Software Engineering Assignment 2 Report

CSE46401

20151509 Hyunjoon Jeong

In this assignment 2, I applied JQF[1] fuzzer on Google Closure, which is Java interpreter to compress the code. There were three versions of Google Closure : maven-v20160911, maven-v20170806, and maven-v20180204. In this report, JQF generated random inputs to three versions and found which points crashed with them.

```
.id_000006 ::= FAILURE (java.lang.RuntimeException)
E
Time: 2.189
There was 1 failure:
1) testWithGenerator(com.google.javascript.zest.CompilerTest)
java.lang.RuntimeException: INTERNAL COMPILER ERROR.
Please report this problem.

Unexpected variable NaN
  Node(NAME NaN): input:1:18
while (((k_0) ^= (undefined))){  }
  Parent(ASSIGN_BITXOR): input:1:8
while (((k_0) ^= (undefined))){  }

        at com.google.javascript.jscomp.VarCheck.visit(VarCheck.java)
        at com.google.javascript.jscomp.NodeTraversal.traverseBranch(NodeTraversal.java:645)
        at com.google.javascript.jscomp.NodeTraversal.traverseChildren(NodeTraversal.java:717)
```

Figure 1. maven-v20160911 random input and result

In maven-v20160911 Google Closure, the random generated input is in above Figure 1. In JavaScriptCodeGenerator, most of statement generations depends on uniform distribution of random class. We can see that there are three types of randomness generation in that class : random.nextBoolean, random.nextChar, random.choose and random.nextInt. These random method helps generations of nodes, so that generated code can be passed to compile function as string type in CompilerTest.java. When the program enters “testWithGenerator” method, JavaScriptCodeGenerator class also enters “generate” method, which is a main entry point of generating random test case. Then, it tries to return random statement at the end of the method. In “generationStatement” method, the program starts to generate random node. Each node, except some specific node such as ident generation, calls “generateExpression” method, which generates expressions by recursive method call. In Figure 1 case, randomness chooses “while” statement, because the depth of statement was not deep enough, and Boolean randomness returns false. In the while statement generation method, it executes generating condition expression and its body block. Since the recursive execution follows depth first, block generation would be executed at the end of order. From Figure 1 result, the selected expression was binary node. In “genearteBinaryNode” method, it chooses binary token, which is predefined in BINARY_TOKENS array, randomly. After token “^=” was selected, the method

divided condition expression into two part. In this case, left side executed first, and selected method was “generateIdentNode”. In this method, there is no more recursive execution. So, the method was returned with random identifier “k_0”. Then, right side of binary node would be executed. The selected expression method was “generateLiteralNode”. In the literal node generation, it operates Boolean random selection first, and it generated primitive literal “undefined” in this case. Like indent node, literal node does not generate other expression anymore. Since binary node was complete, the block generation would be executed. In “generationBlock”, it tries to fill while statement body block into “generateItems” method. In this method, items would be generated randomly, as much as given mean parameter twice. However, in above input case, zero was selected randomly. So, empty body block was generated. This is input generation process and the execution process is represented in Figure 2.

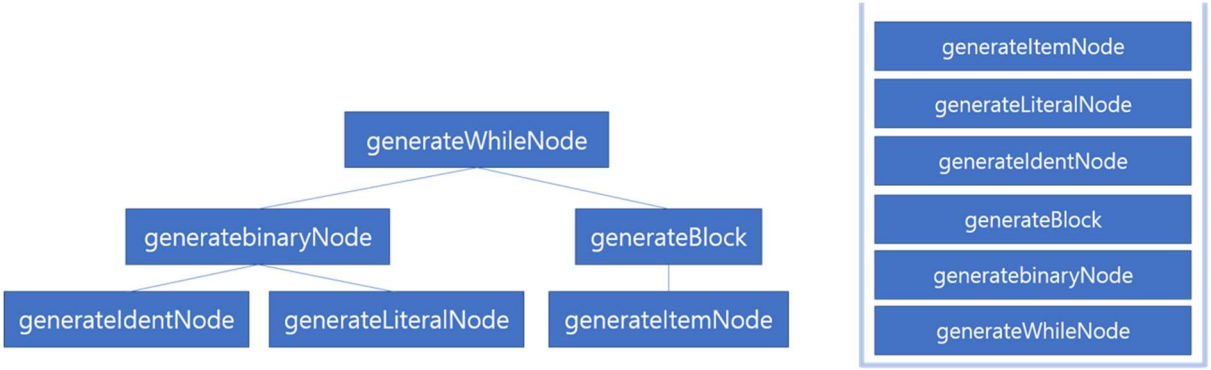

Figure 2. Tree and stack state of maven-v20160911 input code generation

From maven crash log, I identified that randomly generated input caused “Unexpected variable NaN”. This exception log came from VarCheck.java class during JavaScript compression. In Google Closure, it wants to compress generated code, so that the program traverses each nodes. Like Figure 2, the executed order would be constructed tree structure and the program checks variable at leaf of the tree. In this case, when VarCheck class checks each leaf, the literal node would refer “undefined”. In this situation, we can identify “sanityCheck” in “visit” method. This checking ensures that variable have either been declared or marked as extern. However, “undefined” means the primitive type that refer not to have experienced initialization yet. Moreover, in JavaScript, “undefined” is regarded as “Not a Number”. For this reason, the program crashed in above random input.

```
id_000001 ::= FAILURE (java.lang.RuntimeException)
E
Time: 2.501
There was 1 failure:
1) testWithGenerator(com.google.javascript.zest.CompilerTest)
java.lang.RuntimeException: INTERNAL COMPILER ERROR.
Please report this problem.

Unexpected variable NaN
  Node(NAME NaN): input:1:2
(({x_0: (x_1)}) << (s_2))
  Parent(LSH): input:1:1
(({x_0: (x_1)}) << (s_2))
```

Figure 3. maven-v20170806 random input and result

In maven-v20170806 Google Closure, the random generated input is in Figure 3. Like previous version (maven-v20160911), random input generation follows recursive method call. At first, in CompilerTest.java, when "testWithGenerator" method was called, "JavaScriptCodeGenerator" class starts to prepare random input as parameter. So, the program enters "generate" method as main entry point of random input generation. Then, it wants to return the result of "generateStatement" method. In that method, randomness selects "generateExpressionStatement" method, so that the generated input decided to be an expression statement. Then, it calls "generateExpression" method and selects its components recursively, which is similar with previous version. Then, random choose method calls a binary node generation method with binary token "<<". Like previous version, this "generateBinaryNode" furcates two additional expression at the end of the method. Since left side method executes first, "generateLiterNode" selects object literal by randomness way (calling random.nextBoolean). The method tries to generate object property items in bracket as much as "mean" parameter. These are carried as parameters to "generateItems" method, and the method chooses the number of items between [0, mean * 2) integer. In above input case, one item was selected. Then, "generateObjectProperty" would be executed and it generates two ident node. Since "generateIdentNode" returns only ident and does not execute other "generateExpression" method, recursive step would also return. Finally, the right side of binary node would be executed, and ident node generation was selected. So, the process to generate random input would be done. Figure 4 shows that the order of generating expression step as tree and stack structure.

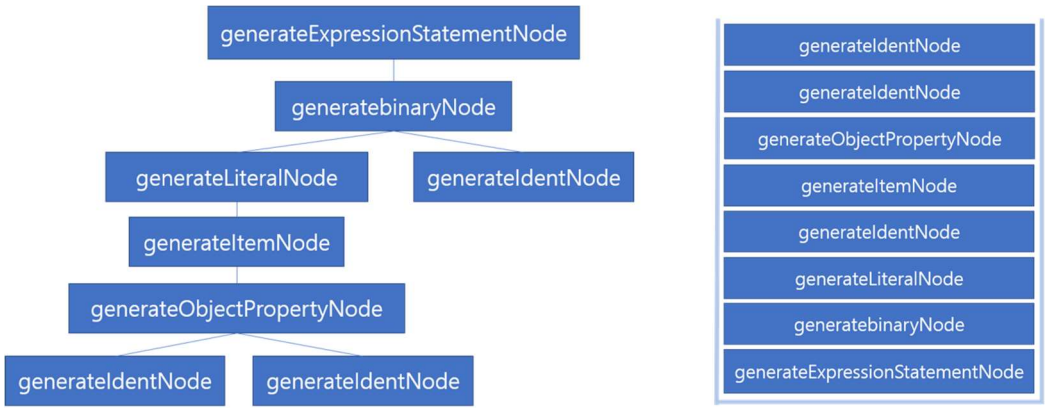

Figure 4. Tree and stack state of maven-v20160911 input code generation

From generated input, the result shows java.RuntimeException with unexpected variable. Like maven-v20160911 version, Google Closure detects generated JavaScript code to contain NaN variable or equivalent to NaN. So, similar with previous version, VarCheck.java tries to check sanity of leaf node and generated input would have problem in sanity check. In generated input, each ident component (x_0, x_1, and s_2) is regarded as externs, because they are undeclared variables. If undeclared variable or code tries to be reference, sanityCheck will throw IllegalStateException. In above input case, left side of binary token construct object, which has property x_0 with x_1. However, x_1 is not declared variable, and binary token "<<" wants to refer it. For this reason, program should crash and generated input should be failed.

```
[INFO] --- jqf-maven-plugin:1.6:repro (default-cli) @ closure-compiler-main ---
.throw (-- ((() => (() => (x_0))).l_1))
id_000010 ::= FAILURE (java.lang.NullPointerException)
E
Time: 2.961
There was 1 failure:
1) testWithGenerator(com.google.javascript.zest.CompilerTest)
java.lang.NullPointerException: NAME x_0 1 [length: 3] [source_file: input]
        at com.google.common.base.Preconditions.checkNotNull(Preconditions.java:787)
        at com.google.javascript.jscomp.RemoveUnusedCode.getVarForNameNode(RemoveUnusedCode.java)
        at com.google.javascript.jscomp.RemoveUnusedCode.traverseNameNode(RemoveUnusedCode.java)
        at com.google.javascript.jscomp.RemoveUnusedCode.traverseNode(RemoveUnusedCode.java)
        at com.google.javascript.jscomp.RemoveUnusedCode.traverseChildren(RemoveUnusedCode.java)
```

Figure 5. maven-v20180204 result

One of the generated inputs, which lead to crash program, was in Figure 5. In this version, as I mentioned before, JavaScriptCodeGenerator depends on uniform distribution. When testWithGenerator in CompilerTest.java was running, JavaScriptCodeGenerator class generated nodes recursively. At first, it entered “generate” method, which is main entry point. Like above other versions, it wants to return automated JavaScript statement, so that it calls “generateStatement” method. This method uses randomness to construct statement, and, in above case, throw node was selected. Since throw statement was selected, it calls “generatedThrowNode” and its “generateExpression” calls “generateUnaryNode” recursively. So, decreasing operation “--” picked from UNARY_TOKENS array by randomness, and it calls “generatePropertyNode” in its “generateExpression”. In property node construction, it calls two functions : “geneartеExpression” and “generateIdentNode”. The process executes expression generation first, and program entered “generateArrowFunctionNode”. In this case, the method calls “generateItems” method for arrow function parameters. In that method, the number of parameters is decided by “mean” parameter. By this “mean” parameter, the random.nextInt function returns integer in [0, mean*2). However, the input shows the case that the random integer was zero. So, there was no parameters in arrow function. Then, by Boolean randomness, it calls “generateExpression”, and it calls “generateArrowFunctionNode” again. Like previous arrow function, it also selected zero, so there were no parameters. Then, it calls “generateIdentNode” in “generateExpression”. Since ident node generation does not call “generateExpression” anymore, it returns recursively and goes up stream util the top of stack is another “generateIdentNode”, which was called by “generatePropertyNode” before. As a

