Order-dependent Variable Detection by Dynamic Taint Analysis

Yuchen Ji, Yiwei Yang, Hongchen Cao
School of Information Science and Technology
ShanghaiTech University
Shanghai, China 201210
{jiych1,yangyw,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 paper, 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 evaluate our approach on the International Dataset of Flaky Tests(IDoFT) and achieve the accuracy that PR created can achieve.

Index Terms—Taint analysis, Flaky test, Parallel Bugs

I. Introduction

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.

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].

In this paper, we propose an approach 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.

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.

In summary, our contributions are as follows

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

II. Background

A. Dynamic Taint Anlysis

Dynamic taint analysis is used to track the flow of information between sources and sinks. Any program value relies on the data calculated from the tainted source is considered tainted. Any other value is considered to be untainted. The taint policy P determines exactly how taint flows during program execution, which operations introduce new taint, and what checks are performed on tainted values. The specifics of the taint policy may vary depending on the analysis application. for example, the taint tracking policy for unlocking malware may differ from that for attack detection. However, the basic concept remains the same. It is natural to express dynamic taint analysis in terms of the operational semantics of the language, since it is performed on the code at runtime. Taint policy behaviour, whether taint propagation, introduction or checking, is added to the operational semantic rules.

B. Flaky tests

Flaky Tests are tests that sometimes fail and sometimes succeed, while both the subject and the test conditions remain the same. Thus, Flaky Tests are in fact unstable tests, or tests that fail (or succeed) at random. Flaky Tests are found in repetitive tests. Flaky Tests are more a product of automated testing than manual testing, and Flaky Tests have become a common and prominent problem with the spread of automated testing. There are many reasons why Flaky Tests arise, such as asynchronous waiting, concurrency, remote services, test dependencies, and so forth. The problem with flaky tests is that they slow the CI/CD pipeline and erode confidence in testing processes. To spot flaky tests, developers need to compare

test results from multiple test runs. This analysis would be a time-consuming process to perform manually.

C. Order-dependent Tests

Test-order dependence results in order-dependent tests. An order-dependent test is a flaky test whose pass/fail outcome depends on the test order in which it runs. In other words, there exists a test order where the order-dependent test passes and another different test order where the test fails. Prior work [7] showed that order-dependent tests are among the top three most common kinds of flaky tests. A widely reported example happens when Java projects updated from Java 6 to Java 7. Java 7 changed the implementation of reflection, which JUnit uses to determine the test order to run tests in. Many tests fail due to the tests being run in a different test order from before, requiring developers to manually fix their test suites [8].

Shi et al. [9] categorized order dependent tests into 2 types: victim and brittle. Victim tests will pass when run in isolation, but fail when other tests run before them and pollute their states. Brittle tests will fail when run in isolation, only pass when run with other tests that set its states. We will also use this categorization.

III. Methodology

A. System Design

To detect order dependent variables, we will use dynamic taint analysis, which requires a taint source and a taint sink. In our analysis, we define any heap write, e.g. write to static variable or array element as source, any heap read, e.g. read of static variable or array element as sink.

Therefore, for the taint analysis part, our system will do the following:

1) Instrument at any heap variable write, including static and non-static ones, mark them as taint sources.
2) Instrument at any heap variable read, including static and non-static ones, check if they point to any tainted data that is not in the same testcase. If they point to other testcases, this is an order dependent variable.
3) Collect order dependent data and generate a report

B. A motivating example

We use the following case as an example:

Listing 1: A prologue case

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

As shown in List. 1, this is an order dependent test of Brittle type. For assetion in test2() to pass, test1() must run before test2(). In our instrumentation, we will taint tmp.x when assigning the value, then when test2() runs, tmp.y will be tainted with taint tag of tmp.x. Finally when tmp.y is read in assertion, we will check its tag, and find that it has a taint tag from other testcase (test1()), which means test2() is a Brittle test and tmp.y is a order dependent variable.

IV. Implementation

Our implementation is based on two important infrastructures, Phosphor and Bramble. Phosphor is used for dynamic tainting, while Bramble is used to integrate with Maven projects and instrument testcases.

A. Phosphor

Phosphor[10] is a system for performing dynamic taint analysis in the JVM that simultaneously achieves goals of performance, soundness, precision, and portability. It achieves this goal by instrumenting byte code without changing the JVM itself. Tracking is done by combination of instrumented code, instrumented library (incluing JRE libs) and Phosphor runtime.

To support easy tainting, Phosphor exports a set of APIs in the MultiTainter class. To taint a variable, simply replace the assigned value with MultiTainter.taintedX() method call. To get taint tag of a variable, simply call MultiTainter.getTaint().

B. Bramble

Bramble is a maven infrastructure to run tests while performing dynamic taint tracking. It will act as a Maven extension/plugin, which does the following things:

1) Compile the project using Maven without modification
2) Before test phase, modifies the pom file to add additional calls to first instrument testcases using Phosphor, then run tests and ensure that the testcase is instrumented.
3) Summarize the instrumented run result and provide it to Maven's surefire report.

C. Actual instrumentation

With the two infrastructure, we can start implementation. Our goal is to taint heap write and detect taint at heap reads. As discussed in IV-A, we have simple APIs to do this. However, instrumentation works at bytecode level, so we need to translate the calls to byte code. Luckily, Phosphor's instrumentation is done using ASM[11] library, which provides handy wrappers around common byte code operations.

For example, if we taint the testcases in List. 1, the result would be List. 2, where createTag and checkTaintTag are helper methods.

Listing 2: Tainted case

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

To add function calls in bytecode, we make use of ASM's visitMethodInsn method, this will take care of inseting appropriate function call instructions.

However, we have two issues to consider. One is to properly support different types, including object and primitive types. The other is to maintain the stack between our inserted method calls.

1) Types: In Java, primitive types are not objects, therefore we must deal with primitive types separately. Luckily, Phosphor provides support for these, we just need to call corresponding function for each primitive type.

As for objects, Phosphor provides a templated method to taint objects. However, templated method will return Object under the hood, so we have to add a cast (by using the CHECKCAST opcode) to cast the value back to the original type.

As for arrays, each array element will be tainted separately, while the array itself will be treated as a normal object. The rationale is that in Java arrays are subclass of object, so we can safely do the cast.

2) Maintain stacks: The other issue is we need to maintain stacks for our method calls. When tainting objects, since we will return the tainted object, we have already maintained the stack to be unchanged. However, when we are checking taint tags, since it just takes an argument and return void, we have to maintain the stack through duplicates.

Our checking method has a signature of void check(Object, Taint), while get taint tag method has a signature of Taint get(Object). So to maintain stack balance, it will undergo the following states:

value -> value,value -> value,taint -> taint,value -> value,taint,value -> value,value,taint -> value

An example of doing this is shown in List. 3

Listing 3: Maintain stack for boolean

```
// v
super.visitInsn(Opcodes.DUP);
// v,v
GET_TAINT_BOOLEAN.delegateVisit(this);
// v,t
super.visitInsn(Opcodes.SWAP);
// t,v
super.visitInsn(Opcodes.DUP_X1);
// v,t,v
super.visitInsn(Opcodes.SWAP);
// v,v,t
CHECK_TAG_BOOLEAN.delegateVisit(this);
// v
```

However, this doesn't work for long and double types, as they take two slots on stack. So we need to define a special swap routine for them, shown in List .4

Listing 4: Swap routine for long and double

```
public void swap(Type stackTop, Type belowTop) {
    if (stackTop.getSize() == 1) {
        if (belowTop.getSize() == 1) {
            // Top = 1, below = 1
            super.visitInsn(Opcodes.SWAP);
        } else {
            // Top = 1, below = 2
            super.visitInsn(Opcodes.DUP_X2);
            super.visitInsn(Opcodes.POP);
        }
    } else {
        if (belowTop.getSize() == 1) {
            // Top = 2, below = 1
            super.visitInsn(Opcodes.DUP2_X1);
        } else {
            // Top = 2, below = 2
            super.visitInsn(Opcodes.DUP2_X2);
        }
        super.visitInsn(Opcodes.POP2);
    }
}
```

Using this new swap routine, we can now maintain the stack for long and double types, as shown in List .5

Listing 5: Maintain stack for long

```
// l
super.visitInsn(Opcodes.DUP2);
// l,l
BrambleMethodRecord.GET_TAINT_LONG.delegateVisit(this);
// l,t
swap(Type.INT_TYPE, Type.LONG_TYPE);
// t,l
super.visitInsn(Opcodes.DUP2_X1);
// l,t,l
swap(Type.LONG_TYPE, Type.INT_TYPE);
// l,l,t
BrambleMethodRecord.CHECK_TAG_LONG.delegateVisit(this);
// l
```

D. Checking taint

The actual checking process is pretty simple. We will check whether the taint tag at the specified read belongs to the same testcase. If not, it will be stored to a map where key is reader and value is writer. After that, this information will be formatted and provided to surefire report.

V. Evaluation

In the following we describe the dataset and evaluation metrics weuse for our experiments. We evaluated 460 current order-dependent flaky tests in 14 repos that exist in the idoft data set. For 120 of the tests have its PR for flaky test elimination, thus we can cross-validate our result for which is the source writer that makes victim pollution by the proposed modification to which variable. Some of them intrinsically have bugs for tests on the tested machine. The evaluation machine is cast on M1 Max with

Mac OS 12.4 with java version 8.0.332-librca since it's the only successful version for JVM instrumentation on Mac.

The phosphor requires the test to be successfully run and then out order-dependent flaky test output. So we remove many of the unrelated test files. Since the order dependency typically occurs within a file through the random order of different tests. For those who specify the order of running by @RandomOrder, Our tool will evaluate which variable triggers the failure.

This chapter will discuss how we managed to compile all the tests in OD flaky tests dataset, and how effective the tool is.

A. Dataset

International Dataset of Flaky Tests(IDoFT) [5], [12] 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 [13], [14], [15], [16], [9].

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. Steps to compile the tests

For All the maven tests, we just need to add the maven project to the sub folder of our bramble tool. Make sure the JUnit version of the project is stuck to 4.12. Moreover, we have to add a bramble-maven-extension to the maven project. We use -Dmaven.test.failure.ignore=true for not passed tests.

The maven extension will hook the maven compile process, first give a phosphor instrumentation output and then Brittle-test output.

1) Pitfalls in the phosphor instrumentation:

a) NoSuchMethod: In jboot ShiroSupportTest, it extends the JbootTestBase. Phosphor failed to instrument the methods from the supper class and output error ShiroSupportTest>JbootTestBase.startApp:-1->JbootTestBase.startApp$$PHOSPHORTAGGED:41 » NoSuchMethod

b) Reject Struct : java.lang.SecurityException: Rejected: edu. columbia. cs. psl. phosphor. struct. TaintedReferenceWithObjTag

The tainted tag may trigger a java language security exception

Name	Revision	Reported	Overall	Non-PR	FN Rate
nifi	41ff6f07	24	37	1	35.14
cloud-slang	2b5914c	10	10	2	0%
jboot	4bffb4df	4	6	1	33.33%
jicofo	2cae36e	3	4	2	25%
light-4j	fcded16	9	14	4	28.57%
openhtmltopdf	4a0612f	34	35	10	2.86%
elasticjob	bdfcaff0	21	10	0	0%
spring-cloud-gcp	7ab8b2c	5	8	0	37.50%
spring-boot	daa3d45	-	2	-	100%
spring-kubernetes	3351926	12	28	2	57.14%
				Overall	13.7984%

We have 13.7984% rate of false negative rate while the Non-PR is 22.

c) ClassCastException: java .lang .ClassCastException: {[}Ljava .lang .Object; cannot be cast to edu .columbia. cs. psl .phosphor .struct .LazyReferenceArrayObjTags

Some of the Objects cast to another self-defined class can not be compatible with this tool because our taint on one class and other classes are not identical and can not cast the ObjTag.

C. Effectiveness of the Tool

Our tool can find the taint source with a good positive rate, but the soundness and correctness.

1) Integration Test: We first run our integration test to dry test the effectiveness of our tool. These test are simple and one-directed.

a) ArrayFlakyTest: It first init static int array and string array, and non-static ones. If the t1() runs prior than t3(), the test run successfully and returns no brittles. However, when we swap the order, it report writer victim static field: edu.gmu.swe.bramble.ArrayFlakyTest.a <- [1234, 2345, 3456]

b) MutableFieldAssertionTest: Without the error triggering, the tool already output the writer to the vitim reader i1-7 that clearly marked flaky pollutor if the reader triggers flaky test by iDFlakies[5].

c) BasicBrittleAssertionTest: It automatically marked the reader and writer to the non-static variable i and b. If we specified the order of assertion failure, it will directly pointer to the pollutor that is definitely helpful for the debugging.

D. Flaky Test Dataset

For the following graph, we mark the Reported as the tests that reported as Order dependent by our test, the false negative rate is marked whether the reported variable is false negative compared to the PR. And Non-PR is marked the variable not reported by the PR so that we can use this source variable to debug the order dependent variable.

VI. 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 [18]. Bell et al. [10] 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 [19]. Flaky test detection is used in a variety of applications. Romano et al. [20] 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. [21] 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.

VII. Conclusion

Order-dependent flaky tests are reported as one of the major challenges on large-scale projects. Existing works like iDFlakies[5]mainly focus on detecting the presence of flaky tests. 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. In this paper, we propose an approach 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. In detail, we implement tainting using Phosphor framework on Maven instrument plugin and use Dynamic Taint Analysis(DTA) to find out order dependent variables among test cases. We evaluate our approach on International Dataset of Flaky Tests(IDoFT) and achieve the accuracy that PR created can achieve. The FN Rate of the tool is good so that it can be widely run on other order dependent flaky testset for programmer to debug their flaky tests. Also, the investigated 22 variables in the flaky testset that have not been observed by other programmer can better help the community to make their bugs eliminated.

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] 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, (FSE-22), Hong Kong, China, November 16 - 22, 2014, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 643–653.

[8] Java, "Junit and java 7," 2012, http://intellijava.blogspot.com/2012/05/junit-and-java-7.html.

[9] 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.

[10] 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.

[11] E. Kuleshov, "Using the asm framework to implement common java bytecode transformation patterns," 2007.

[12] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[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] 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.

[19] 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.

[20] 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.

[21] 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.