Evaluating NONDEX for Modern Java Ecosystem

Kaiyao Ke
University of Illinois Urbana-Champaign
Urbana, IL, USA
kaiyaok2@illinois.edu

Darko Marinov
University of Illinois Urbana-Champaign
Urbana, IL, USA
marinov@illinois.edu

***Abstract*—NONDEX is a testing approach designed to unveil implementation-dependent (ID) flaky tests stemming from incorrectly relying on a deterministic implementation of a Java API with an underdetermined specification, e.g., iterating over elements of a `HashSet` object. Since the original NONDEX work was published in 2016, we have enhanced the tool functionality and expanded its integration with recent Java versions and build tools like Maven and Gradle. This evolution enables NONDEX to analyze a broader range of large, open-source Java projects.**

This paper investigates our updated NONDEX on modern Java projects. We identified 734 ID flaky tests in 31 Maven projects and 267 ID flaky tests in 25 Gradle projects. Comparing these findings to prior work, this study highlights an increase for a modern Java project to contain some ID flaky test(s). We also studied the propagation of ID flakiness through project dependencies and fixed a key non-determinism issue in the Gradle build system itself. Our study emphasizes the importance of proactively employing NONDEX to detect and fix flaky tests, preventing potential disruptions in ongoing and future projects. We put all our results at https://github.com/NonDexFTW/NonDex-Experiments.

I. INTRODUCTION

Flaky tests [1], [2], [3], which can non-deterministically pass or fail even for the same code under test, are an important problem in software development because they give misleading signals to the developers about their code changes. A test that passes before some code changes but fails after the code changes may not indicate a bug in the code changes themselves but can be due to test flakiness. Research on flaky tests has proposed several automated techniques to detect, fix, and mitigate flaky tests; e.g., Parry et al. [2] present an extensive survey of prior work on flaky tests.

NONDEX [4], [5] is a previously proposed approach for detecting a category of flaky tests, in particular *implementation-dependent* (ID) tests that rely on the particular implementation of methods with underdetermined specifications. For example, the standard Java library includes methods for iterating over set or map objects and a method for getting the list of fields for a given class. The specifications for these methods allow returning the values in different orders. Most, but not all, implementations of these methods are deterministic and return the values in the same order. A test that relies on a specific deterministic implementation, e.g., expecting that a set with values `1` and `2` must be printed as `[1, 2]`, would fail when the implementation changes.

NONDEX proactively detects flaky tests resulting from such erroneous assumptions about specifications. To explore whether a test could fail for changed implementations, NONDEX randomly permutes the returned values within the allowed specifications. Developers are typically cognizant of the detrimental impact posed by ID flaky tests and demonstrate a willingness to address them. According to the International Dataset of Flaky Tests (IDoFT) [6], [7], a total of 1,264 fixes for ID tests in open-source Maven projects have been accepted, while only 99 were rejected.

Prior work [5] adopted the definition that a specification is *underdetermined* [8] if it allows multiple implementations to produce different results for the same input, and defined an underdetermined API as an API with such a specification. For instance, the `iterator()` method of `java.util.HashSet` exemplifies an underdetermined API, as its Javadoc [9] specifies that its returned elements do not preserve order. Some code may erroneously assume deterministic order based on a specific implementation, e.g., a particular JDK version, is preserved across all JDK versions. Such assumption can result in creating ID tests that exhibit flakiness across different JVMs or Java versions [10]. The initial development of NONDEX [4] aimed to proactively identify these incorrect assumptions on underdetermined APIs by randomly exploring various allowed behaviors during test execution. We refer to the NONDEX process of permuting orders as "shuffling" for brevity.

The original NONDEX tool [5] provided several features but also had some limitations. It included (1) an instrumentation engine to modify the standard Java library APIs and allow different results from distinct implementations, (2) a runner layer using specified seeds and modes to support test execution using APIs under order modification, (3) a detector running tests both with and without NONDEX shuffling to detect flakiness, and (4) a debugger reporting the single invocation where NONDEX shuffling led to test failure. It also supported command-line invocation and Maven integration. However, the limitations included: (1) supporting only Java version 8, (2) not supporting the Gradle build system, and (3) offering a rather limited support for debugging cases that depend on multiple underdetermined specifications.

This paper summarizes key improvements to the NONDEX tool that enabled us to evaluate NONDEX on more modern Java projects. Since 2016, the NONDEX tool has been expanded in three major aspects: (1) extending support for more Java versions, in fact any version between 8 and 21, especially handling the new Java Platform Module System introduced since Java version 9 [11]; (2) implementation of the NONDEX Gradle plugin, supporting the increasing

2025 IEEE/ACM International Flaky Tests Workshop (FTW) | 979-8-3315-0231-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/FTW66604.2025.00011

979-8-3315-0231-7/25/$31.00 ©2025 IEEE
DOI 10.1109/FTW66604.2025.00011

popularity of the Gradle build tool [12]; and (3) enhancing NONDEX debugging functionality, including an extension for the NONDEX debugger to report all invocations leading to test failures. All these extensions are publicly available in the NONDEX repository [13] on GitHub. These extensions enable running NONDEX on a much broader set of Java projects, enabling us to perform a large study on modern Java projects.

In brief, this paper makes three contributions:

1) Substantial enhancements to the flaky test detection tool NONDEX, including support for multiple Java versions, compatibility with a more modern build tool, and improved debugging functionality;
2) An extensive evaluation of NONDEX on modern Java projects, allowing us to compare the prevalence of projects with flaky tests to a previously reported result;
3) A detailed case study of how one particular cause of flakiness in Gradle propagates through the intricate web of project dependencies.

Improvements to NONDEX: We enhanced NONDEX to improve its functionality and usability. The tool now supports Java versions 8 through 21, with key updates to accommodate changes introduced by the Java Platform Module System (JPMS) in Java 9, including the use of the JRT filesystem for runtime class instrumentation and a custom logger to replace the now-inaccessible `java.util.logging` class. A new Gradle plugin integrates NONDEX into Gradle projects, providing same support for detecting flaky tests and debugging buggy API invocations. NONDEX's debugging capabilities have also been expanded to report all invocations causing flakiness. We also introduced a configurable option to control the number of un-shuffled runs before applying NONDEX shuffling, improving the likelihood of correct distinction between ID and other types of flaky tests.

Extensive Evaluation: Given the prevalence of Maven and Gradle builds in Java projects, we conducted an extensive analysis of NONDEX on hundreds of open-source projects utilizing each build tool. Specifically, we ran NONDEX on 125 Maven projects (with 2,374 Maven modules and 108,008 passing tests) and on 121 Gradle projects (with 860 Gradle subprojects and 103,947 passing tests). Our runs detected 734 ID flaky tests across 73 Maven modules in 31 projects and 267 ID flaky tests across 55 Gradle subprojects in 25 projects. In particular, 28.80% Maven projects we examined contain at least one ID test. Compared to prior work [4], which detected flaky tests in only 10.77% of all Maven projects examined, we observed a substantial increase in the likelihood of projects containing some flaky test(s), presumably attributable to the growing complexity of software projects and reliance on external libraries over the years.

Case Study: As a case study, we identified how a single false assumption regarding an underdetermined API could propagate into a multitude of flaky tests across various projects. We performed a detailed examination of 626 flaky tests reported by NONDEX in 69 smaller-scaled Gradle projects (which are distinct from the 121 large-scale Gradle projects used in our previous experiments), and we found that 109/626 (17.4%) inspected tests, spanning 14/69 (20.3%) of Gradle projects, all resulted from just one false assumption about `java.util.HashSet.iterator()` in the core Gradle source code itself. We opened a pull request (PR) to fix this bug in Gradle [14]. Our recently accepted PR not only fixes the tests we previously identified but also prevents related tests in any additional projects we did not even run. Instances such as this Gradle bug serve as an indicator that non-determinism and flakiness can propagate widely within the intricate web of project dependencies.

II. OVERVIEW OF THE ORIGINAL NONDEX TECHNIQUE

Shi et al. [4] introduced the original NONDEX technique in 2016. The main goal of NONDEX is to provide a way to identify implementation-dependent (ID) flaky tests by explicitly modeling non-deterministic behaviors allowed by the Java Standard Library specifications and intentionally shuffling various allowed orders. NONDEX initially referred to this category of test flakiness as "assumption of deterministic implementations of non-deterministic specifications (ADINS) [4]. Shi et al. identified methods in the Java Standard Library where non-deterministic behavior is permitted by the specification, even if the current underlying implementations are fully deterministic. Identifying such methods is challenging because the non-determinism arises from the specification itself rather than the specific implementation code.

To detect these methods, Shi et al. searched Javadoc keywords (e.g., "order", "deterministic", and "not specified") and public methods returning arrays to identify where the order or return structure is underspecified. After identifying these methods, Shi et al. developed shufflers to simulate different levels of underspecified behavior. They categorized non-determinism into three main types: random (e.g., `Object#hashCode()`), permute (e.g., collections where the order of elements may vary), and extend (e.g., arrays with flexible length constraints). The NONDEX tool then shuffles the output of these methods, exploring possible behavior by modifying the standard Java library itself to return different orders as the output of these methods with respect to a given random seed. This technique exposes flakiness by mimicking scenarios where tests fail due to incorrect assumptions about deterministic behavior in underdetermined specifications.

The main workflow for using NONDEX involves creating and utilizing instrumented versions of standard Java libraries. First, when NONDEX is installed, it generates an instrumented JAR with the injection of new behaviors into the specific Java Standard Library methods. The NONDEX tool also prebuilds a NONDEX-common JAR to control non-determinism (e.g, which APIs to shuffle or not shuffle). After the setup, users can run their application with NONDEX by including the instrumented JARs on the Java boot classpath, enabling NONDEX to identify potential flaky tests that rely on deterministic assumptions in inherently underdetermined methods. A later tool paper [5] provided the initial NONDEX plugin for the widely used Maven build system.

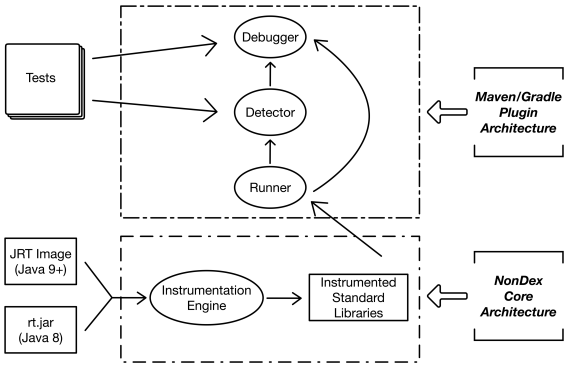

Fig. 1. Architecture of the latest NONDEX version 2.2.1

III. IMPROVEMENTS OF NONDEX

This section outlines our most notable improvements of the NONDEX tool since 2016. Figure 1 shows the architecture of the latest version of the NONDEX tool. The tool is primarily used through the NONDEX Maven and Gradle plugins; the two have distinct architectures for runners, detectors, and debuggers, but they share a common instrumentation engine that currently accommodates Java versions from 8 to 21.

A. Support for Multiple Java Versions

The initially released NONDEX version [5] was 1.0.0, and it has since evolved to version 2.2.1, reflecting substantial changes, particularly in its integration with various versions of the Java programming language. In fact, the versioning scheme of NONDEX deviates from the strict semantic versioning [15], because the last two digits signify the version of Java, e.g., 2.1 in 2.2.1 signify support for Java 21, and the previous version 2.1.7 supported Java 17.

Notably, the key enhancement lies in the transition between Java 8 and 9. Java 9 introduced a novel Java Platform Module System (JPMS) [16]. (Note that the same term "module" is used by both Java 9 and Maven but for completely different concepts.) This change required a substantial redesign of the NONDEX instrumentation of the standard library code. A change in the implementation of `VM.class` in Java 9 includes loading standard libraries earlier (during VM initialization), necessitating postponing NONDEX instrumentation of the standard library to avoid exceptions during VM initialization when executed under Java 9+. A checker added in NONDEX ensures that the VM is booted before initializing NONDEX.

Additionally, the initial instrumentation engine of NONDEX [5] conceptually operated by modifying the `rt.jar` file that stored the classfiles of the standard Java library. However, Java 9+ versions no longer include `lib/rt.jar` within their JRE images. To adapt, NONDEX implements a check for the used Java version and instead of using `rt.jar` utilizes the JRT filesystem to extract runtime classes for instrumentation when NONDEX executes under Java 9+.

Furthermore, the initial version of the NONDEX tool utilized the `java.util.logging` class for logging, which became inaccessible from the `java.base` module in Java 9+ versions. In response, NONDEX now includes a custom logger for all NONDEX executions. The major changes for newer Java versions were accumulated in one pull request for NONDEXversion 2.1.1 that added support for Java 11 [17]. Since then support has been extended to Java 17 (NONDEX version 2.1.7) [18] and Java 21 (NONDEX version 2.2.1) [19].

B. Integration with Gradle

The latest version of NONDEX is seamlessly integrated in the testing process for Gradle builds through our new Gradle plugin, available from the Gradle Plugin Portal [20]. Developers using Gradle can incorporate NONDEX into their projects by adding NONDEX as a plugin to the top-level build configuration file (by default `build.gradle`) and applying it to subprojects if necessary. Our NONDEX Gradle plugin provides three tasks: `nondexTest` for detecting flaky tests, `nondexDebug` for outputting debugging help in the form of invocations leading to flakiness, and `nondexClean` for deleting directories and files generated by NONDEX. Both the detector and the debugger call the NONDEX runner that executes tests using the NONDEX-instrumented libraries. The NONDEX runner incorporates the NONDEX-instrumented standard libraries and the packaged `nondex-common.jar` supporting NONDEX configuration and logging into the `bootclasspath` of the JVM employed for Gradle testing.

C. Improved Functionality

Two major functionality improvements of NONDEX are the new ability of the debugger to report multiple invocations and a new option to control the number of clean test runs before NONDEX applies shuffling. Minor improvements include exemptions from shuffling small empty or singleton collections, optimized data retrieval from `HashMap` objects, and more.

1) Debugger Reports Multiple Invocations: In contrast to prior work [5], where the debugger performed binary search on all invocations of each test and identified a single invocation that could lead to failure under NONDEX shuffling, the current version searches for all such invocations via backtracking on the binary search and returns a `List` of invocations. This refinement is particularly beneficial in large-scale projects where tests often encompass more than one false assumption regarding underdetermined APIs. (In fact, the experience with the extended debugging helped us to later debug the issue in the Gradle core code, described in Section VI.) Before our new extension, the debugging process was slowed down by requiring multiple iterations to fix the multiple causes of non-determinism one-by-one, while rerunning NONDEX on partially fixed code versions.

2) Configurable Number of Runs Without NONDEX *Shuffling:* In its default configuration, NONDEX conducts one un-shuffled run followed by three NONDEX-shuffled runs for each test. The tool identifies and reports a test if it passes in the un-shuffled run but fails in any of the subsequent NONDEX-shuffled runs. These reruns also allow NONDEX to detect flaky tests that are not implementation-dependent (ID) and

whose root causes may not be some false assumptions on underdetermined APIs. For instance, a test that fails due to some timing or concurrency issues may fail with NONDEX, although the root cause is not NONDEX-shuffling.

In fact, we found that some flaky tests documented initially as ID tests in IDoFT [6], [7], because they were reported by NONDEX, are later relabelled through manual inspection. One example are non-idempotent tests that exhibit deterministic pass results in the initial run but fail in subsequent runs [21]. Running the NONDEX debugger on such non-idempotent tests is wasteful, as their execution unnecessarily prolongs the binary search without providing useful information for debugging ID behavior. Similarly, if NONDEX identifies tests that are flaky due to other non-determinism causes (present without NONDEX shuffling), such causes would disrupt the debugger's binary search algorithm. In response, we make NONDEX support a configurable number of clean runs without shuffling. For example, this feature was useful in our study when performing a sanity check on tests reported by the detector, aiding in the exclusion of tests unsuitable for further examination by the debugger.

IV. EXPERIMENTS

Our evaluation follows the experimental methodology similar to the initial studies of NONDEX [4], [5]. We first collected projects from GitHub. We then ran NONDEX to identify flaky tests and in particular likely ID tests. We finally analyzed these collected tests in more detail.

A. Project Selection

We conducted an evaluation of NONDEX on popular, large-scale, open-source projects hosted on GitHub. All the projects we selected use either Maven or Gradle for building. For both build systems, we employed the GitHub Search API to collect a list of Java-based projects. For Maven projects, we applied a filter to include only repositories with more than 1,500 stars. For Gradle projects, we set a lower star threshold of 200, because Maven, being older, generally has projects with higher star counts accumulated over a longer period.

The GitHub Search API returns a maximum of 1,000 results per query; we executed the query once, retrieving the top 1,000 most relevant repositories based on GitHub's ranking algorithm. From these, we further filtered repositories containing a `pom.xml` file for Maven or a `build.gradle` for Gradle. Additionally, we ensured that the latest commit occurred within the past 12 months to confirm active maintenance. This process yielded 418 Maven projects and 522 Gradle projects. From these projects, we randomly selected 207 Maven projects and 244 Gradle projects, and attempted to compile them.

We then excluded the projects that fail to compile on our Ubuntu 22.04 machine under either Java 17 or 21. We selected Java 17 and 21 for evaluating modern Java projects because these two Java versions are the latest Long-Term Support (LTS) and feature releases. For Maven, we encountered build failures in 82 out of 207 projects, while for Gradle, 123 out of 244 projects failed. Note that these build failures are *not* due to NONDEX—common causes of build failures include incompatible Java versions, network errors, or OS dependencies. Our final evaluation includes 125 Maven projects (with a total of 2,374 Maven modules) and 121 Gradle projects (with a total of 860 Gradle subprojects).

B. Flaky Test Collection

Our evaluation specifically targeted tests that passed in the clean run without NONDEX shuffling. We do not evaluate tests that fail in clean runs, although they could in theory be flaky if they pass in some run with NONDEX shuffling. The selected 125 Maven projects (with 2,374 Maven modules) had a total of 108,008 passing tests, and the selected 121 Gradle projects (with 860 Gradle subprojects) had a total of 103,947 passing tests. For each module (in Maven) or subproject (in Gradle), we initially ran tests without NONDEX shuffling and subsequently ran NONDEX with three different random seeds, as per the default NONDEX mode. We use the same random seeds for NONDEX shuffling for both Java 17 and 21. Our script recorded all flaky tests, identified as those passing in the clean run but failing in one of the runs shuffled by NONDEX.

Combining the results of Java 17 and 21, NONDEX reported flaky tests in 36 (of 125) Maven projects and 35 (of 121) Gradle projects. Because NONDEX reports all flaky tests that pass in the clean run but fail in one of the "shuffled" runs, such reported tests need not all be ID. For example, a test may pass or fail non-deterministically due to asynchronous wait [22] or the random seeds selected for random number generators [23].

To distinguish which reported tests are actually ID, we developed a semi-automated pipeline. First, we perform 50 reruns without shuffling any APIs to exclude non-deterministic [24] or non-idempotent-outcome [21] flaky tests. Then, the first author, with three years of experience with ID flaky tests, manually analyzed the remaining tests (e.g., checking if binary search of the debug phase converges and reviewing stacktraces) to determine if the test is likely ID. We merged our results into the IDoFT [6]. Generally, we verified that a relatively large fraction of studied projects has some (ID) flaky test(s), although in a few projects, NONDEX detected tests that are mostly due to other (non-ID) reasons.

C. Summary of Results

Table I presents a summary of our experimental results. We classify a project/module/subproject as "flaky" if it contains at least one flaky test reported by NONDEX, and we classify a project/module/subproject as "contains ID" if it contains at least one ID flaky test verified through the automatic reruns and our manual analysis. Notice that NONDEX reports a different number of flaky tests between Java 17 and Java 21, due to non-ID reasons, such as non-deterministic server timeouts. The natural non-determinism in flaky tests could lead to different results even for the same Java version. When inspecting reported tests, we found the exact same verified ID tests across both versions, likely because no substantial implementation changes in underdetermined APIs were made between Java versions 17 and 21.

TABLE I
SUMMARY RESULTS FOR MAVEN AND GRADLE PROJECTS

	Maven			Gradle		
	Total	Containing Flaky Tests	Containing ID Tests	Total	Containing Flaky Tests	Containing ID Tests
Projects	125	36 (28.80%)	31 (24.80%)	121	35 (28.93%)	25 (20.67%)
Modules / Subprojects	2,374	97 (4.09%)	73 (3.07%)	860	70 (8.14%)	55 (6.40%)

Tables II and III provide a more detailed breakdown, presenting the number of flaky tests alongside the total number of tests evaluated for each identified "flaky" Maven or Gradle project. We also document the short commit `SHA` at which the flaky tests are detected, facilitating the reproducibility of our results and the further study of test flakiness.

V. DISCUSSION

Our experiments on both Maven and Gradle projects indicate a potential increasing prevalence of open-source projects to contain some flaky tests than in the past. In contrast to prior work [4] which identified at least one flaky test (a total of 60 flaky tests) in merely 10.77% of the projects that were built and run, the results of our experiment show that 28.80% of Maven project and 28.93% of Gradle projects contain at least one flaky test. These results indicate a potential considerable increase in the likelihood of flaky test presence in modern large-scale projects. After careful inspection, we also find that 24.80% of Maven projects and 20.67% of Gradle projects contain at least one ID flaky test.

It is noteworthy that the percentage of "flaky" project parts (4.09% for Maven and 8.14% for Gradle)—modules (as per the Maven definition) or subprojects (as per the Gradle definition)—is substantially lower than that of "flaky" projects (28.80% for Maven and 28.93% for Gradle). This discrepancy can arise from the fact that tests are typically concentrated within a small subset of modules or subprojects in extensive multi-module or multi-subproject repositories. This discrepancy can also mean that the overall prevalence of flaky tests is *not* increasing for project parts/modules/subprojects but simply the projects themselves are growing, so they are more likely to have some flaky test somewhere in the overall test suite.

It is also likely that the number of ID flaky tests within a project may increase over time. In the prior work [5], the greatest number of ID flaky tests present in a project was just 8, while in our experiment, several individual projects exhibit more than 50 ID tests. For example, the Maven project `kiegroup/jbpm` contains 361 ID flaky tests distributed across 10 modules, indicating that over 10% of its total tests exhibit ID flakiness. As another example, while prior work [5] only found 1 ID test in an older version of `alibaba/druid`, our current experiment finds 12 ID flaky tests in a more recent version of `alibaba/druid`. The increase in the number of flaky tests—and in particular, ID tests—in projects could be attributed to the expansion of code size, more extensive testing logic, and the widespread adoption of parameterized testing.

```
public class PatternSpecFactory ... {
    private String[] previousDefaultExcludes;
    private ... getDefaultExcludeSpec(...) {
        String[] defaultExcludes = DirectoryScanner
            .getDefaultExcludes();
        if (defaultExcludeSpecCache.isEmpty()) {
            ...
        } else if (invalidChangeOfExcludes(
            defaultExcludes)) {
            failOnChangedDefaultExcludes(...);
            // throws "InvalidUserCodeException"
        }
    }
    private boolean invalidChangeOfExcludes(String
        [] defaultExcludes) {
        return !Arrays.equals(
            previousDefaultExcludes,
            defaultExcludes);
    }
}
```

Fig. 2. Snippet of the PatternSpecFactory class from the Gradle core

```
public class DirectoryScanner ... {
    private static final Set<String>
        defaultExcludes = new HashSet<>();
    public static String[] getDefaultExcludes() {
        synchronized (defaultExcludes) {
            return defaultExcludes.toArray(new
                String[0]);
        }
    }
}
```

Fig. 3. Snippet of the DirectoryScanner class from the Gradle core

However, it is important to note that the absolute number of ID flaky tests does not serve as a direct indicator of the number of false assumptions related to underdetermined Java APIs. The number of test failures may not be strictly proportional to the number of underlying code faults, because multiple test failures may be caused by the same fault. In fact, looking through the International Dataset of Flaky Tests (IDoFT) [6], where numerous contributors document fixes for flaky tests detected by recent versions of NONDEX, we can see many fixes that are just concise one-liners but rectify several flaky tests. For instance, several recent pull requests to large projects such as `wildfly/wildfly` [25] and `google/TestParameterInjector` [26] provide illustrations where approximately 50 NONDEX-detected flaky ID tests in a project stem from a single, straightforward false assumption concerning one underdetermined Java API.

TABLE II
DETAILED VIEW OF MAVEN PROJECTS (SORTED BY THE NUMBER OF FLAKY TESTS IN JAVA 17)

Project	Stars (2024-11-11)	Commit	Total Tests	Flaky (Java 17)	Flaky (Java 21)	ID
apache/pulsar	14,237	fdeb191	3,705	507	298	12
kiegroup/jbpm	1,647	f7e80e8	3,501	364	366	361
apache/tinkerpop	1,972	8bfdf50	29,559	321	321	97
swagger-api/swagger-core	7,386	5186247	649	82	82	82
spring-cloud/spring-cloud-config	1,963	3e423e5	1,019	58	62	0
apache/iotdb	5,612	f94f99a	3,281	58	58	54
apache/servicecomb-java-chassis	1,907	9ba66eb	1,975	33	32	32
NanoHttpd/nanohttpd	6,949	efb2ebf	224	17	17	17
eclipse-vertx/vert.x	14,313	67ba713	4,660	16	16	1
openjdk/jmh	2,218	cb3c3a9	1,031	13	1	0
alibaba/druid	27,972	ed00cab	6,438	12	12	12
knowm/XChange	3,874	5bd548a	1,069	9	9	8
INRIA/spoon	1,753	bd9b16f	4,264	8	8	8
tabulapdf/tabula-java	1,841	8bfa3ad	206	7	7	1
RipMeApp/ripme	3,722	4aec463	119	6	9	0
EsotericSoftware/kryo	6,198	c6d199b	610	6	6	6
apache/tika	2,512	589d02a	1,965	6	6	5
spring-projects/spring-data-redis	1,771	3e8cd16	2,738	6	6	6
karatelabs/karate	8,264	8d4511c	619	5	4	4
apache/incubator-kie-optaplanner	3,327	a586b25	4,993	5	6	3
spring-projects/spring-retry	2,174	e19f5d7	387	4	4	3
hneemann/Digital	4,421	34c9832	820	4	4	4
apache/atlas	1,835	d8cf1e3	415	4	4	4
mybatis-flex/mybatis-flex	1,913	d725995	91	3	3	3
primefaces/primefaces	1,808	0714442	412	2	2	2
networknt/light-4j	3,611	254099b	835	2	2	1
winder/Universal-G-Code-Sender	1,905	061670e	739	1	1	1
wildfly/wildfly	3,061	8f6efd8	1,902	1	1	1
ulisesbocchio/jasypt-spring-boot	2,908	fc0ef8f	50	1	0	0
techa03/goodsKill	1,948	5c99f0e	37	1	1	1
square/keywhiz	2,621	52e77fd	243	1	1	1
spring-projects/spring-data-elasticsearch	2,916	957fe05	525	1	1	1
quarkusio/quarkus-quickstarts	1,965	6d72b2d	236	1	1	1
javaee-samples/javaee7-samples	2,509	4a67b23	2	1	0	0
citerus/dddsample-core	5,012	8870097	129	1	1	1
atomashpolskiy/bt	2,421	6218108	296	1	1	1
Total	162,466	N/A	79,744	1,568	1,353	734

VI. CASE STUDY

We describe one important example that demonstrates how flakiness stemming from underdetermined APIs can permeate widely through the intricate web of project dependencies. Specifically, the Gradle's build policy disallows any modification to the list of excluded files during the build process. In Figure 2, the method `getDefaultExcludeSpec()` within `PatternSpecFactory.java` inspects whether the array of currently excluded files, obtained from `DirectoryScanner.getDefaultExcludes()`, matches the previously retrieved array returned from the same method. However, the `DirectoryScanner` class, depicted in Figure 3, implements its `defaultExcludes` collection as a `HashSet`. Subsequently, `getDefaultExcludes()` returns the elements of the `HashSet` collection as an array, which lacks deterministic order in its output as per the Javadoc specification [9]. Consequently, the comparison using `Arrays.equals()` is susceptible to failure even if the actual list of excluded files remains unchanged throughout the build. Our detailed examination of 626 flaky tests, reported by NONDEX in 69 smaller-scaled Gradle projects, found that 109 of these tests, spanning 14 projects, exhibit flakiness due to invoking the Gradle Runner.

Our accepted pull request [14] to the Gradle project fixes such flaky tests in all the affected projects upon their migration to Gradle 8.6 or a subsequent version. Notably, this case is the first instance where we encounter ID tests that pose a considerable challenge to rectify, particularly as integration-testing a Gradle plugin necessitates invoking the Gradle runner, making target-project-scope fixes elusive. Following our investigation, we pinpointed 322 hard-to-fix ID tests from the 69 smaller-scaled Gradle projects, whose cause of failure (i.e, false assumption on an underdetermined API) is in the code outside of the project, and the project has no way to control that cause without removing the dependency on the external code. We recorded all such tests and the reasons in a separate

TABLE III
DETAILED VIEW OF GRADLE PROJECTS (SORTED BY THE NUMBER OF FLAKY TESTS IN JAVA 17)

Project	Stars (2024-11-11)	Commit	Total Tests	Flaky (Java 17)	Flaky (Java 21)	ID
apple/servicetalk	925	00e740c	19,573	1,045	637	9
java9-modularity/gradle-modules-plugin	233	5fadfd6	82	57	57	0
micronaut-projects/micronaut-core	6,083	6a1c54d	15,630	56	56	55
spinnaker/clouddriver	434	3a1e81b	5,051	46	46	46
aadnk/ProtocolLib	287	e77ed96	141	30	30	30
spring-projects/spring-authorization-server	4,872	6ec92a0	1,047	27	27	27
Consensys/teku	674	da3e51b	1,070	19	20	18
Netflix/Hystrix	24,136	7c03baf	692	11	11	11
conductor-oss/conductor	18,201	2e309b2	1,367	10	8	8
synthetichealth/synthea	2,178	16ee9bc	596	9	9	9
spring-projects/spring-restdocs	1,161	5cd47c6	1,104	9	9	8
elastic/elasticsearch-hadoop	990	3774f8f	1,449	8	8	8
ContextMapper/context-mapper-dsl	220	9129c0f	787	7	7	7
ReactiveX/RxJava	47,909	988e5ba	12,389	5	6	0
LuckPerms/LuckPerms	2,023	e6599a2	862	5	5	5
TeamNewPipe/NewPipeExtractor	1,381	3402cdb	2,119	4	0	0
mc1arke/sonarqube-community-branch-plugin	2,246	5e5ca50	342	4	4	4
JakeWharton/mosaic	1,925	f192af3	4	4	4	4
bitwig/dawproject	776	5818a71	12	4	4	4
spring-projects/spring-kafka	2,189	769da20	784	3	4	2
gazbert/bxbot	830	cd1f0af	231	3	3	3
qiniu/java-sdk	552	9440857	63	2	2	2
igvteam/igv	645	66f76ea	442	2	1	0
deepjavalibrary/djl	4,136	c69aeb6	198	2	2	2
controlsfx/controlsfx	1,576	a05c231	37	2	2	0
webauthn4j/webauthn4j	442	fb723ba	1,002	1	1	1
stripe/stripe-java	821	6c3dc05	62	1	1	1
spring-projects/spring-statemachine	1,562	1b2a868	705	1	1	0
schibsted/jslt	638	2c1d6ac	720	1	1	1
rsocket/rsocket-java	2,360	6d07389	1,078	1	2	0
Netflix/servo	1,418	7dbfafa	362	1	1	0
jvm-bloggers/jvm-bloggers	231	c7cbf81	1,447	1	1	1
jenkinsci/JenkinsPipelineUnit	1,544	6bbc2c3	258	1	1	1
elki-project/elki	792	7f7482c	1,245	1	0	0
antlr/intellij-plugin-v4	468	cf76d88	54	1	1	0
Total	136,858	N/A	73,005	1,384	972	267

repository [27]. Our case study emphasizes the importance of preempting false assumptions on underdetermined APIs to prevent their propagation through project dependencies.

VII. CONCLUSION AND FUTURE WORK

With substantial development strides in recent years, the NONDEX tool has evolved to effectively account for behavioral variations among different Java versions and build tools. The NONDEX tool can now be run on most Java projects using Maven or Gradle, facilitating the identification of ID flaky tests whose executions have false assumptions related to underdetermined APIs. Using the latest NONDEX allowed us to perform a large-scale study of flaky tests, and in particular ID flaky tests, in modern Java projects. Our experiments underscore the potential increasing prevalence of ID flaky tests in large-scale projects, and our case study reveals how flakiness can detrimentally propagate through the intricate web of modern project dependencies.

Our future plans involve integrating NONDEX with tools that can facilitate the automated resolution of NONDEX-reported flaky tests. Specifically, we can aim to leverage approaches based on large language models to automatically generate fixes, thereby alleviating the labor-intensive stages of rectifying flaky tests [28], [29]. While on one hand it is negative to see that the number of flaky tests has increased over time, on the positive more research opportunities arise to seek approaches to streamline the overall process and enhance the efficiency of addressing implementation-dependent flakiness at scale in modern software projects.

ACKNOWLEDGMENTS

We thank Yusen Wang for his help in inspecting test flakiness and starting some new NONDEX extensions for underdetermined APIs related to the Java streams. We also thank Philmon Roberts and Xinyu Wu for developing the first version of the new Gradle plugin and starting some scripts for experiments. This work was partially supported by NSF grants CCF-1763788 and CCF-1956374. We acknowledge support for research on flaky tests from Facebook and Google.

REFERENCES

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[2] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM TOSEM*, 2021.

[3] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao, "Research progress of flaky tests," in *SANER*, 2021.

[4] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[5] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *FSE Demo*, 2016.

[6] W. Lam, "International Dataset of Flaky Tests (IDoFT)," 2020. [Online]. Available: https://github.com/TestingResearchIllinois/idoft

[7] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[8] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[9] Oracle Corporation, "Java™ Platform Standard Ed. 8 - HashSet Javadoc," https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html, November 2024.

[10] ——, "Collections Framework Enhancements in Java SE 8," https://docs.oracle.com/javase/8/docs/technotes/guides/collections/changes8.html, November 2024.

[11] P. Deitel, "Understanding Java 9 Modules: What they are and how to use them," https://www.oracle.com/corporate/features/understanding-java-9-modules.html, 2017, accessed: 2024-11-11.

[12] Gradle Inc., "Gradle vs Maven: Performance comparison." https://gradle.org/gradle-vs-maven-performance, January 2024.

[13] NonDex GitHub Repository, "NonDex Source Code," https://github.com/TestingResearchIllinois/NonDex, February 2025.

[14] Pull Request #26452 in the Gradle GitHub Repository, "Fix order dependency in excluded group." https://github.com/gradle/gradle/pull/26452, November 2023.

[15] T. Preston-Werner, "Semantic Versioning 2.0.0," 2017, accessed: 2024-11-11. [Online]. Available: https://semver.org/

[16] *Java Platform Module System (JPMS) - Java 9 Documentation*, Oracle Corporation, 2017, https://docs.oracle.com/javase/9/docs/api/java/lang/module/package-summary.html.

[17] Pull Request #166 in the NonDex GitHub Repository, "Add support for Java 9+: Addressed code changes and resolved conflicts," https://github.com/TestingResearchIllinois/NonDex/pull/166, November 2022.

[18] Pull Request #183 in the NonDex GitHub Repository, "Support Java 17." https://github.com/TestingResearchIllinois/NonDex/pull/183, December 2023.

[19] Pull Request #196 in the NonDex GitHub Repository, "Support Java 21." https://github.com/TestingResearchIllinois/NonDex/pull/196, October 2024.

[20] Gradle Plugin Portal, "NonDex plugin for Gradle." https://plugins.gradle.org/plugin/edu.illinois.nondex, December 2023.

[21] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam, "Preempting flaky tests via non-idempotent-outcome tests," in *ICSE*, 2022.

[22] D. Silva, M. Gruber, S. Gokhale, E. Arteca, A. Turcotte, M. d'Amorim, W. Lam, S. Winter, and J. Bell, "The effects of computational resources on flaky tests," *IEEE TSE*, 2024.

[23] S. Dutta, A. Arunachalam, and S. Misailovic, "To seed or not to seed? An empirical analysis of usage of seeds for testing in machine learning projects," in *ICST*, 2022.

[24] 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 *ISSRE*, 2020.

[25] Pull Request #17472 in the Wildfly GitHub Repository, "WFLY-18823 Fix Flaky AbstractValidationUnitTest." https://github.com/wildfly/wildfly/pull/17472, December 2023.

[26] Pull Request #41 in the TestParameterInjector GitHub Repository, "fix: sort the nondeterministic arrays." https://github.com/google/TestParameterInjector/pull/41, November 2023.

[27] ID-HtF GitHub Repository, "ID-HtF Tests," https://github.com/kaiyaok2/ID-HtF.

[28] Y. Chen and R. Jabbarvand, "Neurosymbolic repair of test flakiness," in *ISSTA*, 2024.

[29] K. Ke, "NIODebugger: A novel approach to repair non-idempotent-outcome tests with LLM-based agent," in *ICSE*, 2025.