JMLKelinci

JMLKelinci is a tool that runs the AFL fuzzer on Java programs (using Kelinci) that use each program's precondition (written in JML) to bypass invalid generated inputs.

JMLKelinci has four advantages in comparison with Kelinci:

1. It makes testing more efficient since it catches and bypasses invalid inputs before running the program under test.
2. By avoiding covering branches with invalid inputs, the tool finds bugs with valid inputs. Valid inputs are both more efficient to run and more valuable for demonstrating bugs.
3. The pool of (interesting) tests that will be generated with JMLkelinci consists of valid inputs (except for at most one invalid input). Thus, the genetic algorithm used by Kelinci has an increased probability of generating other valid inputs; this should result in achieving branch coverage of the program under test with valid inputs more efficiently.
4. The initial seed for the fuzzer can be chosen with fewer restrictions.

Also, 28 examples from the Java+JML dataset used in a paper under review are available to reproduce that paper's results. **To reproduce those results, first do the installation process and then go to the "Executing Examples" section and use the provided shell scripts.** Each program was run five times with Kelinci and JMLKelinci; valid generated input tests for each run are collected manually in a JUnit test.

Installation

For using JMLKelinci both Kelinci and OpenJML should be installed in a Linux system using the following steps. (We used openjdk version "1.8.0_282" on Ubuntu 18.4, also both Kelinci and OpenJML require Java 8.)

In the Tool directory you can find last release (in March 2021) of "openjml0.8.52", "Kelinci", and "afl2.52b". You need to install these to use JMLKelinci as described below. (You can find later releases of these tools by the following links: OpenJML, Kelinci, and AFL.)

For using JMLKelinci after downolading th Tool directory, go into the Tool directory in your system and open a terminal. Then, run the "setupTool.sh" from that directory to set up the fuzzer on your system. Next, make the "runrac.sh" shell script (which is in the ShellScripts directory) executable for the Linux system with the following steps.

1. Open the shell script (for example with the `vim runrac.sh` command).
2. Change the address of in the shell variable `OPENJML` to the full path to where OpenJML has been extracted in your system; for example you might change the script to say:

   ```
   OPENJML="$HOME/Tool/openjml"
   ```

3. Then run `chmod u+x runrac.sh` to make the shell script executable for your system.

Usage

Requirements:

1. Java 8 in a Linux system. (We used Ubuntu 18.4)
2. Kelinci

3. OpenJML (we used version 8.46, but it also works with later versions, such as 8.52)

Inputs of JMLKelinci:

1. Java Program Under Test (JPUT), in Java source code format. The JPUT will be compiled by Kelinci to instrument the program.
2. JMLKelinci's Driver, also Java source code for a main program. (This driver converts byte streams to JPUT's arguments. Then, it runs JMLDriver with RAC to check preconditions. If precondition is satsified, then it will run JPUT.)
3. Initial seed (see below)
4. JPUT entry method's precondition. The entry method is the method called by the JMLKelinci driver. This method must have a JML annotation for its precondition, and best practice is to remove its body (to save execution time).
5. JMLDriver (To run the entry method with precondition).

The entry method's precondition can be written manually in JML (or it could be inferred, e.g., by using Daikon). A JMLKelinci driver, which is a Java main program, is needed to convert the generated byte stream of the fuzzer to the type of the entry method's arguments. The JMLKelinci driver needs to run the OpenJML's RAC (see examples in the JMLKelinci evaluation directory). JMLKelinci needs an initial seed that does not make the program crash for starting the fuzzing process.

Example Inputs

See the LeapYear example for a sample of these inputs.

1. The JPUT is in the file src/LeapYear.java.
2. The JMLKelinci driver is src/KelinciDriverMain.java
3. The initial seed is in_dir/example.txt
4. The entry method's precondition is the JML annotation in jml/LeapYear.java
5. The JMLDriver is jml/JMLDriver.java

Execution Instructions

After the installation process, and making the "runrac.sh" executable for the Linux system follow the following instruction:

1. Build a driver: Writing a fuzzer driver is essential for using JMLKelinci (like Kelinci and other similar tools). JMLKelinci's driver converts generated byte streams into the program's argument. Also, the driver calls the RAC to check preconditions. 28 drivers are written for both standard Kelinci and JMLKelinci in each corresponding directory.

2. Build an entry method and JML precondition: The program with JML precondition must be prepared by taking out the body of the method called by the driver (to save time by avoiding running the program to check precondition) that is annotated with JML's precondition. (look at JMLKelinci examples, you can find it in the "jml" directory of each program).

3. Build the JMLDriver: A JMLDriver is a simple driver that runs the entry method with its precondition.

4. Set up directory structure: Five directories should be created, named: `src`, `jml`, `bin`, `bin-instr`, and `in_dir`. (Look at JMLKelinci examples.) The `src` directory contains the Java program under test (JPUT) and the fuzzer driver. The `jml` directory contains the entry method (with JML precondition) and JMLDriver. The `bin` directory is initially an empty directory, and will contain the compiled version of the program under test and the fuzzer driver. The `bin-instr` directory is initially an empty directory, and will contain the instrumented bin file of the program under test and the fuzzer driver. The `in_dir` directory contains the initial seed(s) that we will explain later in detail.

The following command will create the needed directories on Linux:

```
mkdir src; mkdir jml; mkdir bin; mkdir bin-instr; mkdir in_dir
```

5. Compile and Instrument:

To compile and instrument the program under test, follow the commands below. (We assume that JPUT and the fuzzer driver are in the `src` directory.)

First open a shell in a directory that has `src`, `jml`, `bin`, `bin-instr`, and `in_dir` subdirectories. Then, run the following commands.
We assume that OpenJML is installed in the directory `$OJ`:

```
cd jml
java -jar "$OJ/openjml.jar" -rac -racPreconditionEntry *.java
cd ..
```

The above command compiles the precondition in the `jml` directory.

Then run the following commands (from the same directory). We assume that Kelinci is installed in the directory `$Kel`.

```
cd src
javac -cp ".:$Kel/instrumentor/build/libs/kelinci.jar" *.java -d ../bin
cd ..
```

The above command compiles programs in the `src` directory, and the destination of the compiled programs is the `bin` directory. (Thus the bin directory should no longer be empty.)

Then, use the following commands to instrument the program for the fuzzing process.

```
java -jar  "$Kel/instrumentor/build/libs/kelinci.jar" -i ./bin -o ./bin-instr
java -cp ./bin-instr/ edu.cmu.sv.kelinci.Kelinci KelinciDriverMain @@
```

6. Initial Seed(s): You should provide an initial seed in a file the directory named `in_dir`. (Any files there will be used.) The program under test should not crash by using this seed. The fuzzer uses this seed in the mutation process to generate new inputs.

7. Start the fuzzer server: To start the fuzzer server open a terminal in a directory that has `src`, `jml`, `bin`, `bin-instr`, and `in_dir` subdirectories. The, use the following commands.

```
java -cp "bin-instrumented:$Kel/instrumentor/build/libs/"*
edu.cmu.sv.kelinci.Kelinci <driver-classname> @@
```

Also, the above command used the default port which is define for JMLKelinci and Kelinci (the default port is 7007). Instead of the above command you can specify a port number with the following command (following command selects port 5000) to run several fuzzer in a system with using different ports:

```
java -cp "bin-instrumented:$Kel/instrumentor/build/libs/"*
edu.cmu.sv.kelinci.Kelinci -port 5000 <driver-classname> @@
```

8. Start fuzzing process: Open a new terminal in a directory that has `src`, `jml`, `bin`, `bin-instr`, and `in_dir` subdirectories. Use the startFuzzing.sh shell script to run the fuzzer, after setting the Kel shell variable to the directory where Kelinci is installed (in the example this is `~/Tool/kelinci`.

```
Kel=~/Tool/kelinci
export Kel
startFuzzing.sh
```

If everything works correctly, then JMLKelinci will use AFL to cover the JPUT's branches with valid inputs.

9. Output: After a short time, the AFL interface will start to discover the JPUT's branches with valid inputs. Also, an output directory will be generated (`fuzzer-out` in this study). The fuzzer will save all (interesting) valid inputs in a `queue` subdirectory that trigger different program behaviors (discovering new branches). Also, "crashes" and "hangs" subdirectories will contain valid generated inputs that resulted in a crash or a time-out. See the AFL website for more details.

The fuzzer will run until stopped. You will thus need to stop it (say, using Control-C) when you believe it has run long enough. Alternatively you can use the Unix `timeout` command when running `startFuzzing.sh` as in the following, which will run the fuzzer for two days:

```
timeout 2d startFuzzing.sh
```

Executing Examples

We used 28 programs from the Java+JML dataset and 28 buggy programs from the BuggyJava+JML dataset.

In these examples, we provide all of the necessary inputs (a Java program under test, an entry method with a JML precondition, a fuzzer driver, a JMLDriver, and an initial seed) to cover branches with valid inputs.
In our experimental study, we ran each of the 28 correct programs of the Java+JML dataset in the `JMLKelinci` and `Kelinci` directory five times (until the fuzzer reached 100% branch coverage), and we manually provided a JUnit test for each run based on the valid inputs. (We took out generated invalid inputs, using the JML RAC to check.)

We provide two shell scripts with the name `instrumentJMLKelinci.sh` and `startFuzzing.sh` to run these examples, which are in the ShellScripts directory. Before running these shell scripts, you should update the `Kel` and `OJ` variables in each of them, setting them to the full paths to the directories where Kelinci and OpenJML are installed (respectively).

For this study, we used OpenJML v.0.8.46, but the examples also work with the (March 2021) last release (v.0.8.52).

Running the Examples for JMLKelinci

To run JMLKelinci examples, open a terminal in the directory that you can see `src`, `jml`, and `in_dir`. Next, run the `instrumentJMLKelinci.sh` script and then open a new terminal (without closing the first terminal) in the same directory and then run `startFuzzing.sh` in the new terminal. As explained earlier, make sure that the shell variable `Kel` is set to the full path of the directory where Kelinci is installed and `OJ` is set to the full path to the directory where OpenJML is installed.

After running both shell scripts, you will see the `fuzzer-out` directory that will have all generated (interesting) inputs that are valid and discover a branch or that lead to a crash or time-out, as explained earlier in Execution Instructions part 9.

For running the buggy versions of the examples, run the same process as above on a buggy program.

Running the Examples for Kelinci

To run the Kelinci examples, open a terminal in a directory that you see `src`, and `in_dir`. Then, 1) run the `instrumentKelinci.sh` script to compile and instrument the Java program under test and the fuzzer driver. 2) Open a new terminal (without closing the first terminal) and run the `startFuzzing.sh` script to start fuzzing and discovering new branches.

After running both shell scripts, you will see the `fuzzer-out` directory that will have all generated (interesting) inputs (which may be invalid) that discover a branch or lead to a crash or time-out, as explained earlier in Execution Instructions part 9.

Extracting test from generated data test by JMLKelinci

After JMLKelinci (or Kelinci) starts to run, the tool will create a new directory named "fuzzer-out" that saves interesting generated data. The generated data by JMLKelinic that are interesting will be saved in the "/fuzzer-out/afl/queue" (those that have "+cov" discovered a new branch), and those that find a crash will be saved in the "/fuzzer-out/afl/crashes" directories.

It is needed to change the generated data by JMLKelinci to actual input tests (like other guided fuzzer tools that have a driver). Thus, the same driver is needed for extracting actual inputs for running the JPUT. For all examples, we provide an "ExtractDriver.java" to help to extract files, but it needs some manual work. First, after running the fuzzer tool, move the "ExtractDriver.java" and the JPUT that is in the "\src" directory to the "/fuzzer-out/afl/queue". Then, copy the name of the each generated data test and paste it in the "FileInputStream fis=new FileInputStream("id:....");" and then run `javac *.java ; java ExtractDriver`.
We did it for our five runs in the experimental results, and we create a JUnit test based on each run.

Architecture

The tool has two main components.

The first is the fuzzer side (Kelinci) that instruments the Java program, generates new inputs, monitors branch coverage, and discovers crashes in the program. Also, Kelinci itself has two parts, which are its Java and C parts. The C part sends the input files generated by AFL to the JAVA part over a TCP connection. It then receives the results and forwards them to AFL. The Java side instruments a target application with AFL style administration, plus a component to communicate with the C side. When executing the instrumented program, this sets up a TCP server and runs the target application in a separate thread for each incoming request. It sends back an exit code (success, timeout, crash, or queue full), plus the gathered path information.

The second component is OpenJML's runtime assertion checker (RAC). The JML RAC bypasses invalid inputs by evaluating the program's precondition. We assume that the JML precondition of the entry method and any necessary constructors are available. There are some tools available that can infer preconditions for JML, like Daikon. OpenJML's RAC is called in a separate process after inputs are generated with Kelinci. If the precondition is satisfied, then Kelinci will run and monitor the program under test. Also, if the precondition is not satisfied, Kelinci will generate another input. In JMLKelinci, the process's exit code is queried to determine if the program's precondition was satisfied.