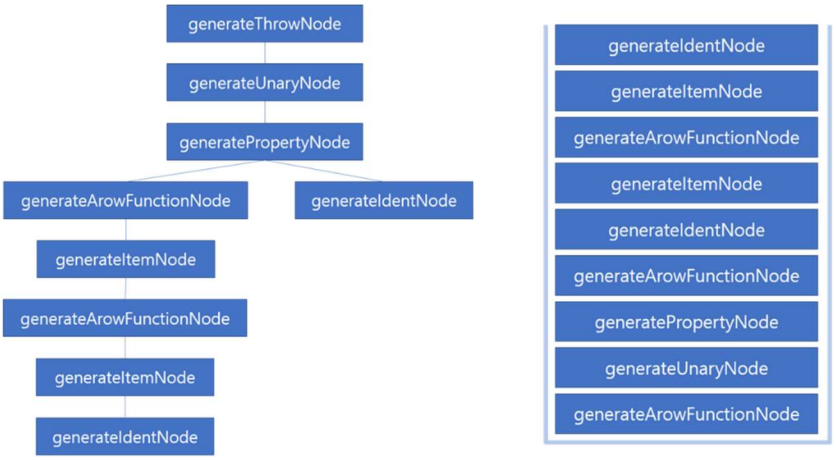

Figure 6. Tree and stack state of maven-v20180204 input code generation

result, Figure 6 shows the method calling as a tree and stack structure of maven-v20180204 input generation process.

Usually, the exception message java.lang.NullPointerException occurs when the application attempts to use null in a case where an object is required. So, there are five cases : calling the instance method of a null object, accessing or modifying the field of a null obeject, taking the length of null as if it were an array, accessing or modifying the slots of null as if it were an array, and throwing null as if it were a throwable value[2]. Among these five cases, the second case represents maven-v20180204 fuzzing crash. In Figure 3, the random input code generated arrow function, which returns x_0. Then, outer arrow function wants to access ident l_1, which is property of x_0. However, x_0 was not initialized before, so that ident l_1 pointed null object. As a result, the program had to throw exception.

From now, I identified Google Closure bugs with fuzzing. In the paper[3], Zest found three types of bugs in Google Closure : NullPointerException, RuntimeException, and IllegalStateException. In my case, I could only find two cases, which are NullPointerException and IllegalStateException. In that paper[3], RuntimeException occurred by new variable declaration after break statement. Also, in the table 2 of that paper, this RuntimeException bug can be found within average 460.42 seconds. However, although the program run almost 1 hour, this bug cannot be found. Since fuzzing depends on randomness and mutation, if parent does not generate this pattern, there can be possibility not to catch related bugs. Other interesting is, as more fuzzing time goes on, generated input become more complex.

```
  Node(NAME NaN): input:1:9
while ((((undefined) - (240)) != ([]))){  }
  Parent(NE): input:1:8
while ((((undefined) - (240)) != ([]))){  }
```

Figure 7. id_000000 input in maven-v20170806 failure

```
throw (((({q_0: ((((q_0)(((q_0).q_0),(this))).y_1).q_0), q_0: (v_2)}) ? (function(h_3, k_4){ (({v_2:
({h_3: (~ (p_5)), u_6: (v_7), j_8: (((k_4) &= (v_2)) ? (q_9) : ((undefined)[(null)])), k_4: (((undefi
ned)((u_10),(true),(h_3),(u_10
),(y_1))) != ((undefined) - (true)))}), u_6: ({}), w_11: (k_4), d_12: ({}), c_13: (k_14)})()) }) : (9
17)).j_8).c_15)
```

Figure 8. id_000049 input in maven-v20170806 failure

Both Figure 7 and 8 shows failure cases, which represent initial input and final input during 1 hour. Since mutation occur a lot of times from their ancestor, Figure 8 input case would become complex than Figure 7. From Zest paper[3], random generation methods, which mean random.nextInt, random.nextChar, random.nextBoolean depends on untyped bits flipping. So, these untyped bits, which paper said “parameters”, can make bit-level mutations and ensure syntatically valid inputs when their generator assumes that only generate syntatically valid input also. Finally, like Figure 3 input case, sometimes generated code tries to access uninitialized variables. JQF fuzzing just generates random input, regardless of expression or statement dependency.

Reference

[1] Rohan Padhye, Caroline Lemieux, and Kousik Sen. 2019. JQF: Coverage-guided Property-based Testing in Java. *In proceeding of the 28th SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19).*

[2] Oracle Java SE 7 document

(*https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html*)

[3] Rohan Padhey, Caroline Lemieux, Koushik Sen, Mike Papadakis, Yves Le Traon. *Semantic Fuzzing with Zest*. ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis