Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications

August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
Email: {awshi2,gyori,legunse2,marinov}@illinois.edu

Abstract—Some commonly used methods have non-deterministic specifications, e.g., iterating through a set can return the elements in any order. However, non-deterministic specifications typically have deterministic implementations, e.g., iterating through two sets constructed in the same way may return their elements in the same order. We use the term ADINS code to refer to code that assumes a Deterministic Implementation of a method with a Non-deterministic Specification. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. Further, ADINS code can lead to flaky tests—tests that pass or fail seemingly non-deterministically.

We present a simple technique, called NONDEX, for detecting flaky tests due to ADINS code. We implemented NONDEX for Java: we found 31 methods with non-deterministic specifications in the Java Standard Library, manually built non-deterministic models for these methods, and used a modified Java Virtual Machine to explore various non-deterministic choices. We evaluated NONDEX on 195 open-source projects from GitHub and 72 student submissions from a programming homework assignment. NONDEX detected 60 flaky tests in 21 open-source projects and 110 flaky tests in 34 student submissions.

I. INTRODUCTION

Non-deterministic specifications are not uncommon for many methods, including in the standard libraries of many programming languages. For example, the specification for the Object#hashCode() method in Java can return any integer. Non-deterministic specifications are not restricted to simple APIs. The order in which elements of a set are returned by an iterator is not-specified—it can be any order. The order in which entries in a SQL table are returned is also sometimes not specified—it depends on the query. Such specifications give implementers more freedom to develop various implementations for different goals, e.g., to optimize performance, while still satisfying the specification.

Even when specifications allow for non-determinism, typical implementations of such specifications are often deterministic, with respect to certain controlled sources. For example, Object#hashCode() could return the same integer (if one controls for all other sources, e.g., OpenJDK Java 8 could return a deterministic value on the first call if the underlying random() implementation in C is deterministic). The implementation of HashSet is such that iterating through the elements returns them in a deterministic order for one Java version, but that order can change between Java versions.

Code that Assumes a Deterministic Implementation of a Non-deterministic Specification—which we call ADINS code—is often bad. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. For example, Java code that assumes a specific iteration order of a HashSet, e.g., that a HashSet with elements 1 and 2 will be always represented as a string \{1, 2\} rather than \{2, 1\}, is ADINS and not robust: the Java implementation of HashSet can change such that the iteration order of the elements changes and the string differs.

Unexpected behavior of ADINS code can lead to flaky tests, which are tests that seem to non-deterministically pass or fail. Flaky tests are bad as they can mask bugs (pass when there are bugs) or raise false alarms (fail when there are no bugs). A test that executes ADINS code can be flaky if it assumes that some values are deterministic even if they can change: when the assumptions hold, the test passes, but when the assumptions do not hold, the test may fail. Not all flaky tests are due to ADINS code, e.g., a test asserting that a file system contains /tmp could pass on one machine but fail on another. Flaky tests are emerging as an active research topic, with recent work on characterizing [25], detecting [2], [4], [10], [12], [14], [38], and avoiding [1], [22] flaky tests. However, no previous research investigated ADINS code as a cause for flaky tests.

While flaky tests are an important problem in software practice and research, we also encountered them in teaching. Typically, the teaching staff grades students’ solutions to programming assignments using automated tests. These tests can be flaky, and as a result students with correct solutions may have failing tests, and students with incorrect solutions may have passing tests. We discuss more details from one recent course in Section IV-B. Besides educating people about flaky tests, how can we help practitioners in the real world and the students in our courses to detect more flaky tests faster?

We propose a simple technique, called NONDEX, to detect flaky tests due to ADINS code. We implement NONDEX for Java, but it can be easily generalized to any other language. In a nutshell, we identify 31 methods with non-deterministic specifications as discussed in Section III-A, wrote models for these methods to produce various non-deterministic choices, and use an execution environment that can explore various combinations of these non-deterministic choices. Our tool, called also NONDEX, modifies a regular Java Virtual Machine (in particular, the OpenJDK JVM version b132) to randomly explore choices by rerunning the test suites multiple times from scratch with different random seeds.
We evaluated NONDEX on two sets of programs: 195 open-source projects from GitHub and 72 student submissions from one homework assignment in our software-engineering course. We find NONDEX to be highly effective at detecting flaky tests in both open-source projects and student submissions. NONDEX detected 60 flaky tests in 21 of the 195 open-source projects. Because our experiments use some older project revisions, three of these tests were already fixed by the developers in the latest revision. (This fixing additionally confirms that flaky tests are important and that developers are willing to address them.) We confirmed that 57 tests are still present in the respective projects’ latest revision. We leave it as future work to properly debug these tests and file bug reports. For student submissions, NONDEX detected that 34 submissions, representing almost half of 72 considered, fail due to some ADINS code, with a total of 110 flaky tests detected. It is important to note that the homework assignment was designed a few years ago by a teaching assistant who had no knowledge of our research on flaky tests. We plan in the future to expose students to NONDEX and teach them to better detect and avoid ADINS code and flaky tests.

This paper makes the following contributions:

- **Problem.** We define the problem of ADINS code, identify it as a cause of flaky tests, and raise awareness about the problem of flaky tests in both software development practice and software engineering education.
- **Technique and Implementation.** We propose a simple technique for detecting flaky tests caused by ADINS code and describe our non-deterministic models and a tool that embody this technique.
- **Evaluation.** We evaluated our NONDEX technique on 195 open-source Java projects and 72 student code submissions. NONDEX detected 57 previously unknown flaky tests in open-source projects and three flaky tests that have been already fixed by the open-source software developers. NONDEX also detected 110 flaky tests in student submissions.

## II. Non-Deterministic Specifications

A non-deterministic specification allows for multiple implementations that can yield different outputs when executed with the same input; we consider “input” in a broad sense to include all interactions of code with its environment. For example, consider the method File#list() that returns a String array with names of all files and directories present in the directory on which the method was invoked. Its JavaDoc specification [6] states “There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.” This specification allows implementations to return names in any order even when executed with the exact same input (the state of the file system), hence this specification is non-deterministic. In contrast, consider the method File#exists() that returns a boolean value indicating whether or not the file on which the method was invoked exists. This specification is not non-deterministic; while the

```java
class Book {
    String title;
    String author;
    String getStringRepresentation() { ... }
}
class BookTest {
    @Test
    public void testGetStringRepresentation() {
        Book b = new Book("book","name");
        assertEquals("\"title\":\"book\",\"author\":\"name\"!",
                b.getStringRepresentation());
    }
}
```

Fig. 1. Example flaky test simplified from student code

returned value depends on the input (the state of the file system) and can be true or false on different machines, when executed on the same input, any implementation that conforms to the specification must return the same value.

### A. An Example Flaky Test

Code that (transitively) calls methods with non-deterministic specifications can be ADINS and lead to flaky tests. Figure 1 shows an example flaky test simplified from a student submission. The Book class has two fields, title and author. The method under test, getStringRepresentation(), uses a third-party JSON library that turns an object into a string. The test asserts that the result equals a hard-coded string that has the two fields in a particular order, first title and then author. However, the library uses a HashMap to store the mapping from fields to values, and iterates over this map to produce the resulting string. The iteration order over elements in a HashMap is not specified, so while this test can pass for one implementation, it can fail for another implementation that has author before title. NONDEX can detect such tests.

### B. Levels of Non-determinism

Some non-deterministic specifications can allow for multiple levels of non-determinism. Figure 2 presents an example: the class HashSet has a non-deterministic specification that can be (mis)interpreted in different ways. The JavaDoc specification [11] states “[HashSet] makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.”. Hence, the elements in the array returned by HashSet#toArray() can be in any order. The code first constructs an Integer HashSet object s with the elements 1 and 2 (lines 1–2). A deterministic implementation could return either of the two orders shown in the two assertions on lines 4 and 5, and one of the assertions should pass, while the other should fail.

Whether the other assertions pass or fail is more open to different interpretations of this non-deterministic specification. First, one could assume that two iterations on the same unchanged set object should yield the same order. However, the specification states that the order can vary “over time”, which could mean that the order in which elements are returned can change from one invocation to another even for the same set. Hence, the assertion on line 7 may get a different order and fail. Second, one could assume that the order should not change if the set is only read. Hence, the assertion on line 10
could fail or pass. Third, one could assume that if a set is modified and then restored to its original state, the order in which the elements are iterated can change from that before the modification of the set. Hence, the assertion on line 13 could either pass or fail. Fourth, one could assume that two sets constructed in exactly the same way would yield the same order, but if that does not hold, the assertion on line 17 can pass or fail. Fifth, one could assume that elements are iterated in the order of addition; line 21 creates a new set using different elements but added in the same order as in set \( s \), by their natural ordering. One could assume that both sets will be iterated in the same order—in which elements are added, or the natural order; depending on whether this assumption holds, the assertion on line 22 can pass or fail. (This final assumption is not unrealistic; LinkedHashSet indeed guarantees the iteration order over elements to be the same as that in which the elements are added [24].)

III. Technique

Our NONDEx technique detects flaky tests due to ADINS code making deterministic assumptions on non-deterministic specifications. Section III-A describes how we identified several non-deterministic methods in the Java Standard Library. Section III-B describes the models we developed for those non-deterministic methods. Section III-C presents some implementation details of NONDEx.

A. Identifying Non-deterministic Methods

Finding methods which have non-deterministic specifications is hard; in particular, one cannot easily look for non-deterministic implementations as individual implementations are deterministic most of the time. Rather, non-determinism occurs when non-deterministic specifications allow multiple implementations to behave differently from one another while still meeting the specification, even if each implementation is deterministic. For example, upgrading from Java 6 to Java 7 changed the order in which the Java Reflection API returned the list of methods in a class. JUnit uses the Reflection API for obtaining the list of methods to run, and on Java 6, methods were returned in one order, but in a completely different order in Java 7. This seemingly innocuous change caused tests run by JUnit to fail [21] due to test-order dependencies [1], [2], [10], [14], [22], [38]. Finding non-deterministic methods solely from the code is infeasible; one must reason about the specification itself to find if a method can be non-deterministic. This makes it inherently hard for any static or dynamic analysis technique to find such non-deterministic methods from one implementation.

To find non-deterministic methods in the Java Standard Library, we first searched for methods that may have such specifications and then carefully reasoned from their JavaDoc to determine if their specifications are indeed non-deterministic. We used two queries, based on (1) JavaDoc keywords and (2) return types. Specifically, the first query searches through JavaDoc for the following keywords that could indicate non-deterministic specifications: “order”, “deterministic”, and “not specified”. The second query searches for all public methods that return arrays. These queries produced many false positives, e.g., because not every method that mentions “order” is non-deterministic, and some methods that return arrays can return elements in a specified order. Our search is definitely not complete, and we leave as future work to develop better approaches to find non-deterministic specifications.

After inspection, we found the non-deterministic methods summarized in Table I. We tabulate the class name, method name(s), and the kind of specification non-determinism. We found three kinds, which we call “random”, “permute”, and “extend”. For random, the specific int returned by Object#hashCode() is not specified, so relying on it to return some specific value is ADINS. For permute, the specifications of some methods that return arrays or collections can have an unspecified order of elements. For extend, the

<table>
<thead>
<tr>
<th>Class</th>
<th>Method(s)</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Object # hashCode</td>
<td>random</td>
<td></td>
</tr>
<tr>
<td>java.util.HashMap # keySet, values, entrySet</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.util.concurrent.ConcurrentHashMap # keySet, values, entrySet, keys, elements</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.io.File # list, listFiles, listRoots</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.lang.Class # getClasses, getFields, getDeclaredFields</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.lang.reflect.Field # getDeclaredAnnotations</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.util.HashMap # keySet</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.DateFormatSymbols # getAvailableLocales</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.SimpleDateFormat # getAvailableLocales</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.DecimalFormatSymbols # getAvailableLocales</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.Locale # getAvailableLocales</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.Date # getAvailableLocales</td>
<td>permute</td>
<td></td>
</tr>
<tr>
<td>java.text.SimpleDateFormat # getZoneStrings</td>
<td>extend</td>
<td></td>
</tr>
</tbody>
</table>

Thus, when run on Java 6, methods were returned in one order, but in a completely different order in Java 7. This seemingly innocuous change caused tests run by JUnit to fail [21] due to test-order dependencies [1], [2], [10], [14], [22], [38]. Finding non-deterministic methods solely from the code is infeasible; one must reason about the specification itself to find if a method can be non-deterministic. This makes it inherently hard for any static or dynamic analysis technique to find such non-deterministic methods from one implementation.

We used two queries, based on (1) JavaDoc keywords and (2) return types. Specifically, the first query searches through JavaDoc for the following keywords that could indicate non-deterministic specifications: “order”, “deterministic”, and “not specified”. The second query searches for all public methods that return arrays. These queries produced many false positives, e.g., because not every method that mentions “order” is non-deterministic, and some methods that return arrays can return elements in a specified order. Our search is definitely not complete, and we leave as future work to develop better approaches to find non-deterministic specifications.
specification of one method specifies just a lower bound on the length of the returned array but not the precise length.

For class Object, it is well known that hashCode() is non-deterministic. The inner class HashMap$HashIterator does not have a specified iteration order and can return the map’s elements in any order; this inner class is exposed to the clients via some methods from Table I (keySet(), entrySet() and values()), so code that calls these methods can be ADINS. Moreover, HashMap is the underlying data structure for many other data structures, e.g., HashSet: we do not count separately the other non-deterministic methods, e.g., HashSet#iterator(), that could lead to ADINS code. However, changing one piece of code in HashMap is similar to the specification for iterating through ConcurrentHashMap is similar to the specification for iterating through HashMap. The File class has multiple list methods that return an array of files in a given directory; the specification allows these arrays to be in any order. The classes Class, Method, and Field provide several reflection methods that return arrays of elements, e.g., an array of all methods in a class or an array of all annotations on a field; the specifications for most of these methods allow these arrays to be in any order. The classes in the package java.text return arrays of available locales and zone strings which can be in any order. Finally, the DateFormatsSymbols#getZoneStrings() method returns an array of arrays, each of which has length at least five; these arrays are indeed of length five in Java 7 but of length seven in Java 8.

We also briefly explored an option of automatically finding non-deterministic methods in the Java Standard Library. We attempted to automatically generate tests that could show a behavior difference between Java 7 and Java 8. To that end, we used Randoop [28] to generate tests. We first instructed Randoop to generate tests for a large number of classes in the Java Standard Library on Java 8 and then ran the generated tests (that still compile) on Java 7. However, the tests (and assertions) that Randoop generated were unable to detect any changes in the behavior of the two Java versions. Even focusing Randoop on only one class, HashMap, did not generate (after one hour) a single test for Java 8 that would fail when run on Java 7. The reason is that the search space for HashMap is large, with 29 methods, and only a tiny ratio of method sequences in that space can show the difference between the two Java versions. In the end, we were able to generate tests that can reveal differences between Java 8 and Java 7 only after manually focusing Randoop to only four methods in the HashMap.

B. Non-deterministic Models

We developed models to explore potential non-determinism allowed by the specifications of the identified methods. NONDEX has a model for each non-deterministic method, and each model has up to four different levels of non-determinism: FULL, ID, EQ, and ONE.

For non-deterministic methods of the permute kind, the FULL level is the most non-deterministic and each different method invocation on an unchanged object will return a different order. The ID level preserves the same order for objects that have the same identity (unchanged objects) but can return different orders for the same object if the object is ever modified. The EQ level preserves the order for objects that are equal (not necessarily the same object) but can change the orders of non-equal objects. The ONE level changes the order of an object only when it is first accessed (so it can have a different order) but this order does not change in the rest of the run. Table II shows which of the assertions in Figure 2 (referred to by line numbers in the column headers) can fail under the four levels that NONDEX supports.

Either assertions 4 and 5 can fail on any of the levels, because all levels explore different orderings of the elements in the HashSet than the ones in the both assertions (recall that in a deterministic JVM, one of the assertions will always pass and one will always fail, whereas in our exploration the they can both pass, both fail or swap the order of pass/fail). Assertions 7 and 10 can only fail in FULL since they will only fail in levels that allow different orders on two successive invocations (including observer methods). Assertions 13 and 17 can fail in FULL and ID since these levels explore different orderings of objects based on their identity. Assertion 22 can fail in all levels except ID which would permute elements in the same way for both objects.

For non-deterministic methods of the random kind, when using the Object class, it should not be assumed that the hashCode() method returns a specific integer value. In particular, it should not be expected to return the same value across different runs. However, the returned value should be unique for an object in the same run. We model these potentially different values by randomizing the value returned by hashCode() on the initial invocation and then cache this value for future calls. For non-deterministic methods of the extend kind, we model the possibility that the lengths of arrays returned are increased non-deterministically on any invocation.

C. Implementations of Models

We implemented our NONDEX technique for the Java programming language by modifying a regular Java Virtual Machine, in particular, the OpenJDK JVM version b132, which corresponds to Java 8. We downloaded the publicly available OpenJDK code, which consists of the C/C++ code that implements the core virtual machine, and the Java code for the Java Standard Library. We changed the Java Standard Library to add non-deterministic models for all methods listed in Table I except for hashCode(), for which we modified the C++ implementation to return different values. For each of the

<table>
<thead>
<tr>
<th>Levels</th>
<th>Assertion 4</th>
<th>Assertion 5</th>
<th>Assertion 7</th>
<th>Assertion 10</th>
<th>Assertion 13</th>
<th>Assertion 17</th>
<th>Assertion 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ID</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EQ</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ONE</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table II: Levels that can fail (✓) assertions from Figure 2.
class HashMap {
    Node<K,V>[] table; // internal table of key-value pairs
    int modCount = ...; // stores modification count
    class Node<K,V> { ... } // stores a key-value pair ...
...
    class HashIterator { // inner class of HashMap
        Node<K,V> next; // next entry to return
        int index; // current in hash
        int expectedModCount; // for fast-fail
        int current; // current entry
        final boolean original_hasNext() {
            return null != next; // original code
        }
        final Node<K,V> original_nextNode() {
            // original code, advances "index" and "next"
            ...
        }
        final void original_remove() {
            // original code, can modify the entire "table"
        }
        ...
    }
    HashIterator() {
        expectedModCount = modCount;
        Node<K,V>[] t = table;
        current = next = null;
        index = 0;
    }
}

class Node<K,V> { ...

class NonDex { // NonDex extension
    static int level; // FULL, ID, EQ, or ONE
    int seed = ...;
    static Random full = new Random(seed);
    public static <T> List<T> shuffle(List<T> l, int v) {
        int size = l.size();
        Random rand = (level == FULL) ? full : // Full
            (level == ID) ? new Random(seed + v) : // Same object
            (level == EQ) ? new Random(seed + v) : // Equal object
            (level == ONE) ? new Random(seed) : // Once
            null;
        int i = 0; i < size - 1; i++) {
            int s = rand.nextInt(i, size);
            current = l.get(s);
            l.set(i, s);
        }
        return l;
    }
}

Fig. 4. Implementation of exploration

<table>
<thead>
<tr>
<th>PID</th>
<th>Project</th>
<th>SHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>EsotericSoftware/reflectasm</td>
<td>4555612e</td>
</tr>
<tr>
<td>P2</td>
<td>EsotericSoftware/yamlbeans</td>
<td>2ccfbd9d</td>
</tr>
<tr>
<td>P3</td>
<td>JodaOrg/joda-time</td>
<td>07002501</td>
</tr>
<tr>
<td>P4</td>
<td>OryxProject/oryx</td>
<td>833c3feaa</td>
</tr>
<tr>
<td>P5</td>
<td>Thomas-S-B/visualee</td>
<td>410a80f0</td>
</tr>
<tr>
<td>P6</td>
<td>apache/commons-cli</td>
<td>a0dcd6a0</td>
</tr>
<tr>
<td>P7</td>
<td>apache/commons-lang</td>
<td>fab946a1</td>
</tr>
<tr>
<td>P8</td>
<td>benas/easy-batch</td>
<td>4761ba5a</td>
</tr>
<tr>
<td>P9</td>
<td>bpsm/edn-java</td>
<td>c1d891d6</td>
</tr>
<tr>
<td>P10</td>
<td>caelum/vraptor</td>
<td>443e0fed</td>
</tr>
<tr>
<td>P11</td>
<td>fernandezpablo85/scribe-java</td>
<td>0311a435</td>
</tr>
<tr>
<td>P12</td>
<td>geosolutions-it/geoserver-manager</td>
<td>a4208d7d</td>
</tr>
<tr>
<td>P13</td>
<td>jknap/handlebars.java</td>
<td>83dd013a</td>
</tr>
<tr>
<td>P14</td>
<td>joel-costigliola/assertj-core</td>
<td>e86966e8</td>
</tr>
<tr>
<td>P15</td>
<td>jscep/jscode</td>
<td>a22cc25</td>
</tr>
<tr>
<td>P16</td>
<td>junit-team/junit</td>
<td>1d63100e</td>
</tr>
<tr>
<td>P17</td>
<td>ning/org-json</td>
<td>9be37018</td>
</tr>
<tr>
<td>P18</td>
<td>qos-ch/slf4j</td>
<td>52fcbb68</td>
</tr>
<tr>
<td>P19</td>
<td>sematext/ActionGenerator</td>
<td>10fa4a3e</td>
</tr>
<tr>
<td>P20</td>
<td>stickfigure/objectify</td>
<td>819eb72f</td>
</tr>
<tr>
<td>P21</td>
<td>versly/wsdoc</td>
<td>894808c5d</td>
</tr>
</tbody>
</table>

Table III
21 PROJECTS (OUT OF 195) WITH AT LEAST ONE FLAKY TEST

Fig. 3. Non-deterministic model for HashMap

methods whose output permutes, we call NONDEX and shuffle on the returned value in the library code.

Figure 3 shows the model we use for exploring different orderings when iterating over a HashMap object. It is based on the inner class HashIterator. We kept the original code and renamed its methods with the prefix original_. The constructor, starting at line 33, computes the order that the original code would have normally returned, applies a shuffle depending on the NONDEX level, and stores the resulting order in an Iterator object. The next method returns the elements in the shuffled order. The hasNext method is based on the new order and delegates to the new Iterator object. The remove method just delegates to the original method that changes the table.

Figure 4 shows how NONDEX performs shuffling depending on the level. For FULL, NONDEX uses the same Random object to perform all shufflings, which means two consecutive shufflings of the same object can yield different orders,. For ID, NONDEX considers a value representing the identity of the object; for HashMap, this value is the sum of the identity hash code and the modCount field counting the number of modifications, which means that for the same object with the same modCount, NONDEX uses a fresh Random object with the same seed. Similarly, for EQ, NONDEX considers the value-based hash code of the object to produce a new Random object. For ONE, NONDEX always creates a fresh Random object using the same seed.

IV. EVALUATION

We evaluated our NONDEX technique on 195 open-source projects and 72 student submissions from a software-engineering course. Section IV-A describes our experiments with the open-source projects, and Section IV-B describes our experiments with the student code.
A. Experiments on Open-Source Projects

We evaluated NonDex on 195 open-source projects. We selected these projects and their specific revisions from our previous studies with open-source projects [4], [23], [33]. All these projects are from GitHub [8], use Maven to build [26], and compile successfully using Java 8. For each project, we first ran NonDex with 10 randomly generated seeds, using the FULL level. If any test failed with these 10 seeds, we examined it to determine what caused the failure. (As a side note, we found that manually inspecting these failures was rather challenging, and we leave it as future work to automate debugging test failures due to ADINS code.)

We detected 60 flaky tests in the 21 projects listed in Table III. We tabulate a short ID for ease of reference, the project name, and the project revision on which we ran NonDex. For each project with a flaky test, we then reran that project’s tests using NonDex with 100 randomly generated seeds, using all non-deterministic levels. We obtained the number of times each flaky test fails out of the 100 seeds.

Table IV shows a partial list of the 60 tests that we examined. We tabulate the PID (from Table III), the name of the test class and its flaky test method, the number of failures detected for the each of the four levels, and the most likely non-deterministic method that causes the failures. The apache/commons-lang project has 14 tests similar to MultilineRecursiveToStringStyleTest#boolArray, and the caelum/vraptor project has 13 tests similar to XStreamSerializerTest#shouldSerializeCollection, so the two table rows show the total number of failures for each level across all 14 and 13 tests, respectively. Our evaluation started on older revisions of these projects, and three tests (GenericTest#testWrite, TestDateTimeZone#testGetShortName, and TagTypeTest#collectSectionAndVars) are already fixed on the current revisions of their respective projects.

Running NonDex using the FULL level may introduce too much non-determinism, and one might initially consider some detected flaky tests to be false alarms. However, Table IV shows that only six flaky tests (FieldAccessTest#testIndexSetAndGet, TestDateTimeZone#testGetShortName, TextUtilsTest#testJSONMap, PrintersTest#testPrettyPrinting, XStreamSerializerTest#shouldSerializeCollection, and OptionGroupTest#testToString) were false alarms. We therefore recommend using the FULL level for detecting flaky tests, as it is the least likely to introduce non-determinism in our experiments.

A partial list of the 60 tests that we examined is shown in Table IV. We tabulate the PID (from Table III), the name of the test class and its flaky test method, the number of failures detected for each of the four levels, and the most likely non-deterministic method that causes the failures. The apache/commons-lang project has 14 tests similar to MultilineRecursiveToStringStyleTest#boolArray, and the caelum/vraptor project has 13 tests similar to XStreamSerializerTest#shouldSerializeCollection, so the two table rows show the total number of failures for each level across all 14 and 13 tests, respectively. Our evaluation started on older revisions of these projects, and three tests (GenericTest#testWrite, TestDateTimeZone#testGetShortName, and TagTypeTest#collectSectionAndVars) are already fixed on the current revisions of their respective projects.
Even in this case, there is more than such a flaky test with 10 seeds, despite the chance of it failing 8%. The FULL level is only the expected probability of a flaky test failing for a seed in the test. In the most

... assertEquals("[--foo Foo, --bar Bar]", gl.toString());
9 )
10 )
11 )
12 )
13 public class OptionGroupTest ...
14 Map<String, Option> om = new HashMap<String, Option>();
15 public OptionOption addOption(Option option) {
16 om.put(option.getKey(), option);
17 return this;
18 }
19 public String toString() {
20 StringBuilder buff = new StringBuilder();
21 Iterator<Option> iter = getOptions().iterator();
22 while (iter.hasNext()) {
23 buff.append("[");
24 /* ... populate buff with the values in iter ... */
25 return buff.toString();
26 }
27 }
28 }
29 
Fig. 5. Flaky test from apache/commons-cli

AndGet, OptionGroupTest#toString, FieldUtilTest#testGetAllFields, FieldUtilTest#testGetAllFieldList, MethodSorterTest#testJvmMethodSorter, and EventLoggerTest#testEventLogger) fail sometimes for the FULL level but not fail at all for any of the 100 randomly generated seeds for any of the other levels. The remaining 54 flaky tests are also detected by the other levels, suggesting that these are not false alarms.

For each flaky test, the table shows the number of seeds/runs on which it fails. For most flaky tests, the number of seeds is fairly high, with only 8–14 flaky tests failing for less than 50 seeds for each level (not counting flaky tests that have 0 failures for a given level), and only two of those tests fail less than 30 times for each level. These high numbers suggest that it is likely that a flaky test can be detected by running NONDEX with just a few seeds. Assume that the actual probability of a flaky test failing for a seed is equal to the percentage of seeds that fail out of the 100 seeds that were run. For example, if the probability of a flaky test failing for a seed is 30%, then the probability of the flaky test not failing for 10 different, independent seeds is $\left(1 - 0.3\right)^{10} = 0.028$; in other words, there is a less than 3% chance of NONDEX missing to detect that flaky test running with 10 seeds. In the most extreme case we detected, in the Thomas-S-B/visualue project, the expected probability of a flaky test failing for a seed in the FULL level is only 8%, so the chance of NONDEX missing this flaky test running with 10 seeds is $\left(1 - 0.08\right)^{10} = 0.434$. Even in this case, there is more than 50% chance of detecting such a flaky test with 10 seeds, despite the chance of it failing for any one seed being rather low.

In summary, a developer using NONDEX to detect flaky tests may not need to run with many seeds and can still have some confidence that NONDEX does not miss to detect any flaky tests. Therefore, we recommend that NONDEX by default be run for 10 seeds while increasing the level of non-determinism, from ONE to FULL. Because the common threats to validity apply to our study, our results may not generalize to other projects or flaky tests. Of particular concern is that our experiments could have missed some flaky tests even in the projects that we ran with 10 seeds. If some test fails infrequently, it may be missed; there might be many such tests that NONDEX missed, so we could not have even studied them in more detail. In the future, we plan to evaluate more systematic exploration to check whether this is indeed the case.

We next discuss in more detail three flaky tests detected by NONDEX in open-source projects.

1) Overly Non-deterministic Level: A case where the FULL level detects a flaky test that is never detected for any other level is OptionGroupTest#toString from the apache/commons-cli project. Figure 5 shows that flaky test. Lines 4 and 5 add some options to the OptionGroup gl. OptionGroup stores options in a HashMap (line 16), and its toString() method (lines 19–27) iterates over this map. The developer realized that the iteration order over the HashMap is not guaranteed, so lines 6 and 7 check that the result of calling toString() on gl is either of the two hard-coded strings. However, toString() is invoked twice, and in the FULL level, NONDEX can reshuffle the order differently for the two invocations, causing the assertion to potentially fail. The developer made a reasonable assumption that calling toString() on the same, unchanged object twice returns the same string both times; we see that the other levels of NONDEX never flag this test as flaky. Nevertheless, the test could be still changed to call toString() only once and then to assert that it returns one of the two possible values.

2) Example New Flaky Test: We detected 57 flaky tests that are not fixed on the current revision of the projects, and Figure 6 shows one such flaky test, MapUtilTest#shouldPrettyPrintMap from the fernandezpablo85/scribe-java project. The test (lines 3–5) makes and populates a HashMap and then compares the result of calling MapUtil#toString() with a hard-coded string (lines 6–8). However, MapUtil#toString() calls entrySet() on
public class DefaultNameProvider implements NameProvider {
    public String getName(...) {
        String[] nameSet = getNameSet(...);
        return nameSet[0];
    }

    private synchronized String[] getNameSet(...) {
        String[][] z = DateTimeUtils.getDateFormatSymbols(...), getZoneStrings();
        String[] setEn = null;
        ...
        for (String[] s : z) {
            if (s != null && s.length == 5 && id.equals(s[0])) {
                setEn = s;
                break;
            }
        }
        return setEn;
    }

    public class TestDateTimeZone extends TestCase {
        public void testGetShortName() {
            DateTimeZone zone = DateTimeZone.forID(...);
            assertEquals("BST", zone.getShortName(...));
        }
    }

    public String getShortName(...) {
        NameProvider np = getNameProvider();
        NameProvider getDefaultNameProvider() {
            nameProvider = ((DefaultNameProvider) np).getShortName(...);
        }
        ...
        return name;
    }
}

Fig. 7. Flaky test from JodaOrg/joda-time

its input Map, and the order of iteration is not fixed, so the assertion on lines 6–8 can sometimes fail. More precisely, it fails in all but one of the 41 orderings, i.e., in about 96% of cases, as also obtained in our experiments.

3) Example Fixed Flaky Test: We next describe a flaky test that NONDEX detected when run on an older revision of the JodaOrg/joda-time project; the test has been fixed since then. Figure 7 shows TestDateTimeZone#testGetShortName and the relevant portions of the code under test. The call to getShortName() on line 23 eventually leads to a call to the DefaultNameProvider#getNameSet() method defined on lines 6–17. The problem is the guard condition, s.length == 5 on line 11. In Java 7, the call to DateUtils.getDateFormatSymbols() on line 7 returned each array element of z of exactly length five. However, the specification of that method only guarantees that each element of z has length of at least five. In fact, in Java 8, the implementation changed such that each array element has length exactly seven, which still satisfies the specification but is different from what was the case in Java 7. This change in the implementation revealed the developer’s reliance on the length of the elements of z. NONDEX was able to detect this on an older revision of the code, and the developers have since fixed this problem by changing checks such as the one shown on line 11 to be s.length >= 5 instead.

B. Experiments on Student Code

We also evaluated NONDEX on 72 student submissions for a programming assignment. We first describe the assignment that the students were supposed to do. We then describe how we set up our experiments for the student submissions. We finally describe high-level results concerning our findings of running NONDEX on the student submissions.

1) Assignment: The assignment asked the students to create a simple library-management application. The students were expected to write both code that implements such an application and unit tests using JUnit [20] to test the different components of the application.

The teaching staff provided the students some skeleton code outlining the basic expected components of the application. The application should represent a library containing books which can be organized into collections. The Book class represents a book and has only two fields, a title and an author, both represented by String objects. This Book class extends the abstract class Element. The Collection class represents a collection of such Elements that are stored in a List. Furthermore, the Collection class also extends Element, so a Collection is allowed to contain other Collection objects, creating a hierarchy that illustrates the composite design pattern [7]. Finally, at the top level, there is a Library class that can hold a List of Collection objects.

Students were expected to implement several methods and constructors for each of these classes. We discuss those that are most relevant for this study. For both Book and Collection, students must implement a method toString() that returns String representations of objects of those classes. Given such a string representation, students must implement a constructor for Book that takes the string representation and constructs the corresponding Book object. For Collection, students must similarly implement a static method restoreCollection that takes a string representation of a Collection and constructs the corresponding Collection object. For Library, students must implement (1) the constructor that takes a file containing string representations of a sequence of Collection objects and constructs the corresponding Library and (2) the method saveLibraryToFile() that writes out the Library to a file.

Along with the skeleton code and implementation requirements, the teaching staff made further restrictions and suggestions. First, the students’ code must build successfully on a common platform used by the entire class. This platform uses OpenJDK Java 7, so students’ code must also compile to Java 7 bytecode and run successfully using the OpenJDK Java 7 JVM. Students must also write tests for each of the

1This library-management assignment was first created three years ago and has been minimally updated by different teaching staff members over the years; this year’s iteration of the assignment was updated by two teaching assistants who were not involved in this study.
three classes they implement, with at least nine tests for the entire application. Finally, the staff strongly encouraged the students to use some third-party library to handle the pretty-printing/parsing of objects to/from strings, as the library can potentially have complex structures involving deeply nested Collection objects. However, the staff did not restrict the students to a specific third-party library, so the students chose whatever library they felt comfortable with. Many used various libraries for JSON or XML.

2) Experimental Setup: For our evaluation on student code, we started from the 89 submissions that built successfully (both compiled and had all tests pass) on the common platform that uses OpenJDK Java 7. With these 89 submissions, we ran the tests on another platform that is exactly the same as the platform provided to the students, except this other platform uses OpenJDK Java 8 instead. By running the students’ tests against their own code on a platform using Java 8, we already detected some students’ tests to be flaky as they assumed specific behavior of the libraries (either the Java Standard Library or the third-party libraries used), and most likely failing due to the presence of ADINS code.

Running the students’ tests in this Java 8 environment, we found 17 submissions that fail. In fact, in the past, running in multiple environments (e.g., on Linux virtual machines and on Mac and Windows laptops from teaching assistants) was the only approach that we could use to detect (some) flaky tests. Using NONDEX, we can have a more thorough detection of flaky tests; even if some tests pass on both Java 7 and Java 8, it does not imply they do not contain any ADINS code that could fail on some future Java 9 (or even on Java 8 on another OS or by another JVM provider, say, IBM). We therefore focus the rest of our evaluation on the remaining 72 submissions.

3) Results: We ran the student submissions using our NONDEX tool in all four non-deterministic levels and for 100 randomly generated seeds. (The tests from student submissions run much faster than the open-source projects, so we could immediately use 100 seeds.) NONDEX detected 34 student submissions where exploring all non-deterministic choices, we used Java PathFinder (JPF) [17], [34]. JPF provides a specialized JVM, implemented in Java, that can explore all non-deterministic choices. However, JPF cannot handle all Java code out-of-the-box. In particular, it cannot handle code that depends on native methods, such as those in Gson or XStream that students used.

To use JPF, we changed our model for HashMap to not make some random choices but rather systematically explore all choices; specifically, considering Figure 4, on line 13, we replaced rand.nextInt(i, size) with Verify.getInt(i, size - 1), where the JPF method Verify.getInt explores all possible values between the bounds i and size - 1, inclusive. Hence, JPF shuffles each List to explore all permutations. Because JPF can systematically explore all choices until either the test fails or all possible permutations execute and the test does not fail, JPF will not miss any test that depends on some ordering for the elements in a HashMap. Because JPF cannot handle most file system operations, we could not run the tests for the Library class which need to read/write some file from/to disk. Even without running the tests for Library, there were still many cases where JPF throws some internal exception because of functionality that is not yet implemented in JPF. As a result, we ignored the 22 submissions that do not work with JPF and only ran the tests for Book and Collection.

Using JPF, we detected 22 submissions where exploring permutations on HashMap iterator order causes a test to fail. In total, JPF detected 51 flaky tests. We compared these results with those obtained from running NONDEX on our modified JVM, using only the non-deterministic methods in HashMap, and running only tests for Book and Collection. We found that NONDEX detected all tests detected by JPF, increasing confidence that 100 seeds suffice for student submissions.

4) Discussion: In the 17 cases where the students’ tests fail just by switching from Java 7 to Java 8, the flaky tests check the functionality of the methods that get the string representation of a Book or a Collection object. The tests generally construct some Book or Collection objects and assert that the return of the method that gets the string representation matches some hard-coded string value. In all but one of these submissions, students either directly use a Java HashMap as part of their implementation for constructing a string representation, or they use a third-party library (e.g., JSON in Java [18] or JSON.Simple [19]) where the serialization is backed by a Java ObjectMapper. The assertions against the hard-coded strings succeed in Java 7 because the order

<table>
<thead>
<tr>
<th>Flaky Tests Found</th>
<th>FULL</th>
<th>ID</th>
<th>EQ</th>
<th>ONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>110</td>
<td>88</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>Total Failures</td>
<td>8159</td>
<td>6785</td>
<td>2031</td>
<td>1827</td>
</tr>
<tr>
<td>Min Failures</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max Failures</td>
<td>100</td>
<td>100</td>
<td>81</td>
<td>78</td>
</tr>
</tbody>
</table>
remains consistent across different runs of the JVM, but in Java 8, the underlying implementation of HashMap changed such that the iteration order can differ from that of Java 7. The one remaining failing submission uses an XML serialization library (XStream [37]) to construct a string representation of a Collection object, but the order of the declared fields for a class is also not guaranteed, so the comparison with a hard-coded string value here once again fails in this later version of Java. In summary, all these 17 submissions have ADINS code and fail due to relying on some assumed order that is not guaranteed to hold.

In the student submissions that do not fail on Java 8, NONDEX detected more flaky tests that fail due to the non-determinism in the ordering provided by the iterator for a HashMap. As with the tests that fail on Java 8, these flaky tests generally construct Book and Collection objects and assert their string representation to be equal to a hard-coded string. Similar to some cases in the open-source projects, some failures are due to “too much” non-determinism in the orderings, e.g., when a test calls getStringRepresentation() on an object and then compares the string against another call of getStringRepresentation() of an equal object rather than asserting the string to be the same as a hard-coded string. In such a case, the FULL or ID level would shuffle both calls to getStringRepresentation() and potentially end up failing the assertion where the other two levels do not fail. Moreover, the FULL level also detects as flaky some cases that depend on the field ordering, which other levels never detect.

V. RELATED WORK

Detecting problems due to wrong assumptions that developers make about specifications and implementations has been explored in other domains. For example, Jin et al. [15] reported how wrong assumptions about code can lead to performance bugs, in particular, they find the second most common reason for the introduction of performance bugs to be that “developers misunderstand the performance feature of certain functions”. NONDEX does not target performance bugs but helps detect another class of bugs that are due to specification misunderstanding. As another example, from a security perspective, Wang et al. [35] propose a technique to analyze implicit assumptions that are necessary for the secure use of libraries. Their work involves building models of methods which are then used to find bugs in software that fail to meet these implicit assumptions, finding serious security vulnerabilities in the process. Their techniques are mostly static, while NONDEX uses a dynamic, randomized exploration of methods with non-deterministic specifications.

Randomness has been applied in different contexts to detect bugs, with many of these applications for concurrent code. For example, Eytani et al. [5] developed a tool that monitors shared variable accesses and applies random context switching when shared variables are accessed in order to trigger bugs in concurrent code; Parizek and Kalibera [29] used an abstract environment in software model checkers that randomly selects sequence of method calls in each thread to detect bugs in concurrent programs; and Joshi et al. [16] applied randomness in thread scheduling to create resource deadlocks in multi-threaded programs. Moreover, JPF can also control thread schedules to potentially explore all paths in the code [34]. In contrast, NONDEX focuses on sequential code and exploration of non-deterministic specifications.

Non-determinism has been also studied for various other domains. For example, for map-reduce programs, Xiao et al. [36] studied non-determinism that arises due to non-commutative reducers and found many bugs due to non-commutative reducers that make assumptions on the order of input data rows. For GUI code, Memon and Cohen [27] showed various factors that may cause non-determinism and hence impact the results of analyses and experiments based on GUI software. For state machines, testing conformance of deterministic implementations against non-deterministic specifications has a long history [13], [30]–[32]. More recently, Cook and Koskinen [3] aim to find restrictions on non-deterministic value-choices using a CEGAR loop; they apply their technique to examples drawn from real code. NONDEX explores non-determinism in the context of abstract data-type specifications using concrete exploration of real code.

VI. CONCLUSIONS AND FUTURE WORK

Non-deterministic specifications are good because they allow implementers to provide various implementations. However, non-deterministic specifications are bad because they can result in seemingly random failures. In particular, ADINS code that assumes a deterministic implementation of non-deterministic specification is susceptible to failures that arise from changing implementations. Tests that depend on ADINS code can become flaky tests that seemingly non-deterministically pass or fail. We proposed a novel NONDEX technique to detect flaky tests due to ADINS code. NONDEX detected many flaky tests in both larger, open-source projects and small-sized student code submissions.

In the future, we plan to investigate how to automate debugging of failures that NONDEX reports. While it is good that NONDEX detected many flaky tests, manually investigating them turned out to be hard. Interestingly, we found that only 7 of the 31 non-deterministic models were the likely causes for all detected flaky tests. Considering our experience with Java 7 and Java 8, we also plan to study advanced approaches to find behavior differences between different JVM implementations, e.g., using automated test generation that previously revealed differences in IDEs, Java compilers, and JPF [9]. Finally, to help developers avoid misunderstandings of non-determinism, we envision that code could have some deterministic annotations, e.g., @Unordered to specify methods that provide no guarantee on the order of elements. Overall, ADINS code seems to open an interesting new line of research.

Acknowledgments: We thank Lamyaa Eloussi, Wajih Ul Hassan, Nicholas Lu, Xinyue Xu, and Tiffany Yang for help in this project. This research was partially supported by the NSF Grant Nos. CCF-1012759, CCF-1409423, CCF-1421503, and CCF-1439957.
REFERENCES