# Order-dependent Variable Detection by Dynamic Taint Analysis

Yiwei Yang
2018533218
yangyw@shanghaitech.edu.cn

Yuchen Ji
2018533115
jiych1@shanghaitech.edu.cn

Hongchen Cao
2019533114
caohch1@shanghaitech.edu.cn

Abstract—Unreliable tests are a living nightmare for software development and test engineers. Flaky tests, which represent tests that can non-deterministically pass or fail for the same code version, become one of the major challenges on large-scale projects, including Facebook, Mozilla, and so on. Order-dependent(OD) tests are one of the widely-studied key categories of flaky tests, which means different execution orders result in different test results. Existing works mainly focus on detecting the presence of such tests at the test case level, so developers still need to find the order-dependent variables themself, which is time-consuming and error-prone. In this project, we implement tainting using the Phosphor framework on the Maven instrument plugin and then find out order dependent variables among test cases by dynamic taint analysis(DTA). We plan to evaluate our approach on the International Dataset of Flaky Tests(IDoFT) through both accuracy and efficiency.

Index Terms—Taint analysis, Flaky test, Parallel Bugs

## I. Introduction

### A. Motivation

Flaky tests are tests that can non-deterministically pass or fail for the same code version. This can affect the effectiveness of tests since developers cannot determine whether the test is failing because of code bugs or due to the flakiness of the test itself.

Many organizations have reported flaky tests as one of the major challenges on large-scale projects, including Facebook(Meta)[1], Mozilla[2] and so on.

### B. Research problem

As pointed out in prior studies[3], [4] test order dependency is one common cause of flaky tests. In this kind of test, because of some resources shared between the tests (e.g, variables or shared files), different execution orders will result in different test results (pass or fail). This kind of flaky test is categorized as order-dependent(OD) tests in previous work[5].

Our main challenge is to detect this kind of flaky test, at the same time, report the order-dependent variables that cause the test to be flaky to help the developer debug the flaky test.

### C. The state-of-art

Existing works like iDFlakies[5] or FlakeScanner[6] mainly focus on detecting the presence of flaky tests. For example, iDFlakies will run the test suite multiple times, each time permutes the order of tests. Then it will compare the result of different runs. If the test has both success and fail results and fails for at least two rounds, the test will be marked flaky. These tools mainly focus on detecting order-dependent flaky tests at the test case level, so developers still need to find the order-dependent variables themself.

### D. Contributions

In our project, we will make the following main contribution:

- Implement tainting using Phosphor framework on Maven instrument plugin;
- Use Dynamic Taint Analysis(DTA) to find out order dependent variables among test cases.

## II. Methodology

### A. System Design

The framework we use is based on the maven plugin, which does the following things:

1) Pass all the compilation options and source code to a maven pass.
2) Compile the code using the wrapper and do the instrumentation.
3) Provide the instrumentation Summary to maven report class.

The framework Bramble developed by Jon Bell plugged their Phosphor[7] to the instrumentation part which cast analysis on java bytecode assembly using visitor pattern logic to iterate all the declarations and statements.

1) Bramble: Then maven extension modifies the pom file to add additional calls to run tests (using taint tracking), ensure that the JVM is instrumented, and add Bramble reporting executions to build. Entry point here is BrambleLifecycleParticipant. You can see/debug what it does by running mvn -X (or mvnDebug and attach debugger)

In the test running process (debug using mvn -Dmaven.surefire.debug or -Dmaven.failsafe.debug), DependencyTainter.startingTest is called before each test, and is passed the test name. This will walk the heap and apply taint tags. Right now this does not add a marker for the test name, but it could in createLabel. Taint tags for assertions are checked in AssertionChecker.

The existing instrumentation code in Bramble is already powerful, where we find the call to DependencyTainter, and also in the RWTrackingMethodVisitor in datadep-detector[8], which implements tainting and checking on heap values.

## B. A prologue case

Listing 1: A prologue case

```
static class MutableHolder{
 int x; int y;
}
public static MutableHolder tmp;
public void test1(){ //run first
  tmp = new MutableHolder();
  tmp.x = 5;
}
public void test2(){ //run second
  tmp.y = tmp.x;
}
public void test3(){
  asserEquals(tmp.y, 5);
}
```

As shown in List. 1, we see the taint start can be arrays and mutable fields, MutableHolder in the above case which will be labeled OD by iDFlakies. Then we have the load-store array and mutable read-write using a new field and concatenation of arithmetic operations logic in Phosphor to easily do the modification. And the sinks to tmp.y = tmp.x which is a heap modification on tmp. The mutable variable tmp will be labeled as both polluter and victim through this case.

