

# Paper Reading of Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs, and its application on flaky tests

Yiwei Yang

School of Information Science and Technology  
ShanghaiTech University  
Shanghai, China  
yangyw@shanghaitech.edu.cn

**Abstract**—With the flaky test [6], that in continuous unit test/ fuzzing/ regression tests integration, some of them may pass and not pass from time to time because of (non)order dependencies variables, , is getting more severe for concurrent programs, we have to come up with a software engineering method to identify and fix. Other than [2], a tool to identify the flaky test by checking original order passes, running different thread order configurations, and finally rerunning the truncated failing order and truncated original order. We could use a previous Dynamic Taint Analysis on order dependent variables to get a possible order dependent variable to automate the debugging process. Phosphor [1] is a taint tracking system based on bytecode modifications. The object of taint tracking is the tracking of specific data variables. can be used to find brittle test,

**Index Terms**—JVM, program analysis, dynamic taint analysis, flaky test

## I. Paper Summary

### A. Basic principles of dynamic taint analysis

Taint analysis can be abstracted into a triplet  $\langle \text{sources}, \text{sinks}, \text{processor} \rangle$  where source represents the direct introduction of untrusted data or confidential data into the system, sink represents direct generation of security-sensitive operations and processor the entire process of data transmission and processing.

For taint convergence points, they can be broadly classified conceptually into 3 categories.

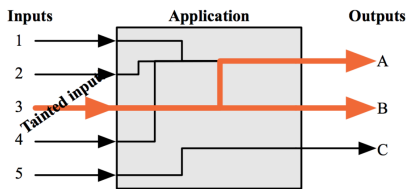


Fig. 1. Dynamic Taint Analysis

- 1) Use heuristic strategies for tagging. e.g. mark network data transmission.
- 2) Manually mark sources and aggregation points depending on the API or important data types called by the specific application. e.g. filesystem access.

- 3) Automatically identify and tag taint sources and aggregation points using statistical or machine learning techniques. [7]

Phosphor first taints the variable, cast processor on java bytecode to propagate the taint.

- 1) Leverages benefits of interpreter-based approaches (information about variables) but fully portably
- 2) Instruments all byte code that runs in the JVM (including the JRE API) to track taint tags
  - a) Add a variable for each variable
  - b) Adds propagation logic

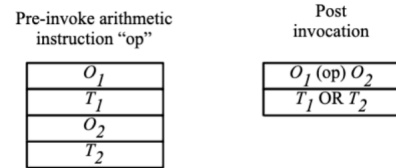


Fig. 2. Stack Code Propagation logic

### B. Connection with Flaky tests

When a developer commits code to a repository, tests are run to see if the changes have broken any functionality. Every new test failure would ideally be due to the developer's most recent modifications, allowing the developer to focus on fixing these problems. Unfortunately, some failures are due to flaky tests rather than the most recent updates. When executed on the same version of the code, a flaky test can pass or fail non-deterministically, flaky tests may also pass when they should have failed. Flaky testing are unavoidable in most modern applications. Take, for example, a Google system test that involves loading a website with an ad inserted in it. If the ad serving system becomes overburdened and is unable to deliver an ad within a reasonable amount of time, the test may be sent a page without ads. In this situation, the test runner might not be able to tell the difference between a malfunctioning ad server (which may not be serving ads to any clients) and a functional ad server that just dropped the request. Test

order dependencies are closely linked to flakiness or tests that can fail unexpectedly if run in a different order. This early work (PraDet [8]) looked at how to quickly isolate tests to avoid flakiness and how to exactly discover which tests depend on one other, allowing developers to detect which orderings will result in flakiness.

## II. Positive

- 1) The framework is generic and easy to use. You can embed any type of DTA to your project and report as a maven plugin.
- 2) Give a sound and complete propagation rule for all Java ASM.

Opcode(s)	Brief Description	Phosphor Modifications
aastore	Stores reference to array	Removes the taint tag for the index before storing; if the ref. is to a primitive array, boxes before storing
aload	Loads reference to array	Removes the taint tag for the index to load
anewarray	Allocates new array for references	If the array type is a multi-d primitive array, change to a container type
arraylength	Returns length of array as integer	Place the tag "0" just below return val on the operand stack after execution
areturn	Exit a method, returning the object reference at the top of the stack	If the top of the stack is a primitive array, boxes the array and its taint tags before return
astore	Store an object to a local variable	If the variable type is a primitive array, store the taint tags also to their variable. If the variable type is "Object" and the item being stored is a primitive array, box it.
baload, caload, daload, laload, faload, iload, laload	Loads a value from a primitive array	Removes the taint tag for the index to load; loads the taint tag for the corresponding element too
bastore, castore, dastore, fstore, istore, lstore, lstore	Stores a value to a primitive array	Removes the taint tag for the index to store to; stores the taint tag for the corresponding element too

Fig. 3. JVM propagation logic array part

- 3) Use taint tag storages

	Local variable	Method argument	Return value	Operand stack	Field
Object	Stored as a field of the object				
Object array	Stored as a field of each object				
Primitive	Shadow variable	Shadow argument	"Boxed"	Below the value on stack	Shadow field
Primitive array	Shadow variable	Shadow argument	"Boxed"	Array below value on stack	Shadow array field

Fig. 4. Phosphor Taint Tag Storage

## III. Negative

- 1) For specific purposes, because of the overhead of framework instrumentation, the effectiveness may not be as good as the specifically designed cases. For example, PraDet [5] introduced before is 10 times faster in finding order dependent variables.
- 2) Java ASM debugging is harder than you think. And the reference is not fruitful.

## IV. Soundness

The primary concern of dynamic implicit flow analysis is how to determine the range of statements to be marked under taint control conditions. Since the dynamic execution trajectory does not reflect the control dependencies among the instructions being executed, most of the current research uses offline static analysis to assist in determining the scope of implicit flow marking in dynamic taint propagation.

## A. Taint propagation analysis

- 1) Explicit Flow Analysis: Analyze how taint marks propagate with "data dependencies" between variables in the program.

```

void explicit_foo () {
    int a = source();
    int b = source();
    int x, y;
    x = a * 2;
    y = b + 4;
    sink(x);
    sink(y);
}

void implicit_foo () {
    String X = source();
    String Y = new String();
    for (int i=0; i<X.length(); i++) {
        int x = (int) X.charAt(i);
        int y = 0;
        for (int j=0; j<x; j++) y=y+1;
        Y = Y + (char) y;
        sink(Y);
    }
}

```

a and b are marked as taint sources by the predefined taint source function source. Assume that a and b are given the taint tags taint\_a and taint\_b, respectively.

Since the variable x in row 5 has a direct data dependency on variable a and the variable y in row 6 has a direct data dependency on variable b, explicit stream analysis will propagate the taint markers taint\_a and taint\_b to the variable x in row 5 and the variable y in row 6, respectively. Since x and y can reach the taint convergence point in rows 7 and 8, respectively, at the taint convergence point, we can follow a predefined strategy to draw conclusions, such as the information leakage problem of the code shown above.

- 2) Implicit Flow Analysis: Analyze how taint marks propagate with "control dependencies" between variables in the program, that is, how taint marks propagate from conditional instructions to the statements they control.

Variable X is a tainted string type variable. There is no direct or indirect data dependency (explicit flow relationship) between variable Y and variable X, but the tainted token on X can be implicitly propagated to Y through the control dependency. Specifically, the outer loop controlled by the loop condition in line 4 sequentially takes out each character in X, converts it to an integer and assigns it to variable x. The inner loop controlled by the loop condition in line 7 then assigns the value of x to y in a cumulative manner, and finally the outer loop passes y to Y one by one. In the end, the value of Y in line 12 is the same as the value of X. The program has an information leakage problem.

## V. Significance

The Phosphor framework has been widely used in [11]

- 1) Unknown Vulnerability Detection, like finding concurrent bugs [10].
- 2) Automatic Input Filter Generation. Forward symbolic execution can be used to automate the generation of filter.
- 3) Malware Analysis.
- 4) Test Case Generation.

"Polluter tests" are tests that change (or "pollute") the common state among tests in a test suite. Finding these tests is critical since the sequence in which the tests in the test suite are run can result in various test outcomes. Previously, the PolDet technique was presented for locating polluter tests in JUnit test runs on a normal Java Virtual Machine (JVM). Given that JavaPathFinder (JPF) provides desirable infrastructure

support, such as methodically investigating thread schedules, re-implementing techniques like PolDet in JPF is a worthwhile endeavour. [4]

## IX. Brainstorming

### A. Use in Order Dependent flaky tests

Flaky tests are a common problem in software testing, that the failure of test is non-deterministic. The flaky tests are usually order dependent. e.g.

```
class D {
    static Mutable2 staticField; // this will not be changed
    static {
        staticField = new Mutable2();
        staticField.next = new Mutable2();
    }
}
class Mutable2 { Mutable2 next /* = null */;}
class BasicFlakyAssertionTest {
    // should be null
    void t1() {System.out.println(D.staticField.next.next);}
    // passes if run after t2 but fails if run before
    void t3() {assertNotNull(D.staticField.next.next);}
    void t2() {D.staticField.next.next = new Mutable2();}
}
```

The solution to identify the above case is pretty simple in phosphor.

- 1) Taint all mutable field, array, primitive variable
- 2) propagate as in the Phosphor
- 3) if 2 paths meet in the assertion's variable there's OD.

### B. Refinement of the effectiveness

Basically, if you get more information from the analysis, the less information you need to cache during runtime.

- 1) Use JPF - symbolic execution for getting runtime type info/deterministic path to minimize the processor state
- 2) Can utilize prediction-based taint tracking (propagation) approach, and design statistical experiments to observe some special properties of load and store instructions in the CPU instruction stream.
- 3) Can transform dynamic taint analysis into a problem handled by deferred exceptions.

## References

- [1] BELL, Jonathan; KAISER, Gail. Phosphor: Illuminating dynamic data flow in commodity jvms. *Acm Sigplan Notices*, 2014, 49.10: 83-101.
- [2] W. Lam, R. Oei, A. Shi, D. Marinov and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 312-322, doi: 10.1109/ICST.2019.00038.
- [3] ANAND, Saswat; PĂSĂREANU, Corina S.; VISSER, Willem. JPF-SE: A symbolic execution extension to java pathfinder. In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer, Berlin, Heidelberg, 2007. p. 134-138.
- [4] YI, Pu, et al. Finding Polluter Tests Using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 2021, 46.3: 37-41.
- [5] HUO, Chen; CLAUSE, James. 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*. 2014. p. 621-631.
- [6] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 202 (November 2020), 29 pages.
- [7] SHE, Dongdong, et al. Neutaint: Efficient dynamic taint analysis with neural networks. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020. p. 1527-1543.
- [8] GAMBI, Alessio; BELL, Jonathan; ZELLER, Andreas. Practical test dependency detection. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018. p. 1-11.
- [9] Fang-Hsiang Su, J. Bell, G. Kaiser and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1-10, doi: 10.1109/ICPC.2016.7503720.
- [10] SUN, Mingshen; WEI, Tao; LUI, John CS. Taintart: A practical multi-level information-flow tracking system for android runtime. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016. p. 331-342.
- [11] SCHWARTZ, Edward J.; AVGERINOS, Thanassis; BRUMLEY, David. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: *2010 IEEE symposium on Security and privacy*. IEEE, 2010. p. 317-331.
- [12] YAN, Lok Kwong; YIN, Heng. SoK: On the Soundness and Precision of Dynamic Taint Analysis. *Formal. Taint*, 2017, 2017: 1-15.