Fuzzing and Analysis Report

Prayani Singh *psingh25@ncsu.edu*
Shefali Agarwal *sdagarwa@ncsu.edu*
Sujal Sujal *ssujal@ncsu.edu*
Vaibhav Singh *vsingh7@ncsu.edu*

I. Introduction

Static Analysis is a method for analyzing code to detect code-smells without executing the project itself. Fuzzing is a technique used mainly to identify bugs that maybe syntactical, a code-smell or security-critical. Fuzz testing is a method which is randomly modifies or generates inputs which are then committed into the code and unit tests are run against this modified code. This testing is used to discover highly-dependent functions and code. In this report, we conduct fuzz testing on a project called iTrust and static analysis on a project called Checkbox.

II. iTrust

A. Creating a Fuzzer

For iTrust, with a single Fuzz testing run, we perform the following steps:

- Making changes to .java files in the iTrust project through Ansible.
- Commit and push repo changes into forked repo. (Fuzzer branch)
- Build Tests for that commit.
- Reset Git to original state.
- Get console-output logs from Jenkins.

1. Making changes to files
Initially we were trying to get all files at given path via Ansible but fuzzing files with a Python script seemed more convenient and thus we switched to that. Once the fuzzer script was ready, we changed all files and committed it to the repository.

Later we added probability to our commits so that we can avoid changing all files every time. This was also done in order to gain some confidence about test case that are more likely to pass over the test cases that are more likely to fail.

2. Commit and push repo changes into forked repo
Once we have random file commits in place, we committed those changes into git and pushed them to our forked repository for iTrust based on the 'fuzzer' branch.

3. Build Tests for that commit
Build should automatically be triggered once a commit has been made. After this commit, we added a timeout of 250 seconds (which is configurable) so that our Jenkins build can get triggered and runs for that particular commit.

4. Reset Git to original state
Once the build is run, and completed with success or failure, we reset git back to its original state where all test cases pass.

5. Get console-output logs from Jenkins
Jenkins store the console output of recent builds that are in following states: Unstable, Stable, Successful, Unsuccessful, Failed.

We take the output from the last failed build and store it in a log file where all our test case outputs will be combined.

B. Test Prioritization

For unit test parsing and prioritization:

We create a combined and parsed log file of all test cases Parsing includes getting the following data for each test-case

- Number of times test-case passed
- Time taken to build
- Collect data for n test-runs (in our case n=100)
- We run the following commands in a loop, where each command is sequentially executed 100 times.

Run Fuzzer: Collect test run and parsed output for each test run in a combined log To combine logs, we accumulated data for each test case into number of times passed/failed and time to build. We had to do come changes in pom.xml file in iTrust repo so that the logs would have time and status information about all tests.

Set test prioritization based on the data provided: Based on number of passed test cases and the build time for each, we set the priority of each test and added them to the report in order.

C. Analysis

We ranked test cases on the bases of number of times the test case failed as well as the time taken to build a test case.

1) ***What type of problems do you think the fuzzer discovered?****:* Fuzzer discovered the following problems: It discovered issues with functions with different inputs and outputs which when changed were giving different results.

Based on our tests:

14 tests out of 95, failed every time that is 100 out of 100 times and 2 out of 95 failed 93 times out of 100 The functions representing these tests represent are tightly coupled. Even though the files are randomly modified, these test case are most likely related to functions that test the over view or a higher order function that requires result of multiple internal functions.

6 out of 95, never failed. These functions do not have any logical dependency and have a very high probability of running and passing in every test. It also means that these functions are more robust and most likely will not break when making changes into the project.

73 out of 100 test case are in the range of 1 to 12 in terms of failure out of 100 runs. These tests failed with the range of 1 to 12. It feels like these functions are more or less in the same

range in terms of complexity and thus should be thoroughly tested after making a change in the project.

*2) **What are some ways fuzzing operations could be extended in the future?**:* Fuzzing operations can be extended in the future by:

Changing long and complicated functions to smaller functions and adding unit tests accordingly. Better String manipulation logic Removing keywords that define classes and variables' access (protected, private, public etc) to test if that can lead to test case failures and security for variables and values. Fuzzing Security or authentication tokens or keys so that we can see if we can break into the system. It can be used to discover memory leaks. It can be used to check if rules and restrictions defined for variables or project are being followed by the given unit tests.

*3) **Why do you think those tests were ranked the highest?**:* We believe that the test cases were ranked the highest as they have more conditional and logical statements and thus are more likely to fail in scenarios when changes are being made.

Moreover, we have configured the build job to fail according to build minimum testing criteria in jacoco. The thresholds for iTrust job are configured as follows:

Branch	Complexity	Line	Method	Class	
healthy	65	70	75	85	95
unhealthy	50	50	50	50	50

The JaCoCo thresholds were chosen based on existing test coverage numbers. Class complexity is kept the highest while branch complexity is kept the lowest as Class Complexity is a subset of branch complexity and is easier to cover (a Class can be considered "Covered" even if all the conditional branches in the class are not covered.)

	All priorities	High	Normal	Low
unstable	16900	0	16800	100
fail	16900	0	16800	100

The Checkstyle metrics were chosen based on existing Checkstyle warnings in iTrust repo. A margin of 40 warnings was added to Priority Normal as well.

III. CheckBox

A. Creating a Static Analyzer

We implemented the following static analysis heuristics. - Number of conditions in a function - Number of lines in a function - Number of characters in a line For implementing these, we used a library, Esprima for creating an Abstract Syntax Tree (AST) and based on the tree and it's structure, we calculated these metrics. Apart from the ones we have already implemented, we can added other metrics like checking duplicate methods.

B. Generating Report

The reports were generated using logs that we printed out while analyzing our file. It consists of the following metrics threshold that we have made configurable through Ansible variable prompt. Our script triggers a build in Jenkins which passes or fails based on threshold conditions for each of the custom heuristics. We have done this so that the threshold for each metric can be modified as per the requirement of the software.

Metric	Threshold
Number of Conditions	6
Number of Lines	80
Number of characters	170

These metrics were taken into consideration as we analyzed the checkbox project and concluded that it suffered from tightly coupled code and really long functions that could be broken down into smaller functions.

C. Describe approach for analysis. How do you think these checks might help software developers?

Max condition was implemented by parsing the AST tree recursively. The number of conditions within an IfStatement was incremented whenever the parser encountered a "Logical Expression".

A Long method was detected by taking the difference of the location of the starting and ending line of the function. A method is considered long when it exceeds more than 80 lines of code. The maximum method's length in checkbox is 80 currently and none of the methods in checkbox is violating it.

Character Count was kept at maximum of 165, if the criteria is violated, the build for checkbox is failed. We chose this limit considering the fact that none of the functions/classes exceed this limit in the current checkbox files.

Static analysis tests on code would help keep code clean and maintainable, and reduce the overhead in getting a new developer. They will also help in reducing potential bugs and root causing any issues which arrive. Metrics and checks like we have implemented will specifically help Software developers to write modular code and segregate multiple decisions into different parts of the code. This will also help in maintaining good coding practices.

IV. Links

Wiki:
https://github.ncsu.edu/ssujal/CSC519_Project/wiki

Reports:
https://github.ncsu.edu/ssujal/CSC519_Project/tree/master/reports

Git Repositories:
https://github.ncsu.edu/ssujal/CSC519_Project
https://github.ncsu.edu/vsingh7/iTrust2-v4
https://github.com/Shh25/checkbox.io