1) Taint Sources instrumentation using Phosphor: [9] introduces the test pairs - polluters and victims appear at the same time, and both of them can take place multiple times from probability analysis, and brittle tests are the false positive for the flaky tests, we can filter them out by tests reported by our implemented Oracle Polisher[10]. The polluters are the order-dependent sink, in this case, the victim is the sink variable we have to mark the destination of dynamic taint analysis. We will add New Taint at writes to Array, Field, that marked the possible polluter in the paper.

For instance, List. 2 shows the code if we want to lable a variable.

Listing 2: Label Variable

```
private static void taintField(Object instance, FieldInfo field) {
    if (!field.isFinal()) {
        Taint<Dependency> tag = Taint.withLabel(createLabel(instance, field));
        field.setTaint(instance, tag);
        if (field instanceof ClassOffsetInfo.ArrayFieldInfo) {
            deepTaint(((ClassOffsetInfo.ArrayFieldInfo)
                    field).getWrapper(instance), tag);
        } else if (field.isReferenceType()) {
            deepTaint(field.getValue(instance).getValue(), tag);
        }
    }
}

private static void deepTaint(Object obj, Taint<Dependency> tag) {
    if (obj instanceof TaintedPrimitiveWithObjTag) {
        ((TaintedPrimitiveWithObjTag) obj).taint = tag;
    } else if (obj instanceof LazyArrayObjTags) {
        deepTaint((LazyArrayObjTags) obj, tag);
    } else if (obj instanceof String) {
        getStringValueTag((String) obj).setTaints(tag);
    }
}

private static void deepTaint(LazyArrayObjTags wrapper, Taint<Dependency>
        tag) {
    if (wrapper != null) {
        wrapper.setTaints(tag);
        if (wrapper instanceof LazyReferenceArrayObjTags) {
            LazyReferenceArrayObjTags arr = (LazyReferenceArrayObjTags)
                    wrapper;
            if (arr.val != null) {
                for (Object obj : arr.val) {
                    deepTaint(obj, tag);
                }
            }
        }
    }
}

private static void getTaint(Object obj, Taint<Dependency> tag) {
    ClassOffsetInfo classInfo = ClassOffsetInfo.getInstance(obj.getClass());
    Set<Dependency> s = null;
    for (FieldInfo field : classInfo.getInstanceFields()) {
        Taint<Dependency> taint = (Taint<Dependency>) field.getTaint(obj);
        if (taint != null && !tag.isEmpty()) {
            s.addAll(Arrays.asList(taint.getLabels(new Dependency[0])));
        }
    }
    dependencyMap.put(obj, s);
}
```

## C. Taint Processor Indentification

The logic we just have to insert by the following modification to Bramble.

1) Turn off the DependencyTainter in Bramble
2) Add something that applies taint tags at heap writes, and store the test name as part of that taint tag.
3) Add something that checks taint tags at heap reads (store these results somewhere useful for later, like the AssertionChecker)

## D. Taint Sinks Logic

The sinks can be taints at any heap read, which includes: Array, Field, Static field, Check taints at assertions (this is what AssertionChecker in Bramble does).

Notably, when Null Pointer Exception comes. We can detour the exception and accept as much taint result as possible. For instance, as shown in List. 3 we may not report the NPE although the orders t3,t2 can make NPE.

Listing 3: Taint Sinks Logic

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

## E. Output

It will output the tainted program into a separate folder. The bash script will collect the result of instrumented logic, e.g. logs of how much order dependent variables are collected, the debug info on how the variable is propagated to the sink.

## F. Key features/functions

1) Assertion Detector: If we want to record all the assertions, we can first let the checking mv accept the MethodVisitor from Phosphor's asm API, since the exception will appear in the main body in all methods. Here,

we record all the information that we needed to store in the class and finally report in the BrambleReportFactory.

2) Write Labeler and Write Detector: Here's the proposed implementation for the write source, processor, and sink. We have to insert code for label creation in the array creation which has been down in the upper deep taint. Here we mark all the field creations starting with an instrumented print and then instrument the tag using dup and add taint manually, here we mark all the inside variables as the source.

The heap modification consists of array modification assembly and arithmetic on and heap variable. Then we implement the processor logic. For the sink part, the checking process is similar to the labeling, we just need to call the Phosphor API to end the taint process.

3) The Output Flow: The output flow is stored in a map of variable line code to the resulting assembly, other metadata consists of the class metadata. These will be formatted as the surefire standard output.

## III. Evaluation

In the following we describe the dataset and evaluation metrics we plan to use for our experiments, which we may modify depending on the actual progress of the future project.

### A. Dataset

International Dataset of Flaky Tests(IDoFT) [5], [11] is a dataset of current and fixed flaky tests in real-world projects. The goal of IDoFT is to crowd-source such a dataset and to compile a variety of information (e.g., failure rates, flakiness-introducing commits) about flaky tests. This dataset is made possible by Wing Lam, Garvita Allabadi, and the students from the Fall 2020 CS 527 class at the University of Illinois. In addition there are many publications that have contributed to this dataset, including [9], [12], [13], [14], [15].

IDoFT contains 314 projects, 3742 flaky tests, 1263 fixed flaky tests, 191 flaky tests where no fixes are needed, and 2070 unfixed flaky tests. For each flaky test, IDoFT provides detailed information including project URL, SHA detected, module path, fully-qualified test name, category, status, PR link, days to address PR, and notes. We believe that the dataset of this size and widely used is sufficient for future objective evaluation.

### B. Metric

We plan to evaluate both in terms of accuracy and efficiency, which are widely-used metrics in previous studies [7], [16]. Specifically, the former represents the accuracy of our method in discovering order-dependent variables on the dataset. The latter covers two aspects - runtime and memory usage. We will compare with the SOTA approaches mentioned in Sec. I-C to validate the performance of the method.

## IV. Related Work

### A. Taint analysis

Taint analysis is quickly becoming a staple technique in security analyses. It can be categorized into dynamic and static taint analysis, called DTA and STA, respectively. DTA runs a program and observes which computations are affected by predefined taint sources such as user input [17]. Alternatively, STA, propagates taints based on an overestimation of all possible program paths leading to the detection of all possible taint flows with no false negatives but some false positives due to infeasible paths [16]. Bell et al. [7] present Phosphor, the first portable general-purpose dynamic taint tracking system for the Java Virtual Machine (JVM) that simultaneously achieves performance, soundness, precision, and portability.

### B. Flaky test

Regression testing is a widely adopted practice in modern software development. If a test does not behave deterministically but passes and fails when run multiple times without any changes to the code, the test is regarded as flaky [18]. Flaky test detection is used in a variety of applications. Romano et al. [19] analyze 235 flaky UI test samples found in 62 projects from both web and Android environments. They identify the common underlying root causes of flakiness in the UI tests, the strategies used to manifest the flaky behavior, and the fixing strategies used to remedy flaky UI tests. Dong et al. [6] present an approach and tool FlakeScanner for detecting flaky tests through exploration of event orders. Their experiments on the subject-suite FlakyAppRepo show FlakeScanner detected 45 out of 52 known flaky tests as well as 245 previously unknown flaky tests among 1444 tests. In addition to accuracy, some studies focus on improving the speed of flaky test detection. Cordeiro et al. [20] present SHAKER, an open-source tool for detecting flakiness in time-constrained tests by adding noise in the execution environment. SHAKER was able to discover more flaky tests than ReRun, which is the most popular approach in the industry, and in a faster way; besides, their approach revealed tens of new flaky tests that went undetected by ReRun even after 50 re-executions.

## References


[1] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2018, pp. 1–23.

[2] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840.

[3] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 385–396.

[4] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653.

[5] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019. IEEE, 2019, pp. 312–322.

[6] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," in ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 367–378.

[7] J. Bell and G. E. Kaiser, "Phosphor: illuminating dynamic data flow in commodity jvms," in Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 83–101.

[8] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018, pp. 1–11.

[9] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12651. Springer, 2021, pp. 270–287.

[10] C. Huo and J. Clause, "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, pp. 621–631.

[11] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of nondeterministic specifications," in 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016. IEEE Computer Society, 2016, pp. 80–90.

[12] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 298–311.

[13] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in 31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020, M. Vieira, H. Madeira, N. Antunes, and Z. Zheng, Eds. IEEE, 2020, pp. 403–413.

[14] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," in Software Engineering 2022, Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25. Februar 2022, Virtuell, ser. LNI, L. Grunske, J. Siegmund, and A. Vogelsang, Eds., vol. P-320. Gesellschaft für Informatik e.V., 2022, pp. 57–59.

[15] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: a framework for automatically fixing order-dependent flaky tests," in Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 545–555.

[16] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," in 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. IEEE, 2021, pp. 796–812.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. IEEE Computer Society, 2010, pp. 317–331.

[18] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in Software Engineering 2022, Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25. Februar 2022, Virtuell, ser. LNI, L. Grunske, J. Siegmund, and A. Vogelsang, Eds., vol. P-320. Gesellschaft für Informatik e.V., 2022, pp. 37–38.

[19] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, 2021, pp. 1585–1597.

[20] M. Cordeiro, D. Silva, L. Teixeira, B. Miranda, and M. d'Amorim, "Shaker: a tool for detecting more flaky tests faster," in 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. IEEE, 2021, pp. 1281–1285.