symmetry

MDPI

Article

JDriver: Automatic Driver Class Generation for AFL-Based Java Fuzzing Tools

Zhijian Huang and Yongjun Wang *

College of Computer, National University of Defense Technology, Changsha 410073, China; zjhuang@nudt.edu.cn

* Correspondence: wangyongjun@nudt.edu.cn

Received: 31 August 2018; Accepted: 27 September 2018; Published: 3 October 2018

Abstract: AFL (American Fuzzy Lop) is a powerful fuzzing tool that has discovered hundreds of real-world vulnerabilities. Recent efforts are seen to port AFL to a fuzzing Java program and have shown to be effective in Java testing. However, these tools require humans to write driver classes, which is not plausible for testing large-scale software. In addition, AFL generates files as input, making it limited for testing methods that process files. In this paper, we present JDriver, an automatic driver class generation framework for AFL-based fuzzing tools, which can build driver code for methods' processing files as well as ordinary methods not processing files. Our approach consists of three parts: a dependency-analysis based method to generate method sequences that are able to change the instance's status so as to exercise more paths, a knowledge assisted method to make instance for the method sequences, and an input-file oriented driver class assembling method to handle the method parameters for ordinary methods. We evaluate JDriver on commons-imaging, a widely used image library provided by the Apache organization. JDriver has successfully generated 149 helper methods which can be used to make instances for 110 classes. Moreover, 99 driver classes are built to cover 422 methods.

Keywords: driver class generation; java fuzzing; AFL; software security

1. Introduction

Fuzzing is an efficient and effective testing method by generating numerous inputs to reveal the vulnerabilities in the software-under-test (SUT). Recent efforts have been seen to port one of the most popular fuzzing tools AFL [1] to fuzzing Java programs. Different from binary programs, Java programs runs on Java Virtual Machine (JVM) and every public method can be tested directly with a driver class to provide basic runtime environments. Normally, the software developers write driver classes to test certain functions in the method-under-test (MUT). Their hand-written driver classes are filled with constant inputs that are only able to exercise limited paths, leaving a large part of the program not tested. AFL-based Java fuzzing tools [2,3] solve the coverage problem by making use of fuzzing techniques to generate inputs to exercise more paths. However, these tools fail to address the problem of automatic driver class generation. Both Kelinci [2] and JQF [3] rely on driver class written by testers to direct testing. This makes them not convenient for testing large-scale software. In addition, the AFL-based fuzzing tools employ files to store input, and this hinders the generation of the driver class for ordinary methods not processing files by requiring additional statements to converting the input file to correct variables.

Our goal is to build driver classes automatically for AFL-based Java fuzzing tools. Except for providing basic runtime environments for the MUT, the generated driver class should be able to mutate the status of the class instances as well as method parameters so as to exercise all paths in the MUT with the input file generated by the fuzzing tool. The status of the class instance is decided by the fields in the class, and some class fields can only be modified through invoking methods that change the method. Thus, for each MUT, the driver class should contain method sequences to change the instance's status, statements to prepare runtime time environments for the method sequences and statements to parse the data from the input file.

We face the following challenges: the first challenge is how to build method sequences for a target method. In Java, both member fields and method parameters can affect the branch statements. The member fields declared with keyword `private` and `protected` can only be modified by member methods. We need to know what member fields are accessed by the MUT and what methods can modify them so that we can build method sequences that are able to change the status of the instance. The second challenge is how to build instances to make the method sequences work. Instances need to access member fields and invoke member methods. To instantiate classes defined in SUT as well as built-in classes properly, we need to have the knowledge of what methods can be used to create instances. The last challenge is how to handle the input file. AFL-style fuzzing tools employ file to store input, so the methods in the driver class should process the input file, and prepare basic runtime instances for the target method with data extracted from the file.

We design and implement JDriver [4], an automatic driver class generation framework for AFL-based fuzzing tools. It employs dependency analysis to build method sequences that can modify method parameters as well as the field values. It collects knowledge and uses it to instantiate classes. JDriver supports making driver classes for general methods with input-file oriented driver class assembling methods, which can handle different method parameters properly. To summarize, we make the following contributions:

(1) **First study on automatic driver class generation for AFL-based Java fuzzing tools**. To the best of our knowledge, we are the first to study how to make driver classes for AFL-based fuzzing tools.
(2) **A novel approach to automatic driver class generation based on dependency analysis.** The approach consists of a dependency analysis based method to make method sequences, a knowledge assisted method to generate class instances and an input-file oriented method to assemble driver classes.
(3) **An open framework for driver class generation.** We implement JDriver, an open framework that aims to support driver class generation for different purposes. Evaluation results show we are able to generate 99 driver classes containing 422 driver methods for common-imaging.

The remaining paper is organized as below: Section 2 introduces related works. Section 3 describes our approach, Section 4 depicts the implementation, and Section 5 shows our evaluation results. We illustrate our thoughts on future work in Section 6, and conclusions are given in Section 7.

2. Related Work

In this section, we first introduce related work on fuzzing (Section 2.1), and then we describe the the research on Java fuzzing (Section 2.2). Finally, we summarize the works on automatic unit testing (Section 2.3).

2.1. Fuzzing

Based on how inputs are generated, fuzzing techniques can be categorized into two types: generation-based fuzzing and mutation-based fuzzing [5].

Generation-based fuzzing tools generate inputs with the knowledge of the input format. These tools are easy to implement, but their effectiveness relies on the accuracy of the input model. The earliest fuzzing tool [6] generates data randomly without any knowledge of the programs, thus it is limited to test software whose input is not formatted. To test software processing formatted data, sulley [7] and peach [8] provide ways for the testers to define specification for the formatted files and network protocols. BFuzz [9] builds the knowledge of HTML and XML files into fuzzer so that it can generate valid web pages to test the browsers. Patrice et al. [10] propose using machine learning algorithms to learn the format of file format, and their method has been used to test the PDF parser of Microsoft's Edge browser. Although the techniques are evolving, it is still hard to find an effective and general way to model input for various software.

Mutation-based fuzzing tools generate inputs by mutating the existing input. These tools don't rely that much on the knowledge of input, but they suffer from low code coverage because the mutation is performed on existing inputs. AFL [1] improves this by using coverage to direct fuzzing. It instruments the SUT with codes to record the branch coverage status and mutate the input that could exercise new branches to generate new input. AFL has achieved great success in bug finding and claims to have discovered hundreds of real-world bugs [1]. A number of variants have also come out to improve AFL. In order to mutate the seed efficiently, VUzzer [11] and Steelix [12] bring in program analysis to identify "magic bytes", and use them to generate new inputs. To avoid generating inputs exercising same paths, AFLFast [13] employs a Markov-chain to direct the mutation of inputs to less-frequency branches. Angora [14] targets each branch, it solves path constraints by its efficient byte-level tainting tracking algorithm, and mutates the tainted bytes with a gradient descent based searching method to generate inputs that exercise new paths. Except for improving AFL, projects to transplant AFL to other languages and other platforms have also been built [1]. The efficiency and scalability features have won AFL a lot of research focus.

2.2. Fuzzing Java Programs

Although fuzzing is widely used in testing binary programs, there are fewer tools for fuzzing Java programs. jFuzz [15] is a concolic whitebox fuzzing tool based on the explicit state software model checker Java PathFinder [16]. It begins with running a program with seed input in concolic execution to resolve the path constraints, then it solves the constraint and continuously generates modified input. Like most symbolic execution tools, it suffers from path explosion, so that it is not a good choice for fuzzing large software. JFuzz [17] is another effort, which defines several mutators to mutate existing inputs and generate new inputs. It uses metamorphic relations to detect failures. However, defining metamorphic relation is difficult, and this makes it even not practical to test ordinary software, not to mention large software.

Recent efforts [2,3] have been found to port AFL to fuzzing Java codes which make use of AFL to generate inputs for the driver class written by testers. Kelinci [2] provides the Java program with AFL-style branch counting codes to track the branch exercising status of the SUT. It acts as an agent that gets input from AFL, executes the input with the Java program and transfers execution results to AFL. Similarly, JQF [3] applies AFL to fuzzing Java code in a unit testing style. Both [2] and [3] have claimed to find real-world vulnerabilities. Unfortunately, these tools require human testers to write driver classes that hinders their applications.

2.3. Automatic Unit Testing

Automatic unit testing aims at generating small but effective testsuites to exercise all the paths in the program. Existing approaches can be categorized into two types: random-based generation and search-based generation.

Random-based generation methods build testcases by assembling methods and assigning values randomly. These tools are simple and easy to implement, but blind, which fail to achieve high code coverage. JCrasher [18] is among the earliest efforts. It builds testcases by choosing methods from

"parameter graph" to form method sequences and set random values for method parameters. Feedback directed random testing uses the execution information to direct future execution [19,20]. With the execution feedback, Randoop [19] builds inputs incrementally and classifies the inputs into redundant, illegal, contract-violating by defined contracts. In this way, the quality of the generated testcases is improved [19]. Lei et al. propose Guided Random Testing (GRT) [21], which uses program analysis to assist testcase generation. It employs static analysis to extract constants and dependency information. During the execution phase, the static information, together with dynamic analysis results, are used to create instances that may reach unexercised MUTs.

Search-based testing methods always maintain a goal, i.e., branch coverage to avoid making testcases blindly. The Search based Software Testing (SBST) Java Unit Testing Tool Contest [22] was started in 2012, which has attracted researchers to compete in this area. Evosuite [23] has been a winner for the last five contests. It employs an evolutionary algorithm to generate new testcases by merging previous testcases and minimize the size of testsuites according to the coverage metric. Sakti et al. [24] divide the test input generation problems input selecting proper methods to build class instance, calling methods to set the class under test into the proper state, and invoking target method call and searching the input space to build test input. Although these methods [19,23,24] have achieved great success in the SBST contest, their abilities to reveal real-world bugs are under question [25].

3. Approach

3.1. