, int.class);
6     MethodHandle mh = MethodHandles.lookup().
        findStatic(Example.class, "add", mt);
7     int j = (int) mh.invokeExact(23, 4);
8   }
9
10  static int add(int x, int y) {return x + y;}
11 }
```

Listing 1. A program that uses a `MethodHandle`.

```
1 static int invokeExact_MT(Object o, int i, int j,
      Object mt) {
2   MethodHandle mh = (MethodHandle) o;
3   Invokers.checkExactType(mh, (MethodType) mt);
4   Invokers.checkCustomized(mh);
5   return mh.invokeBasic(i, j);
6 }
```

Listing 2. A generated adaptor method.

```
1 static int invokeStatic(Object mh, int i, int j) {
2   Object mn = DirectMethodHandle.internalMemberName(mh
      );
3   return MethodHandle.linkToStatic(i, j, (MemberName)
      mn);
4 }
```

Listing 3. `Holder#invokeStatic`.

type, specified by the symbolic reference in or inherited by the class specified by the symbolic reference [Lindholm et al. 2013]. However, calls to signature polymorphic methods can be resolved regardless of their type descriptor [Oracle Corporation 2023b]. This allows signature polymorphic methods to be called with a variety of signatures and return types.

When a signature polymorphic method is invoked, the symbolic type descriptor at the call site is compared against the type descriptor expected by the receiver of the method call (the method or variable handle) to ensure that the call is type safe [Lindholm et al. 2013; Oracle Corporation 2023b]. If this comparison succeeds, then the handle's underlying behavior or variable is directly invoked or accessed [Oracle Corporation 2023b]. The specifics of how this is achieved are left largely up to the implementation of the JVM[Oracle Corporation 2023b].

Consider an execution of the program in Listing 1 on an OpenJDK Corretto (version 11.0.22.7.1) runtime. The program begins by creating a method handle for the `add` method (lines 5–6). On line 7, a call to the signature polymorphic method `MethodHandle#invokeExact` appears. When the JVM goes to execute this instruction, it first makes an up-call into application code to the method `java.lang.invoke.MethodHandleNatives#findMethodHandleType` to obtain a `java.lang.invoke.MethodType` instance to represent the symbolic type descriptor for this call site. Next, the obtained `MethodType` instance is passed along with additional information about the call site to `MethodHandleNatives#linkMethod`. The method `linkMethod` computes a pointer to an adapter method and an appendix value for the call site; these results are stored into the constant pool cache [Rose 2013]. Subsequent executions of this call to `invokeExact` can use the stored method pointer and appendix directly without re-resolving them [Rose 2013]. In this example, the adapter method is dynamically generated by the JVM. Decompiled source code for this adapter method is shown in Listing 2. Once the adapter method pointer and appendix value are resolved, the adapter method is called. This call is passed the receiver of the original call to `invokeExact` (the `MethodHandle` instance), followed by the original arguments passed to `invokeExact`, and trailed by the resolved appendix value [Rose 2013].

The adapter checks whether the `MethodHandle`'s type matches the symbolic type descriptor for the call site. This check succeeds. Next, the adapter calls another signature polymorphic method, `MethodHandle#invokeBasic`. This call is handled specially by the JVM, which intrinsically links it to the method `Holder#invokeStatic` shown in Listing 3. The method `invokeStatic` obtains a `java.lang.invoke.MemberName` for the passed `MethodHandle` and calls another signature polymorphic method, `MethodHandle#linkToStatic`. This call is also handled specially by the JVM, which pops the trailing `MemberName` instance passed to `linkToStatic` and links the call to the target method indicated by the `MemberName`, in this case `Example#add`. Finally, `add` executes normally returning the value 27 which is passed up the call stack, to the call to `invokeExact`, and assigned to the local variable `j` on line 7 of `Example#main`.

As demonstrated in this example, the invocation semantics for signature polymorphic methods allow the JVM to "intrinsically" modify the arguments passed to signature polymorphic method calls. As a result, the arguments passed at a signature polymorphic method call site may not match the arguments received by the callee method. We experimented with a variety of signature polymorphic method calls on eight different JDKs (Java Development Kits): OpenJDK Temurin (versions 1.8.0_402-b06, 11.0.22+7, 17.0.10+7, and 21.0.2+13) and OpenJDK Corretto (versions 8.402.08.1, 11.0.22.7.1, 17.0.10.8.1, and 21.0.2.14.1). In these experiments, we observed the following intrinsic modifications: addition of trailing reference-typed values; removal of trailing reference-typed values; and conversion of `boolean`, `byte`, `short`, or `char` primitive values to type `int`.

Unfortunately, the flexibility that makes signature polymorphic methods useful also presents problems for dynamic taint tracking systems. Dynamic taint tracking systems need to reason about the flow of information between method calls. As described above, mirroring pass tags between method calls using mirrored frames and shadowing uses shadow parameters. Before executing a signature polymorphic method, the JVM can make calls into the Java runtime to resolve symbolic references used by the method invocation instruction. These calls can prevent mirroring from correctly associating the mirrored frame with the actual callee method. Shadowing is unaffected by any calls made by the JVM into the Java runtime because it explicitly passes taint tags from the caller to the callee as arguments. However, the modifications that shadowing makes to the type descriptors of method calls to add shadow parameters causes a mismatch between the type descriptor at the call site and the type descriptor expected by the method or variable handle.

2.4 The Java Platform Module System

The Java Platform Module System (JPMS) was introduced in Java 9 [Oracle Corporation 2024d]. The JPMS adds support for modules, groupings of related packages and resources. Each Java module contains a module declaration which specifies the name of the module, the other modules on which it depends (requires), the packages it exports to other modules (exports), the packages it makes available to other modules via reflection (opens), the services it provides (provides), and the services it consumes (uses) [Lindholm et al. 2013]. The dependencies and constraints defined in a module declaration are enforced by the JVM at runtime. Making use of the JPMS, the Java Class Library (JCL), the standard library included with the Java runtime, was decomposed into several modules starting in Java 9 [Oracle Corporation 2024d]. One of these modules, `java.base`, is the primordial module; every other module depends on `java.base` [Lindholm et al. 2013].

Instrumentation-based taint tracking systems modify classes' bytecode often inserting references to classes that are part of the taint tracking system (e.g., the class that represents a taint tag). If the instrumentation inserts a reference to a class that is not part of a module on which the module containing the class being instrumented declares a dependency, then the JVM will fail to resolve the inserted reference [Lindholm et al. 2013]. Fortunately, Java provides support for instrumenting classes at runtime as they are loaded through the use of a Java agent, and this support includes considerations for modules. If a class is instrumented at runtime using a Java agent, the JVM will transform the class' module to add a dependency on the unnamed module of the bootstrap class loader and the class loader that loads the main agent class [Oracle Corporation 2023b].

Unfortunately, not all classes can be instrumented as they are loaded using a Java agent; certain classes are loaded during the initialization of the JVM before a Java agent can be attached (e.g., `java.lang.String`). In theory, these classes can be instrumented using the method `java.lang.instrument.Instrumentation#retransformClasses`, which allows classes that have already been loaded to be instrumented [Oracle Corporation 2023b]. However, in some Java versions, there are restrictions on the types of changes that can be made when re-transforming a loaded class. For example, you cannot add or remove methods when re-transforming a class in Java 11 [Oracle

Corporation 2023a]. These restrictions can prohibit necessary bytecode transformations, particularly for shadowing. Therefore, classes that are loaded during the initialization of the JVM need to be instrumented statically before the Java runtime is started. However, this static instrumentation will introduce dependencies on classes that are part of the taint tracking system violating constraints imposed by the module system. Standard approaches for dynamically adding a dependency between modules cannot be used to correct this. Some classes are loaded before the Java command line option `--add-reads` is processed and before `java.lang.Module#addReads` can be called. If these core Java classes are not instrumented by a taint tracking system, it will result in taint tags being lost in critical classes like `java.lang.String`. Furthermore, approaches like shadowing assume the presence of instrumentation in classes and may crash in its absence.

3 Approach

Galette, our approach for dynamic taint tracking in modern JVMs, incorporates elements of both shadowing and mirroring to provide precise, robust taint tag propagation while minimizing deviations from an application's normal behavior. By strategically using elements of both approaches, we are able to preserve the precision of shadowing without the fragility introduced by certain bytecode modifications. This added robustness allows Galette to accommodate modern Java features. We also present techniques for reasoning about the flow of information through signature polymorphic methods and for statically instrumenting Java classes that are loaded before a Java agent can be attached without violating constraints imposed by the module system. These techniques enable Galette to track taint tags through the entirety of a Java runtime preventing taint tags from being lost due to missing propagation logic. In describing our approach, we highlight several key design decisions that have a significant impact on the overall compatibility and performance of the tool.

3.1 Applying Instrumentation

In order to associate taint tags with values and propagate those taint tags in response to operations, Galette instruments Java classes, modifying their bytecode. Because taint tags will not be propagated through code that is not instrumented, all classes being analyzed need to be instrumented to ensure correctness. Galette uses a Java agent to dynamically instrument classes as they are loaded. However, as noted in Section 2.4, certain JCL classes are loaded before a Java agent can be attached and need to be instrumented statically before the Java runtime is started. Therefore, Galette statically instruments the entirety of the JCL

By contrast, existing mirroring approaches do not statically instrument the JCL and, therefore, fail to propagate taint tags through certain JCL classes [Ouyang et al. 2023]. For instance, MirrorTaint does not instrument common classes like `java.lang.String` and `java.util.ArrayList`. Existing shadowing approaches require the JCL to be fully instrumented due to the invasiveness of the changes that they make to the bytecode [Bell and Kaiser 2014; Perkins et al. 2020]. However, static instrumentation applied by these approaches to the JCL violates constraints imposed by the JPMS, since it introduces an undeclared dependency between the module of the code being instrumented and the taint tracking runtime.

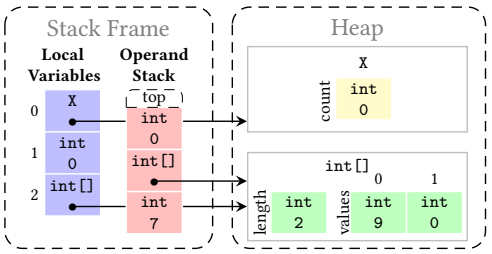
Galette introduces a novel approach to statically instrument the JCL, packing its classes and packages into the primordial module, `java.base`. Every module other than `java.base` has an implicit dependency on `java.base` [Lindholm et al. 2013]. Therefore, any class can reference a class in a package exported by `java.base` without violating the constraints imposed by Java's module system. In addition to packing its classes into `java.base`, Galette updates the module declaration for `java.base` to export the packages of Galette's classes so that they can be accessed from other modules.

```
1 public class X {
2     int count = 7;
3     int sum(int i, int
        [] a) {
4         return this.
        count + a[i];
5     }
6 }
```

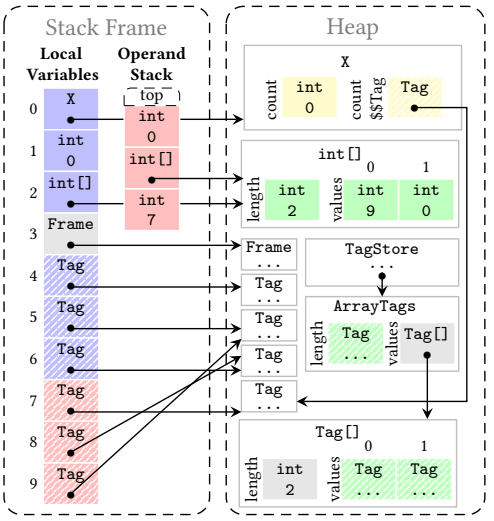
Listing 4. A Java class

```
1 aload_0
2 getfield X#count:I
3 aload_2
4 iload_1
5 iaload
6 iadd
7 ireturn
```

Listing 5. Bytecode for the sum method



(a) Original runtime memory.



(b) GALETTE-instrumented runtime memory.

Fig. 1. Figures 1a and 1b show the active JVM stack frame and heap before executing an array access, the `iaload` instruction in Listing 5, for an original runtime and a GALETTE-instrumented runtime, respectively. Locations containing original values found in the local variable array, the operand stack, fields, and arrays are colored solid blue, red, yellow, and green, respectively. Locations containing the taint tags associated with original values found in the local variable array, the operand stack, fields, and arrays are colored with hatched lines in blue, red, yellow, and green, respectively.

3.2 Associating Taint Tags with Values

Like traditional shadowing approaches, GALETTE stores taint tags alongside original values in order to ensure precise, performant taint tag propagation. However, unlike existing shadowing approaches, GALETTE preserves the locations of original values in runtime memory through its use of shadow operand stacks (Section 3.2.2) and a global array tag store (Section 3.2.4). This key design decision minimize deviations from an application's normal behavior allowing GALETTE to provide robust, full-system dynamic taint tracking.

3.2.1 Local Variables. Similar to shadowing, GALETTE stores the taint tag associated with each original local variable in a corresponding "shadow local variable" in the local variable array of the same JVM stack frame as the original local variable. For example, in the original runtime shown in Figure 1a, there are three local variables: a reference to an instance of type `X`, an `int` 0, and a reference to an array of `int` values. In the corresponding instrumented runtime (Figure 1b), local variables 4–6 are used to store the taint tags associated with the three original local variables.

3.2.2 The Operand Stack. GALETTE stores the taint tag associated with each original operand in the operand stack in a "shadow operand stack". The shadow operand stack is placed at the end of the local variable array in the same JVM stack frame as the original operand stack.

By contrast, existing shadowing approaches store the taint tags associated with elements of the operand stack directly on the operand stack [Bell and Kaiser 2014; Perkins et al. 2020]. This shifts the position of the original values on the operand stack and complicates the handling of opcodes that manipulate the order of elements on the stack. For example, the opcode `swap` is used to swap the top two elements on the operand stack. PHOSPHOR stores the taint tag of each

element of the operand stack immediately beneath it [Bell and Kaiser 2014]. Therefore, each original `swap` instruction must be modified to swap both the original values and their corresponding taint tags. While the `swap` opcode is relatively easy for Phosphor to implement, other opcodes such as `DUP2_X2` (which copies the top two words on the stack, placing them underneath the two words below them) quickly devolve into many special cases. The generated code can be so verbose as to result in instrumented methods larger than Java's maximum method size, a fundamental limitation of Phosphor [Phosphor Contributors 2018].

Galette's shadow operand stack is contained within the local variable array. This ensures that the positions of original values on the operand stack remain unchanged, allowing original instructions to be performed normally.

The top of Galette's shadow operand stack is always located in the last local variable index. For example, in the original runtime shown in Figure 1a, the operand stack has three operands: an `int` 0 (on the top), a reference to an array of `int` values, and another `int` 7. In the corresponding instrumented runtime (Figure 1b), local variables 7–9 are used to store the taint tags associated with the original operands. The taint tag of the top operand, the `int` 0, is stored in index 9, the last local variable.

3.2.3 Fields. Similar to shadowing, Galette stores the taint tag associated with each field by adding a corresponding "shadow field"' to the declaring class. This shadow field uses the same name as the original field with a consistent, known suffix (e.g., `$$Tag`) appended to its end. For example, in the original runtime shown in Figure 1a, the class `X` declares a single field, `count`. The first local variable contains a reference to an instance of type `X`; this instance has a `count` value of 0. In the corresponding instrumented runtime (Figure 1b), the class `X` declares two fields: `count` and its shadow, `count$$Tag`. The first local variable still contains a reference to an instance of type `X`. However, this instance now has a `count` value of 0 and a `count$$Tag` value which points to a `Tag` object in the heap.

Shadow fields are declared with the same modifiers as the original field, on the same class, and with a consistent, known suffix. This enables a shadow field to be accessed in the same fashion as the corresponding original field causing the reference to the shadow field to be resolved by the JVM in the same fashion as the original field. This removes the need to model the JVM's field resolution semantics, which, as noted in Section 2.2.2, is a limitation of mirroring fields.

3.2.4 Arrays. Unlike shadowing, Galette does not create shadow wrappers for arrays. Instead, Galette maintains a global array tag store similar to the mirrored heap described in Section 2.2.2. This tag store maps each array to a record containing the taint tags associated with the array's length and elements. For example, in the original runtime shown in Figure 1a, the local variable in index 2 refers to an array of `int` values. In the corresponding instrumented runtime (Figure 1b), the `ArrayTags` instance mapped to this array can be retrieved from a global `TagStore`. This `ArrayTags` instance contains a `Tag` representing the taint tag associated with the array's length and an array containing the taint tags associated with the array's elements.

Unlike field accesses, the semantics of array accesses do not depend on the type hierarchy. Thus, a global tag store can be used to precisely store the taint tags associated with an array's length and elements. The use of a global tag store over array wrappers removes a source of brittleness that affects shadowing. In particular, if an array wrapper is passed into code that is not instrumented (e.g., native methods), it may cause the program to behave unexpectedly or even crash.

Even though Galette's array tag store is similar to the mirrored heap used in mirroring, operations on arrays differ significantly for Galette compared to mirrored approaches due to differences in how the taint tags of other values are stored. For example, Galette collects the taint tags associated with array operations within the instrumented method and the forwards these tags

```
1 class X {
2   static int i = 7;
3   static void set(int b) {
4     X.i = b;
5   }
6
7   static int run(int[] arr
        , int y) {
8     int a = arr[y];
9     set(a);
10    return a;
11  }
12 }
```

Listing 6. Original Java class.

```
1 class X {
2   static int i = 7;
3   static Tag i$$TAG;
4   static void set(int b, Frame
        frame) {
5     Tag b$$TAG = frame.get(0);
6     X.i$$TAG = b$$TAG;
7     X.i = b;
8   }
9
10  static int run(int[] arr, int y
        , Frame frame) {
11    Tag y$$TAG = frame.get(0);
12    Tag arr$$TAG = frame.get(1);
13    Tag a$$TAG = TagStore.getTag(
        arr, y, arr$$TAG, y$$TAG);
14    int a = arr[y];
15    Frame frame2 = new Frame(
        a$$Tag);
16    set(a, frame2);
17    frame.setReturnTag(a$$TAG);
18    return a;
19  }
20 }
```

Listing 7. Instrumentation applied by Galette to the class in Listing 6.

to the array tag store in order to manipulate the record associated with the array. This is shown on line 13 of Listing 7. Whereas, in mirroring the collection and manipulation of all taint tags occurs in separate method calls that operate entirely on the mirrored JVM stack frame and mirrored heap.

3.3 Propagating Taint Tags

Galette modifies the bodies of methods to insert logic for propagating taint tags in response to operations. This propagation happens in the same method call as the original program logic. Listing 7 shows the instrumentation applied by Galette to the class in Listing 6.

When an original instruction modifies the operand stack, Galette performs the same modification to the shadow operand stack. When an original instruction accesses a local variable, Galette accesses the corresponding shadow local variable. For example, on line 14 of Listing 7, the local variable `a` is declared and defined. On line 13, Galette's instrumentation declares and defines a shadow local variable `a$$TAG` to store the taint tag associated with `a`. Since the definition of `a` uses the value of the variables `arr` and `y`, the definition of `a$$TAG` directly uses the taint tags associated with `arr` and `y` which are stored in the shadow variables `arr$$TAG` and `y$$TAG`, respectively.

Since it uses a novel combination of shadowing and mirroring, Galette must employ multiple strategies to propagate taint tags. When an original instruction accesses a field, Galette accesses the corresponding shadow field. For example, since a value is stored to the field `i` on line 7 of Listing 7, Galette stores a value to the corresponding shadow field `i$$TAG` on line 6. When an original instruction accesses the length or an element of an array, Galette accesses the taint tags associated with the accessed value from a global array tag store. For instance, on line 14 of Listing 7, the element at index `y` of the array `arr` is accessed. On line 13, Galette retrieves the taint tag associated with that element from the tag store by calling `TagStore#getTag`.

3.4 Passing Taint Tags Between Methods

Like mirroring, Galette uses a "tag frame" to pass taint tags between method calls. This tag frame stores the taint tag associated with each argument passed to a method call and, after the method call finishes, the taint tag of the return value of the call. Unlike mirroring, Galette does not use a mirrored JVM stack to pass this tag frame from caller to callee. Instead, Galette directly passes a tag frame as an argument to method calls by modifying method descriptors and call sites. This ensures that tag frames are associated with the correct method call even in the presence of up-calls from the JVM into the Java runtime.

For each original method, Galette creates a corresponding "shadow method". The shadow method's signature contains the original method's parameters plus a trailing tag frame. Before a method call, Galette creates a tag frame containing the taint tags of the arguments to be passed to the call (including the receiver of the method call). This tag frame is passed as the last argument to

the call. The original method call is then replaced with a call to the corresponding shadow method. At the beginning of the callee method, the tag frame passed from the caller is used to initialize the taint tags associated with received arguments. At the end of the callee method, Galette sets the taint tag associated with the return value of the method invocation on the tag frame passed from the caller. After the method call returns, in the caller method, Galette retrieves the taint tag associated with the method invocation's return value from the passed tag frame.

For example, on line 15 of Listing 7, Galette creates a new tag frame for a call to the method `set`. This frame is then passed to the call to set on line 16. On line 5 of the shadow method for `set`, the taint tag for the parameter `b` is retrieved from the tag frame and stored to the shadow local variable for `b`.

3.4.1 Wrapper Methods. The original methods defined in a class cannot safely be removed in favor of the corresponding shadow methods, because it is possible that these methods may still be called by native code or an up-call from the JVM. Galette keeps all original method definitions in a class, but replaces the bodies of these methods with wrapper code that obtains a tag frame and then calls the corresponding shadow method for that original method. Existing shadowing approaches also contain wrappers for JVM-managed methods to handle method calls from unmanaged code. Unlike Galette, these wrappers simply call the corresponding shadow method with empty taint tags. By contrast, Galette's wrappers contain logic for receiving tag frames stored at signature polymorphic method call sites. We describe this in more detail in Section 2.3. Mirroring does not modify methods' signatures and, therefore, does not need method wrappers. However, as a result, mirroring cannot distinguish between method calls made from JVM-managed code and unmanaged code. As described in Section 2.2.2, this can result in incorrect taint tag propagation between method calls.

3.5 Signature Polymorphic Methods

As described in Section 2.3, signature polymorphic methods do not follow the same invocation semantics as normal methods. As a consequence of these alternative semantics, the type descriptor of signature polymorphic method call cannot be modified to explicitly pass a tag frame as an argument to the call. Such a modification would produce a mismatch between the type descriptor at the call site and the type descriptor expected by the method or variable handle receiver of the call. Therefore, Galette indirectly passes the taint tags of arguments to the targets of signature polymorphic method calls using a thread-local "frame store". The frame store for a thread of execution may be empty or hold a single frame containing the taint tags associated with the arguments of the next target of a signature polymorphic method call to be executed on that thread.

Galette's frame store is impacted by similar limitations as the mirrored JVM stack used in mirroring. In particular, up-calls made by the JVM between when a tag frame is stored and the callee method is executed may cause an indirectly passed frame to be associated with the wrong method call. To minimize the likelihood of these misassociations, Galette records additional information about the method call in the frame store. This information is used to distinguish between the actual target of the signature polymorphic method call and up-calls made by the JVM into the runtime.

Galette instruments signature polymorphic method call sites to insert code to prepare the frame store for the call and restore the state of the frame store after the call. First, this instrumentation creates a tag frame containing the taint tags associated with each argument passed to the method call (including the receiver of the method call). This is identical to how a tag frame is created for a non-signature polymorphic method call. Next, this frame and the actual arguments to be passed to the method call are stored into the frame store for the current thread of execution. Then, the signature polymorphic method is called normally — without a trailing tag frame. Finally, when this

call completes (normally or exceptionally), the frame store for the current thread of execution is cleared.

To receive the frame stored at signature polymorphic method call site, the target callee method must check the frame store. Thus, Galette applies instrumentation to any potential target of a signature polymorphic method call to check the frame store. Because Galette does not modify method or variable handles, the underlying method referenced by a method or variable handle will always be a method declared in the original, un-instrumented class — not a shadow method added by Galette. As described in Section 3.4.1, original methods are converted by Galette into wrappers which obtain a tag frame and then call the corresponding shadow method. Since any wrapper can be a potential target of a signature polymorphic method call, when a wrapper attempts to obtain a tag frame, it first checks the frame store for the current thread of execution. If the store is empty, then the wrapper will call its corresponding shadow method with an empty frame, a frame indicating that none of the arguments are tainted. If the store is not empty, the wrapper will retrieve the stored entry and clear the store. Next, the wrapper checks whether the arguments received by the wrapper match the arguments passed at the signature polymorphic call site, the ones stored in the retrieved entry. This matching procedure is relaxed and compensates for the intrinsic modifications discussed in Section 2.3. If the wrapper's arguments match the stored arguments, then the wrapper will call its corresponding shadow method with the stored tag frame. Otherwise, the wrapper will call its corresponding shadow method with an empty frame. Before returning, the wrapper restores the state of the frame store to what it was at the start of the execution of the wrapper.

For example, consider the call to the signature polymorphic method `linkToStatic` shown in Listing 3. Before this call, Galette will create a tag frame containing the taint tags associated with the arguments `i`, `j`, and `mn`. Next it will store this frame and the argument values `23`, `4`, and a `MemberName` instance to the frame store for the current thread of execution. Then call to `linkToStatic` is performed. The JVM pops the trailing `MemberName` instance passed to `linkToStatic` and links the call to the wrapper method for `Example#add`. This wrapper retrieves the stored entry for the current thread of execution from the store and clears the store. Next, the wrapper checks whether the arguments it was passed, `x` and `y`, match the arguments stored by the caller. The value of `x` will be equal to the stored value `23`, and the value of `y` will be equal to the stored value `4`. The relaxed matching algorithm will discard the trailing `MemberName` which was removed by the JVM allowing the match to succeed. As a result, the wrapper will call its corresponding shadow method with the stored frame allowing the taint tags stored for the arguments to propagate to the callee.

3.6 Native Methods

The Java Native Interface (JNI) allows Java bytecode programs to interoperate with programs written in other languages through the use of native methods [Oracle Corporation 2024a]. Galette does not track the flow of information through these native methods. For each native method, Galette creates a special shadow method which discards the passed tag frame and calls the original native method with the remaining arguments.

Galette's native method wrappers are simpler and less fragile than what is required for traditional shadowing approaches, like Phosphor [Bell and Kaiser 2014] and ClearTrack [Perkins et al. 2020]. Because shadowing uses wrapper objects to store array tags and stores taint tags directly onto the operand stacks of method calls, the placement of original values in memory is modified. These modifications need to be corrected before a native method can be safely called. In traditional shadowing, native method wrappers perform these corrections. However, these corrections can be expensive particularly for large arrays. Furthermore, modifications made for multidimensional arrays and array fields cannot be fully corrected while ensuring program semantics. Because

Galette preserves the locations of original values in runtime memory, Galette's native wrappers only need to load the native method's arguments onto the runtime stack, call the native method, and, if the native method is non-void, return the value returned by the called method.

3.7 Reflection and Unsafe

Java's Reflection API allows developers to inspect the attributes of classes at runtime, access fields, manipulate arrays, and invoke methods. Java's Unsafe API allows developers to perform low-level, unsafe operations including accessing array elements and fields. Galette inserts special instrumentation to ensure that taint tags are propagated through reflective and unsafe operations. Additionally, Galette "masks" calls to reflective methods that expose changes made by Galette to an application which could cause deviations from the program's original behavior. For example, to hide shadow fields from reflection, Galette masks calls to reflective methods that expose the fields of a class (e.g., `java.lang.Class#getFields`). After a method that exposes the fields of a class is called, Galette intercepts the return value to remove added shadow fields.

3.8 Limitations

As mentioned in Section 3.6, Galette does not track the flow of information through native methods. More broadly, Galette does not track the flow of information outside the Java runtime. This includes through native code, databases, the network, and the file system.

Galette approach to tracking the flow of information through signature polymorphic method calls is not guaranteed to be correct. It is still possible that an up-call into the JVM is incorrectly matched as the target of a signature polymorphic method call. It is also possible that the relaxed matching procedure is not sufficiently permissive to ensure that the arguments received by the true target method call are always considered to be a match to the arguments passed at the call site. However, because Galette replaces all non-signature polymorphic method calls with calls to the corresponding shadow method, these mismatches can only occur for signature polymorphic method calls and up-calls made by the JVM. Furthermore, mismatches are limited to up-calls with almost identical arguments to the true target method call.

Galette currently only supports tracing the flow of information through "explicit" or "data" flows [Sabelfeld and Myers 2006]. Galette does not support tracing the flow of information through "implicit" or "control" flows [McCamant and Ernst 2008]. However, Galette could be extended to support different propagation logic.

Like existing dynamic taint tracking systems for the JVM, Galette does not ensure accurate taint tag propagation in the presence of data races. A data race on a value will produce a second data race on the taint tag associated with the value. Without the use of some form of thread synchronization on all operations that access the heap, a dynamic taint tracking system cannot ensure that these two races will have consistent outcomes.

4 Implementation

Our implementation of Galette uses the ASM [OW2 Consortium 2024] Java bytecode manipulation and analysis framework to instrument Java classes. As mentioned in Section 3.1, we use a Java agent to dynamically instrument most classes as they are loaded, but we statically instrument the JCL. For Java versions 8 or less, the JCL is statically instrumented by processing the class files contained in a Java installation. For Java versions 9+, we statically instrument the JCL using jlink, a tool included in the JDK starting in Java 9 to build custom runtime images [Oracle Corporation 2024b]. Before running a Java 9+ application with Galette, jlink is invoked with two custom plugins to create an instrumented Java runtime. The first plugin applies Galette's instrumentation

to Java classes. The second plugin is responsible for packing Galette's classes into `java.base` and updating the module declaration for `java.base` as described in Section 3.1.

To implement the global array tag store described in Section 3.2.4, we used `java.lang.ref.WeakReference` instances to store the array keys. Unlike standard Java references, weak reference objects do not prevent their referents from being reclaimed by JVM's garbage collector. This ensures that when an array is no longer reachable through standard references it can still be reclaimed by JVM's garbage collector even if there is an entry for it the tag store.

To create the thread-local frame stores described in Section 3.5, we instrumented `java.lang.Thread`, the class that the Java platform uses to represent a thread of execution, to add an instance field that holds a frame store. The method `Thread#currentThread` can be used to obtain the `Thread` instance representing the current thread of execution [Oracle Corporation 2023b]. Then, the current thread's frame store can be accessed from the added field on the obtained `Thread`.

As discussed in Section 3.5, Galette uses a relaxed matching procedure when indirectly passing tag frames from signature polymorphic method call sites to callee methods. Our implementation of Galette uses a matching procedure that compensates for the intrinsic modifications described in Section 2.3, but could easily be modified to accommodate different intrinsic modifications.

5 Evaluation

Our evaluation of Galette focused on the following research questions:

- **RQ1**: How well does Galette preserve program semantics?
- **RQ2**: How accurately does Galette track taint tags?
- **RQ3**: What is the runtime overhead of Galette?
- **RQ4**: What is the memory overhead of Galette?

5.1 Methodology

5.1.1 Baselines. To evaluate these questions, we compared Galette against two state-of-the-art dynamic taint tracking systems for the JVM: Phosphor [Bell and Kaiser 2014] and MirrorTaint [Ouyang et al. 2023]. We used the latest release of Phosphor (version 0.1.0) [Bell and Phosphor Contributors 2024] and the latest commit of MirrorTaint (commit 54fcbfc) [MirrorTaint Contributors 2023].

Phosphor uses shadowing and requires the JCL to be instrumented [Bell and Kaiser 2014]. For the reasons discussed in Section 2.4, this instrumentation must be applied statically to certain parts of the JCL, and this static instrumentation violates constraints imposed by the JPMS. To evaluate Phosphor on Java 9+ JDKs, we worked with the maintainers of Phosphor to create custom jlink plugins, similar to the ones we created for Galette, for Phosphor that implement the approach described in Section 3.1. For all experiments, Phosphor was configured with the options "forceUnboxAcmpEq", "withEnumsByValue", and "serialization". These options were selected based on the advice of the maintainers of Phosphor.

MirrorTaint uses mirroring and does not support statically instrumenting the JCL. Instead, MirrorTaint relies upon manually defined propagation rules for commonly used methods in the JCL [Ouyang et al. 2023]. This approach is unsound — taint tags will not be propagated through methods that are not instrumented for which a propagation rule has not been defined. However, it allows MirrorTaint to run on Java 9+ without violating constraints imposed by the JPMS. We encountered some defects in the latest commit of MirrorTaint which prevented us from using the unmodified release in our evaluation. We made minor changes to MirrorTaint to correct these defects. This corrected version of MirrorTaint is what we used in our evaluation.

5.1.2 Benchmarks. Our evaluation featured two benchmark suites: a functional and a performance benchmark suite. We used the functional benchmark suite for RQ1 and R2 which evaluate Galette's conformance with the functional requirements of a taint tracking system. We used the performance benchmark suite for RQ3 and RQ4 which evaluate Galette's runtime performance.

The functional benchmark suite consists of a set of deterministic, synthetic programs designed to exercise both newer and older Java features. To identify deviations from expected program behavior and measure the accuracy of taint tag propagation, we needed a ground truth for both the expected behavior of the program and the expected propagated taint tags for each benchmark program. As noted by Pauck et al. [2018], the ground truth for expected taint tag propagation needs to be manually determined. However, it is challenging to manually determine the ground truths for arbitrary, real-world programs. Therefore, we manually constructed synthetic programs for which we could manually determine the ground truth expected taint tag propagation. We grouped these programs by the type of functionality that they exercise. A detailed listing of these benchmark groups is available in the supplemental materials for this paper (described in Section 8). Each synthetic program contains handwritten assertions which check that the program has executed normally; these assertions are intended to always pass on a Java runtime that has not been instrumented. For each program, we manually apply taint tags to selected input values and determine the set of expected taint tags for selected values computed in the program.

For the performance benchmark suite, we used the third major release of the DaCapo benchmark suite (version dacapo-23.11-chopin) [DaCapo Project Contributors 2023b]. The DaCapo benchmark suite is a well established suite of JVM performance benchmarks first proposed by Blackburn et al. [2006]. The DaCapo benchmark suite (version dacapo-23.11-chopin) consists of non-trivial workloads for 22 real-world, open-source applications [Blackburn et al. 2006; DaCapo Project Contributors 2023b]. We excluded two of these benchmarks, "cassandra" and "kafka", due to failures that we observed when executing these benchmarks on a Java runtime that had not been instrumented. Of the remaining twenty benchmarks, eight required Java version 11 or greater; the rest are compatible with Java version 8 or greater [DaCapo Project Contributors 2023a]. We used the "small" workload size for each benchmark. The timeouts used in the benchmarks were increased by a factor of 1000 to give adequate time for the slower taint tracking systems to complete tasks. We also modified two of the benchmarks, "tradebeans" and "tradesoap", to remove problematic timeouts. We did not apply taint tags to any values for these benchmarks. Generally, overheads for taint tracking systems will be higher in the presence of tainted data.

5.1.3 Metrics. For RQ1 and RQ2, we ran each functional benchmark program with each taint tracking system on JDKs for the four active LTS versions of Java (8, 11, 17, and 21). If a program uses features that are not available on a specific JDK, then that program was skipped on that JDK. For each program execution, we recorded the outcome as either passing, failing due to deviations from the original program semantics, or failing due to incorrect taint tag propagation. If an unexpected crash, exception, timeout, or assertion failure occurred during the execution of the program, then that execution was marked as failing due to deviations from the original program semantics. Otherwise, if the set of taint tags reported by the taint tracking system did not match our manually determined set of expected taint tags, then the execution was marked as failing due to incorrect taint tag propagation. All other executions were marked as passing.

For RQ3 and R4, we collected measurements on each DaCapo benchmark for four treatments: without tainting, with Galette, with Phosphor, and with MirrorTaint. For each treatment on each DaCapo benchmark, we performed five warm-up iterations, which are discarded, followed by five measurement iterations on each of twenty JVM processes for a total of 100 measurements. Warm-up iterations mitigate major performance fluctuations that occur when the JVM is first started

due to dynamic optimizations like Just-in-Time (JIT) compilation [Barrett et al. 2017; Georges et al. 2007; Traini et al. 2023]. For each measurement iteration, we record the execution time and the peak memory usage. The execution time is the elapsed real (wall-clock) time taken to complete the workload. Peak memory usage is computed by sampling the resident set size (RSS) of the Java process with a sample interval of one millisecond. The maximum RSS sampled during the iteration is recorded as the peak memory usage. We observed that on certain benchmarks the instrumentation used by some taint tracking systems resulted in infinite loops or deadlock. Therefore, we imposed an overall timeout of 24 hours for each DaCapo process. Each DaCapo benchmark performs a checksum-based "validity check" that confirms that the benchmark execution is correct [Blackburn et al. 2006]. If any validity check failed, an unhandled exception was thrown, the DaCapo process crashed, or the 24-hour timeout elapsed, we discarded all measurements for the process.

5.1.4 Experimental Setup. All experiments were conducted on a virtual machine with four 2.6 GHz AMD EPYC 7H12 vCPUs, with 16 GB of RAM, running Ubuntu 20.04.3. For RQ1 and RQ2, we used OpenJDK Temurin versions 1.8.0_402-b06, 11.0.22+7, 17.0.10+7, and 21.0.2+13. For RQ3 and RQ4, we used OpenJDK Temurin version 11.0.22+7. For all experiments, we used the Java options `-Xmx6G` and `-ea` in addition to any options needed to use a particular taint tracking system.

5.2 Results

5.2.1 Functional. Table 1 reports the number of programs failing due to semantic and propagation failures in each benchmark group for each taint tracking system using each JDK. A total of 11,364 tests were attempted for each taint tracking system across 26 groups and four JDK versions.

For both Galette and MirrorTaint, we observed no semantic failures. For Phosphor, there were 7,214 semantic failures including all 6,902 tests for JDK versions 17 and 21. Even when using the custom jlink plugins described in Section 5.1.1, the Phosphor-instrumented JVMs for Java 17 and 21 crashed during initialization. Without these plugins, the Phosphor-instrumented JVM for Java 11 also crashed during initialization. The remaining semantic failures for Phosphor were in a total of five groups each of tested either reflection or signature polymorphic methods. In several of these cases, Phosphor's use of array wrappers to store taint tags caused the JVM to crash.

There were no observed propagation failures for Galette. There were 6,584 propagation failures for MirrorTaint from twenty unique groups of the 26 total functional groups. For Phosphor, we observed 888 propagation failures from eleven unique groups. Both systems performed particularly poorly on groups that tested signature polymorphic methods: "Lambda", "Method Handle", "Method Handle 9+", "Record Type", "String Indy Concat" and "Var Handle". There were 4,321 and 826 propagation failures for MirrorTaint and Phosphor in these groups, respectively. Most of the propagation failures for MirrorTaint were due to incorrect or missing modeling of JVM semantics which as discussed in Section 2.2.2 is a limitation of mirroring. The propagation failures for Phosphor were generally due to an inability to track the flow of information through signature polymorphic method calls and missing special handling for parts of the Reflection and Unsafe APIs.

Overall, we found that Phosphor was prone to introducing deviations into original programs' semantics particularly in the presence of modern Java features like signature polymorphism. On JDK versions 17 and 21, these deviations prevented the JVM from successfully initializing. By contrast, there were no semantic failures for MirrorTaint. However, MirrorTaint failed to accurately propagate taint tags in tests from twenty of the 26 total functional groups. Galette was able to accurately propagate taint tags without introducing deviations into programs' semantics.

5.2.2 Performance. Table 2 reports the execution time and peak memory usage for each taint tracking system and the baseline (without taint tracking) on each performance benchmark. For the baseline, we report the median value. For each taint tracking system, we report the overhead

Table 1. Semantics Preservation and Propagation Accuracy. For each taint tracking system, we report the number of tests in each group (Group) that failed due to deviations from the original program semantics (Semantic) and incorrect taint tag propagation (Tag) on JDK versions 8, 11, 17, and 21. Next to each group, we list the total number of test cases in that group (#). Non-zero entries are colored red. Values are omitted (—) for a group on a JDK version if that group uses features that are unavailable on that JDK version.

		Galette								MirrorTaint								Phosphor							
		Semantic				Tag				Semantic				Tag				Semantic				Tag			
Group	#	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21	8	11	17	21
Array Access	74	0	0	0	0	0	0	0	0	0	0	0	0	55	55	55	55	0	0	74	74	0	0	0	0
Array Length	5	0	0	0	0	0	0	0	0	0	0	0	0	3	3	3	3	0	0	5	5	0	0	0	0
Array Reflection	75	0	0	0	0	0	0	0	0	0	0	0	0	65	65	65	65	0	0	75	75	7	7	0	0
Assignment	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	12	0	0	0	0
Boxed Type	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	8	0	0	0	0
Class Reflection	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	7	0	0	0	0
Collection	6	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	0	0	6	6	0	0	0	0
Conditional	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	0	0	0	0
Constructor Reflection	88	0	0	0	0	0	0	0	0	0	0	0	0	12	12	12	12	18	18	88	88	0	0	0	0
Field	6	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	0	0	6	6	0	0	0	0
Field Reflection	176	0	0	0	0	0	0	0	0	0	0	0	0	31	31	31	31	0	0	176	176	0	0	0	0
Jdk Unsafe	322	—	0	0	0	—	0	0	0	—	0	0	0	—	213	213	213	—	0	322	322	—	27	0	0
Lambda	7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	7	7	1	1	0	0
Loop	4	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	4	4	0	0	0	0
Method Call	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	14	0	0	0	0
Method Handle	40	0	0	0	0	0	0	0	0	0	0	0	0	34	34	34	34	3	3	40	40	36	36	0	0
Method Handle 9+	13	—	0	0	0	—	0	0	0	—	0	0	0	—	11	11	11	—	4	13	13	—	7	0	0
Method Reflection	213	0	0	0	0	0	0	0	0	0	0	0	0	25	25	25	25	38	38	213	213	1	1	0	0
Record Type	68	—	—	0	0	—	—	0	0	—	—	0	0	—	—	16	16	—	—	68	68	—	—	0	0
Static Initializer	2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	2	2	0	0	0	0
String	26	0	0	0	0	0	0	0	0	0	0	0	0	13	13	13	13	0	0	26	26	2	1	0	0
String Builder Concat	37	0	0	0	0	0	0	0	0	0	0	0	0	19	19	19	19	0	0	37	37	6	6	0	0
String Indy Concat	37	—	0	0	0	—	0	0	0	—	0	0	0	—	19	19	19	—	0	37	37	—	19	0	0
Sun Unsafe	270	0	0	0	0	0	0	0	0	0	0	0	0	177	177	177	177	0	0	270	270	0	0	0	0
Throwable	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	2	2	0	0
Var Handle	1,932	—	0	0	0	—	0	0	0	—	0	0	0	—	1,353	1,353	1,353	—	190	1,932	1,932	—	726	0	0

compared to the baseline. Overhead was computed as $\frac{\tilde{x_t}-\tilde{x}}{\tilde{x}}$, where $\tilde{x_t}$ was the median value for the taint tracking system and $\tilde{x}$ was the median value for the baseline.

The missing entries in Table 2 indicate cases where we were unable to collect valid measurements for a tool on a benchmark due to failures. In all such cases, we were able to consistently reproduce these failures. There were nine benchmarks for which we could not collect results for Phosphor. On seven of these benchmarks, Phosphor's instrumentation induced a deviation from the benchmark's original semantics that produced an exception or error. On the remaining two benchmarks, "tradebeans" and "tradesoap", Phosphor's instrumentation induced a deviation from the benchmark's original semantics that produced an infinite loop. There were eleven benchmarks for which we could not collect results for MirrorTaint. On ten of these benchmarks, MirrorTaint's instrumentation induced a deviation from the benchmark's original semantics that produced an exception or error. On the remaining benchmark, "graphchi", the JVM ran out of heap space when run with MirrorTaint. As mentioned in Section 5.1.4, we used the Java option `-Xmx6G` to specify a maximum heap size of six gigabytes. Galette was able to run all twenty of the benchmarks successfully.

We performed Mann-Whitney U tests to compare the overhead of Galette with respect to execution time and memory usage against the other taint tracking systems on each benchmark. A continuity correction of 0.5 was used when calculating two-tailed, asymptotic p-values. Following current best practices as described by Arcuri and Briand [2014], a base significance level of 0.05 was adjusted for three comparisons resulting in a Bonferroni-corrected significance level of 0.0167 per test. We used the Vargha-Delaney $\hat{A}_{12}$ statistic [Vargha and Delaney 2000] to quantify effect

Table 2. Execution Time and Peak Memory Usage. For the baseline (Base), we report the median ($\tilde{x}$) execution time in milliseconds (left) and peak memory usage in megabytes (right) on each benchmark. For each taint tracking system, we report the overhead (OV) for execution time and peak memory usage relative to the baseline. For each reported overhead, we also report the lower confidence limit (LCL) and upper confidence limit (UCL) of a two-tailed, bias-corrected 95% bootstrap confidence interval. We used 1,000 resamples to compute each confidence interval. For MirrorTaint and Phosphor, values that are statistically significantly greater than or less than Galette's are colored green and red, respectively.

	Execution Time										Peak Memory Usage									
	Base	Galette			MirrorTaint			Phosphor			Base	Galette			MirrorTaint			Phosphor		
	$\tilde{x}$	OV	LCL	UCL	OV	LCL	UCL	OV	LCL	UCL	$\tilde{x}$	OV	LCL	UCL	OV	LCL	UCL	OV	LCL	UCL
avrora	2,360	0.90	0.84	0.98	662.53	628.58	689.92	1.59	1.47	1.76	126.89	0.71	0.44	0.83	16.72	15.83	17.98	1.56	1.45	1.74
batik	272	11.30	10.33	12.21	2,908.87	2,715.91	3,101.19	—	—	—	226.22	1.48	1.40	1.56	9.25	8.96	9.54	—	—	—
biojava	134	29.28	27.82	30.31	2,642.57	2,421.95	2,781.04	15.32	14.69	16.22	171.58	2.34	2.27	2.42	4.01	3.94	4.06	1.30	1.26	1.33
eclipse	628	1.68	-0.49	3.33	—	—	—	—	—	—	295.46	0.99	0.83	1.25	—	—	—	—	—	—
fop	110	3.01	2.57	3.50	—	—	—	5.64	4.84	6.22	144.34	1.10	1.08	1.13	—	—	—	2.25	2.21	2.28
graphchi	504	5.40	5.12	5.77	—	—	—	13.81	13.17	14.51	404.80	1.27	1.12	1.36	—	—	—	0.36	0.34	0.38
h2	134	4.40	4.06	4.71	866.75	834.94	898.80	7.69	7.22	8.31	330.46	1.15	1.11	1.19	1.26	1.23	1.30	1.49	1.46	1.52
h2o	600	16.17	15.59	17.38	—	—	—	—	—	—	393.08	0.75	0.72	0.80	—	—	—	—	—	—
jme	1,022	3.16	2.86	3.47	3,916.11	3,730.04	4,183.92	3.65	3.41	3.96	265.96	1.64	1.52	1.75	8.34	7.89	8.74	1.27	1.16	1.38
jython	381	10.37	9.81	10.93	—	—	—	—	—	—	420.20	6.50	6.28	6.76	—	—	—	—	—	—
luindex	1,100	5.32	2.54	7.18	—	—	—	10.05	5.19	13.37	226.56	0.86	0.81	0.90	—	—	—	1.54	1.49	1.59
lusearch	182	9.59	8.61	10.25	3,630.84	3,320.09	3,825.41	8.50	7.62	9.12	216.05	2.29	2.10	2.45	2.72	2.61	2.85	0.93	0.89	1.01
pmd	98	2.88	2.56	3.29	—	—	—	—	—	—	137.40	0.88	0.85	0.92	—	—	—	—	—	—
spring	98	1.68	1.52	1.86	—	—	—	4.62	4.23	5.04	408.66	1.01	0.99	1.03	—	—	—	1.67	1.65	1.70
sunflow	285	14.18	13.52	15.05	27,399.37	26,446.86	27,774.67	16.94	16.33	17.42	166.03	1.18	1.12	1.24	7.21	7.03	7.44	1.15	1.12	1.20
tomcat	352	4.03	3.58	4.38	63.63	60.30	68.10	—	—	—	280.74	1.49	1.45	1.54	3.88	3.54	4.13	—	—	—
tradebeans	162	6.90	6.48	7.49	—	—	—	—	—	—	607.82	1.40	1.36	1.46	—	—	—	—	—	—
tradesoap	453	8.74	7.98	9.42	—	—	—	—	—	—	681.13	1.30	1.25	1.32	—	—	—	—	—	—
xalan	106	5.29	4.46	5.90	—	—	—	4.81	3.90	5.55	132.14	0.82	0.78	0.90	—	—	—	1.64	1.59	1.68
zxing	162	3.57	3.36	3.85	8,252.57	7,972.77	8,712.91	—	—	—	108.98	1.56	1.52	1.63	36.35	35.40	37.87	—	—	—

sizes for these comparisons. The effect size for all comparisons marked as statistically significant in Table 2 was large ($\hat{A}_{12} \geq 0.71$) except for the comparison between Galette and Phosphor on "jme" with respect to execution time which had a medium effect size ($\hat{A}_{12} \geq 0.64$). The full results of these statistical tests are available in the supplemental materials for this paper.

We found that the overhead of Galette was statistically significantly less than MirrorTaint with respect to both execution time and memory usage on all benchmarks. This supports the claim made in prior work that using a mapping structure, like the MirrorTaint's mirrored heap, to store taint tags is slower than using shadow variables [Bell and Kaiser 2014]. Galette's overhead was statistically significantly less than Phosphor's on eight benchmarks and greater than Phosphor's on two benchmarks with respect to execution time. Galette's overhead was statistically significantly less than Phosphor's on six benchmarks and greater than Phosphor's on four benchmarks with respect to memory usage. By using a mirrored mapping structure to store only the taint tags associated with arrays, Galette was able to achieve a performance profile that was competitive with that of Phosphor, which uses pure shadowing.

5.3 Threats to Validity

Our functional benchmark suite included only synthetic programs and may not be representative of all JVM programs. As noted in Section 5.1.2, we needed to use synthetic benchmarks in order to determine the ground truth expected taint tag propagation for each benchmark. Furthermore, our suite does not cover all of the functionality offered by the JVM. However, we tried to ensure that common functionality like array operations and method calls were well tested. We also tried to cover new Java features like record types and variable handles particularly thoroughly.

The ground truth expected taint tag propagation for each functional benchmark was manually determined and, therefore, could be incorrect. However, we constructed each functional benchmark so that the expected taint tag propagation was easy to follow and double-checked benchmarks on which at least one taint tracking system reported incorrect labels. Our ground truth expected taint tag propagation and functional benchmark suite are available in the artifact for this work (described in Section 8).

Our performance benchmark suite included only twenty workloads and may not be representative of all workloads. However, it is a well-established benchmark suite constructed for performance benchmarking in the JVM.

Our choice for the number of warm-up iterations in performance trials may not have been large enough to fully mitigate performance fluctuations that occur when the JVM is first started. To further mitigate the effect of these fluctuations, we took repeated measurements using multiple JVM invocations and report confidence intervals across these measurements in accordance with the recommendations of Georges et al. [2007].

Our choice of maximum heap size, 6 gigabytes, prevented measurements from being recorded on "graphchi" for MirrorTaint. Additionally, the maximum heap size can affect the frequency and duration of garbage collection in the JVM impacting execution times.

6 Related Work

Dynamic Taint Tracking for the JVM. Franz et al. [2005] trace relationships between strings in the JVM by instrumenting string-related classes. Halfond et al. [2006] also track relationships between strings by replacing instances of string-related classes with instances of corresponding meta-classes. These meta-classes are used to store and propagate taint tags. Chin and Wagner [2009] improve upon these approaches by allowing individual characters in a string to be tainted and traced independently. These approaches are unable to track non-string values.

Nair et al. [2008] perform dynamic taint tracking by modifying the Kaffe JVM interpreter. They extended the JVM stack, objects, and arrays to add space for storing associated taint tags. Enck et al. [2010] instrument the Dalvik VM interpreter to store and propagate taint tags. Taint tags are stored adjacent to original values in Dalvik's internal data structures. These approaches are able to perform fine-grained taint tracking on all forms of data. However, they rely on interpreter-specific properties and, therefore, are difficult to port to commodity JVMs.

Bell and Kaiser [2014] and Perkins et al. [2020] propose shadowing which, as described in Section 2.2.1, uses Java bytecode instrumentation to store and propagate taint tags on all forms of data by directly shadowing original values and instructions. Wang et al. [2022] extend Bell and Kaiser [2014]'s approach to support inter-node taint tracking in JVM-based distributed systems. Ouyang et al. [2023] propose mirroring which, as described in Section 2.2.1, uses a separate mirror-space to store and propagate taint tags. Like Galette, mirroring and shadowing are able to port to different JVMs, because they do not rely on interpreter-specific properties. However, neither mirroring nor shadowing address signature polymorphism or the JPMS.

Dynamic Taint Tracking for Other Managed Runtimes. Karim et al. [2020] present a platform-independent approach for dynamic taint tracking in JavaScript that instruments applications to omit instructions for an abstract machine that encode the flow of information. When the application terminates, the omitted instructions are executed to trace the flow of information. Aldrich et al. [2023] improve upon Karim et al. [2020]'s approach to add support for `async` and `await`. Eghbali and Pradel [2022] propose an instrumentation-based, general-purpose dynamic analysis framework for Python and use this framework to build a dynamic taint analysis for Python. Son et al. [2013] track the flow of string values in PHP applications using a PHP interpreter extension.

Applications of Dynamic Taint Tracking. Halfond et al. [2006] detect and prevent SQL injection attacks by tracking the flow of trusted values into SQL queries. Hough et al. [2020] detect injection vulnerabilities in web applications by tracking the flow of untrusted values into vulnerable program locations. Cox et al. [2014] prevent passwords from being leaked by tracking the flow of confidential information into public channels. Huo and Clause [2014] identify test cases quality issues by tracking information flows into test assertions. Attariyan and Flinn [2010] aid in troubleshooting configuration errors by tracking the flow of configuration values into observed errors. Clause and Orso [2009] assist in program debugging by tracking the flow of program inputs into failures. Kukucka et al. [2022] generate inputs capable of finding bugs deep in a program's execution by tracking the flow of input values into conditional branches.

7 Conclusion

Existing approaches to dynamic taint tracking in the JVM perform poorly on modern JVMs due to signature polymorphism and the JPMS. Galette, our approach for dynamic taint tracking in the JVM, provides precise, robust taint tag propagation in the presence of modern Java features. Galette is able to trace the flow of information through signature polymorphic methods by using a thread-local frame store. By modifying the primordial module, Galette is able to circumvent constraints imposed by the JPMS. In our empirical evaluation, we found that Galette was able propagate taint tags with perfect accuracy while preserving the original semantics of programs. We also found that Galette's runtime and memory overheads were significantly less than MirrorTaint's overheads and competitive with Phosphor's overheads.

8 Data Availability

The full results of the statistical tests that we conducted in our evaluation of this work and a detailed listing of our benchmark groups are available in our supplemental materials: https://doi.org/10.6084/m9.figshare.28737908.v1. Galette's source code, our experimental scripts, and our raw experimental data are publicly available under the BSD 3-Clause License as part of our artifact [Hough and Bell 2025].

Acknowledgements

This work was supported in part by the US National Science Foundation under grants NSF CCF-2100015 and CCF-2100037.

References

Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2023. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 153, 4 pages. https://doi.org/10.1145/3551349.3559522

Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI '10)*. USENIX Association, USA, 237–250.

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

Jonathan Bell and Phosphor Contributors. 2024. Phosphor. https://central.sonatype.com/artifact/edu.gmu.swe.phosphor/Phosphor.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Pierre Carbonnelle. 2024. PYPL PopularitY of Programming Language Index. https://pypl.github.io/PYPL.html.

D. Chandra and M. Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE Computer Society, 463–475. https://doi.org/10.1109/ACSAC.2007.37

Erika Chin and David Wagner. 2009. Efficient Character-Level Taint Tracking for Java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services* (Chicago, Illinois, USA) *(SWS '09)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1655121.1655125

James Clause and Alessandro Orso. 2009. Penumbra: Automatically Identifying Failure-relevant Inputs Using Dynamic Tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1572272.1572301

Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure Password Tracking for Android. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 481–494. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox

DaCapo Project Contributors. 2023a. dacapo-23.11-chopin RELEASE NOTES 2023-11. https://github.com/dacapobench/dacapobench/blob/fd292e92f8c40495a6ca05ff3b8a77c6c4265606/benchmarks/RELEASE_NOTES.md.

DaCapo Project Contributors. 2023b. The DaCapo Benchmark Suite (version dacapo-23.11-chopin). https://github.com/dacapobench/dacapobench/releases/tag/v23.11-chopin.

Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/taintdroid-information-flow-tracking-system-realtime-privacy-monitoring

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

M. Franz, D. Chandra, and V. Haldar. 2005. Dynamic Taint Propagation for Java. In *Computer Security Applications Conference, Annual*. IEEE Computer Society, Los Alamitos, CA, USA, 303–311. https://doi.org/10.1109/CSAC.2005.21

Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. 474–484. https://doi.org/10.1109/ICSE.2009.5070546

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 175–185. https://doi.org/10.1145/1181775.1181797

Katherine Hough and Jonathan Bell. 2025. Artifact for "Dynamic Taint Tracking for Modern Java Virtual Machines". (4 2025). https://doi.org/10.6084/m9.figshare.26880619.v2

Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 284–296. https://doi.org/10.1145/3377811.3380326

Chen Huo and James Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE 2009, Proceedings of the 31st International Conference on Software Engineering*. Vancouver,

BC, Canada, 199–209.

James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628

Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2013. The Java Virtual Machine Specification: Java SE 21 Edition. https://docs.oracle.com/javase/specs/jvms/se21/jvms21.pdf.

Stephen McCamant and Michael D. Ernst. 2008. Quantitative Information Flow As Network Flow Capacity. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 193–205. https://doi.org/10.1145/1375581.1375606

MirrorTaint Contributors. 2023. MirrorTaint (commit 54fcbfc). https://github.com/MirrorTaint/MirrorTaint.

Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. 2008. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electron. Notes Theor. Comput. Sci.* 197, 1 (feb 2008), 3–16. https://doi.org/10.1016/j.entcs.2007.10.010

New Relic, Inc. 2023. *2023 State of the Java Ecosystem.* Technical Report. New Relic, Inc. https://newrelic.com/sites/default/files/2023-04/new-relic-2023-state-of-the-java-ecosystem-2023-04-20.pdf

Oracle Corporation. 2023a. Java Platform, Standard Edition & Java Development Kit: Version 11 API Specification. https://docs.oracle.com/en/java/javase/11/docs/api/index.html.

Oracle Corporation. 2023b. Java Platform, Standard Edition & Java Development Kit: Version 21 API Specification. https://docs.oracle.com/en/java/javase/21/docs/api/index.html.

Oracle Corporation. 2023c. JDK Releases. https://www.java.com/releases/.

Oracle Corporation. 2024a. Java Native Interface: Introduction. https://docs.oracle.com/en/java/javase/21/docs/specs/jni/intro.html.

Oracle Corporation. 2024b. jlink Tool Reference. https://docs.oracle.com/en/java/javase/11/tools/jlink.html.

Oracle Corporation. 2024c. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. https://jcp.org/en/jsr/detail?id=292.

Oracle Corporation. 2024d. JSR 376: Java Platform Module System. https://www.jcp.org/en/jsr/detail?id=376.

Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. 2023. MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2514–2526. https://doi.org/10.1109/ICSE48619.2023.00210

OW2 Consortium. 2024. ASM (version 9.6). https://asm.ow2.io/.

Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

Jeff Perkins, Jordan Eikenberry, Allessandro Coglio, and Martin Rinard. 2020. Comprehensive Java Metadata Tracking for Attack Detection and Repair. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 39–51. https://doi.org/10.1109/DSN48063.2020.00024

Phosphor Contributors. 2018. Phosphor Issue #62. https://github.com/gmu-swe/phosphor/issues/62.

Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 135–148. https://doi.org/10.1109/MICRO.2006.29

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, USA. https://doi.org/10.14722/ndss.2017.23404

John Rose. 2013. Method handle invocation. https://wiki.openjdk.org/display/HotSpot/Method+handle+invocation.

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1181–1192. https://doi.org/10.1145/2508859.2516696

TIOBE Software BV. 2024. TIOBE Index for January 2024. https://www.tiobe.com/tiobe-index/.

Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2023. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Softw. Engg.* 28, 1 (jan 2023), 57 pages. https://doi.org/10.1007/s10664-022-10247-x

András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. http://www.jstor.org/stable/1165329

Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1072–1084. https://doi.org/10.1109/ICSE43902.2021.00100

Dong Wang, Yu Gao, Wensheng Dou, and Jun Wei. 2022. DisTA: Generic Dynamic Taint Tracking for Java-Based Distributed Systems. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 547–558. https://doi.org/10.1109/DSN53405.2022.00060

T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy*. 497–512.

Received 2024-09-03; accepted 2025-04-01