NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications

Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{gyori, blambet2, awshi2, legunse2, marinov}@illinois.edu

ABSTRACT
We present NonDex, a tool for detecting and debugging wrong assumptions on Java APIs. Some APIs have underdetermined specifications to allow implementations to achieve different goals, e.g., to optimize performance. When clients of such APIs assume stronger-than-specified guarantees, the resulting client code can fail. For example, HashSet's iterator order is underdetermined, and code assuming some implementation-specific iteration order can fail. NonDex helps to proactively detect and debug such wrong assumptions. NonDex performs detection by randomly exploring different behaviors of underdetermined APIs during test execution. When a test fails during exploration, NonDex searches for the invocation instance of the API that caused the failure. NonDex is open source, well-integrated with Maven, and also runs from the command line. During our experiments with the NonDex Maven plugin, we detected 21 new bugs in eight Java projects from GitHub, and, using the debugging feature of NonDex, we identified the underlying wrong assumptions for these 21 new bugs and 54 previously detected bugs. We opened 13 pull requests; developers already accepted 12, and one project changed the continuous-integration configuration to run NonDex on every push. The demo video is at: https://youtu.be/h3a9ONkC59c

CCS Concepts
•Software defect analysis → Software testing and debugging;

Keywords
NonDex, flaky tests, underdetermined API

1. INTRODUCTION
Some commonly used Java APIs have underdetermined specifications. Following Liskov [9], we say that a specification is underdetermined if it allows multiple implementations to return different results for the same input, even if each implementation is itself deterministic and always returns the same result for the same input. We refer to an API with such a specification as an underdetermined API. An example underdetermined API is the iterator() method in java.util.HashSet, whose Javadoc specification states, “The elements are returned in no particular order” [4]. Similarly, libraries for generating JSON typically do not guarantee any order for elements in a JSON document [2]. Having such underdetermined specifications is good because it gives implementers of the underdetermined APIs the flexibility to optimize various implementations of the API for different goals, e.g., they may optimize performance in different ways. However, it is important to precisely state underdetermined specifications in the API documentation to express all expected behaviors of all API’s implementations.

Unfortunately, even when underdetermined APIs have precise documentation, developers do make wrong assumptions about the underdetermined APIs. While such APIs could allow even non-deterministic implementations, each typical implementation is deterministic, i.e., two runs of the same implementation (on the same platform) give the same result for the same input. For example, two runs of a program that iterates over a HashSet may return the elements in the same order. However, such deterministic implementations can mislead the developers of API clients, who may assume that all API implementations are guaranteed to behave in the same deterministic manner. For HashSet, while one Java version could provide a deterministic iteration order, different Java versions provide different iteration orders (e.g., the order in Java 7 differs from the order in Java 8). If clients of an underdetermined API assume stronger-than-specified guarantees, the resulting code can fail when the API implementation changes, albeit the specification remains the same. A well-known example of such wrong assumptions is that many projects with JUnit tests relied on a particular order in which tests are executed; when these projects upgraded from Java 6 to Java 7, many tests failed because the order changed from Java 6 to Java 7 [8], albeit the specification of the order did not change.

The state-of-the-practice in detecting the negative effects of wrong assumptions on underdetermined APIs is rather reactive. Most developers discover such assumptions only after failures happen (e.g., after platforms are changed). Unexpected behaviors can also manifest as so called “flaky” tests [10], which can pass or fail seemingly without any changes to the code. A flaky test that assumes a certain behavior, which is not guaranteed by the API specification, can fail when the API implementation changes. The devel-
The non-Dex tool, a plugin entry to their build file (pom.xml).

2. USAGE
We describe how to use NonDex as a Maven plugin or from command line.

2.1 Integration with Maven
NonDex is easily integrated in the testing process via the Maven plugin, available from the Maven Central repository. To use NonDex, developers only need to add NonDex as a plugin entry to their build file (pom.xml).

The publicly released NonDex does not handle the native hashCode, because it did not expose any bugs during our experiments and would unnecessarily complicate the tool.
our instrumentation simply adds a custom NonDex rational. For APIs that should be modified and return an
the APIs that should be modified to add random explo-
view of the
sumption was made. Figure 1 shows an architectural over-
debugger identifies the API invocation(s) where a wrong as-
original files. However, this solution was brittle, because the
tool would often not work unless the exact same Java ver-
from the rt.jar that was instrumented.

3.1 Instrumentation Engine
The goal of the instrumentation engine is to modify stan-
dard Java library classes to allow random exploration. The
challenge is to develop instrumentation that can automati-
cally handle a large number of Java versions. For our original
prototype [16], we manually modified the Java sources of the
relevant files, compiled them, and used them in place of the
original files. However, this solution was brittle, because the
tool would not work even when the exact same Java version
e.g., 1.8.0-b132 was used for the run as the version
for which we manually modified the sources. The reason
the tool did not work was that some internal parts of the
modified files changed between Java versions, even when the
signatures of the public APIs we modified did not change.
Hence, we developed our current, instrumentation-based so-
lution that is much more robust, and we have tested it on
14 different versions of OpenJDK and Oracle’s JDK imple-
mentations of Java 8, on Linux, OS X, and Windows.

The instrumentation engine takes as input the rt.jar file
containing the classfiles of the standard Java library that
will be used when running the tests. The instrumentation
engine selects from rt.jar the classfiles corresponding to
the APIs that should be modified to add random explo-
ration. For APIs that should be modified and return an
array, our instrumentation simply adds a custom NonDex
helper method to explore different orders of the returned
array (effectively randomly permuting the array before re-
turning it). This modification is robust as long as the type
signature of the API does not change. The instrumentation
is much more involved for the ConcurrentHashMap classes,
because their iterators are lazy, implemented as private data
structures that change even within the same Java major ver-
version, e.g., the ConcurrentHashMap iterator was implemented using a class
called Entry until OpenJDK version 1.8.0-b108 [5] and
using a class called Node since then; we developed customized
instrumenters that can generate appropriate modified code
based on whether rt.jar uses Entry or Node. This modifica-
tion would need to change in the future if the standard Java
library implements ConcurrentHashMap using a third approach. We used
ASM [1] to implement all classfile manipulation.

Performing instrumentation from scratch on every run is
unnecessary, so we reuse each previously instrumented class
in subsequent runs, as long as the instrumented class from
rt.jar did not change. (The original class does not change
until/unless the user switches to another version of Java.)
To decide when to reuse the instrumented classes, NonDex
stores for each instrumented class the checksum of the class-
file from the rt.jar that was instrumented.

3.2 Runner
The runner is a thin layer of code that enables random exec-
ution for APIs instrumented by NonDex. On every
invocation of an instrumented API, the runner randomly
chooses one behavior from the behaviors appropriate for that
API. NonDex currently supports two kinds of behaviors:
(1) permutation for APIs where order is underdetermined,
and (2) extensions for APIs where only lower bounds on ar-
ray size(s) are specified. The runner takes as inputs (i) a
random seed, which completely determines the choice of be-
haviors, (ii) the mode of exploration—ONE or FULL (the two
modes differ in the kind of wrong assumptions they can de-
tect, as described in detail in our prior work [16]), and
(iii) optionally the range of choices to be randomized (which
is used by debugging).

3.3 Detector
The detector first runs all tests once without randomiza-
tion and then calls the NonDex runner a number of times,
with different random seeds, to rerun all the tests. The de-
tector reports tests that pass without NonDex randomiza-
tion but fail with NonDex randomization; such tests likely
make wrong assumptions on underdetermined APIs. The
detector first runs the tests without NonDex because tests
that fail on their own are due to some other causes and
should not be reported as failures due to wrong assump-
tions. After the first run, the detector invokes the instru-
mation engine to create the instrumented APIs before it
starts running tests with NonDex.

We originally evaluated four different modes, but the pub-
licly released NonDex offers only two modes, ONE and FULL,
because they are the easiest to understand and correspond
to the two extremes of nondeterminism.

The tests may be flaky [10] due to other reasons and fail
irrespective of NonDex.
The detector stores information about failing tests in a .nondex directory which also contains information about each execution, without and with NONDex, as well as the configuration used for test executions, the seed needed to reproduce the failure and the number of invocations of the runner’s choice generator; this number helps the debugging phase to search for the invocation(s) that caused the failure(s).

3.4 Debugger

When a test fails with NONDex, the test may invoke several underdetermined APIs, e.g., it may iterate over several HashSet objects. Many of these invocations are correct, making no wrong assumptions, so manually locating the invocation(s) that caused the detected failure can be tedious. The debugging phase automatically identifies such invocation(s).

To locate such invocation(s), NONDex uses a binary search that keeps track of a range of API invocations and selectively enables exploration for half of them. Even for disabled invocations, our search advances the random-number generator that shuffles the order of elements, but NONDex returns the original, not the shuffled, order. (Without this control, the search could get different behaviors for the same random seed, making it harder to reproduce the failure.) Debugging continues until a single invocation is identified, or the remaining range cannot be further halved. If a single invocation cannot be identified from running just one test method, NONDex re-starts debugging for the entire test class, and if again a single invocation cannot be identified, NONDex re-starts debugging for the entire test suite. Debugging is repeated for each failing test reported by the detector.

The debugging phase reports to the user an API that causes the detected failure together with the call stack of the API’s invocation which further helps in localizing the context in which the wrong assumption was made. In our prior work [16], we performed all debugging manually; after implementing automated debugging, we found that we had made an error in manually identifying the root cause of one failure, which shows that the automated debugging helps to more reliably identify the root causes.

4. EXPERIMENTS

Our initial NONDex prototype [16] detected dozens of tests in open-source code with wrong assumptions on underdetermined APIs. We experimented with our new NONDex tool by adding it to pom.xml for several open-source projects, running nondex:nondex (which detected 21 new bugs), and running nondex:debug (for the 21 new bugs and 54 old bugs).

4.1 Detecting Failures

To test the NONDex tool in general and the NONDex Maven plugin in particular, we integrated NONDex in the pom.xml files of several Maven-based projects from GitHub. Our goal was to test whether NONDex works with these projects “out-of-the-box” and not necessarily to detect any bugs. We found that integrating NONDex into these projects was indeed easy, and that by just adding a few lines to pom.xml, we could run NONDex on all these projects. NONDex worked well with projects that use different testing frameworks (e.g., JUnit 4, JUnit 3, and TestNG) and even various test runners (e.g., parameterized tests). Along the way, we also detected 21 new failing tests in eight projects (eight in alibaba/fastjson, five in checkstyle/checkstyle, three in nutzam/nutz, and one in each of alibaba/druid, bukit/bukkit, jankotek/mapdb, pedrogs.algorithms/algorithms, and perwendel/spark).

4.2 Debugging Failures

We further applied the automated NONDex debugging on these 21 newly detected and 54 previously detected failing tests to determine the root cause of each failure. The number of underdetermined API invocations that NONDex randomized per failure ranged from 5 to 9,710. The results showed that our simple binary-search debugging works extremely well for these cases—for 74 out of 75 failures, NONDex minimized the cause down to only one invocation; the remaining failure is for a test written in JUnit 3 for which the Surefire Maven plugin (used by NONDex to run tests) cannot easily run single test methods. We also counted the number of wrong assumptions on various APIs supported by NONDex; the invocations causing the failures were getDeclaredFields (41 cases), HashMap iteration (32 cases), and getGenericExceptionTypes (1 case). Because binary search is simple, we were surprised that it sufficed to identify only one invocation in all but one of the cases we tried. In the future, we plan to explore more sophisticated search strategies, such as delta debugging [18], and automated fixing.

4.3 Case Studies and Adoption

We opened 13 pull requests (PRs) for failures detected by NONDex, reporting the issue and providing a fix, in four open-source projects: five in alibaba/fastjson, five in checkstyle/checkstyle, two in scribejava/scribejava, and one in square/retrofit. We did not open PRs for all bugs that NONDex detected because we are not experts in the projects and could not easily provide a fix for each bug. All PRs we opened were accepted by developers except one PR in alibaba/fastjson. One of the developers of Checkstyle was quite pleased with the PRs we opened, asked us about the tool we used to detect the issues, and recommended that we integrate NONDex in their continuous integration; we indeed integrated NONDex in both pom.xml and .travis.yml for Checkstyle [2]. Furthermore, we are piloting the use of NONDex in a software testing course to educate students about wrong assumptions on underdetermined APIs. Students are using NONDex to find issues both in their own code and in open-source projects they are familiar with. Overall, we found NONDex to be robust enough for use both in real-world projects and in teaching.

5. CONCLUSIONS

We presented the design and implementation of the NONDex tool we developed to help in detecting and debugging wrong assumptions on underdetermined APIs in Java. NONDex is open source, integrates well with Maven, and can be also run from the command line. Using NONDex, we detected and debugged several failures in open-source projects.

6. ACKNOWLEDGMENTS

We thank Lamyaa Eloussi, Zach Gleason, Farah Hariri, John Micco, Timothy Tunnell, and Tiffany Yung for providing feedback and suggesting improvements to the NONDex tool. This research was partially supported by the NSF Grant Nos. CCF-1409423, CCF-1421503, and CCF-1439957. We also gratefully acknowledge the Google Faculty Award.
7. REFERENCES


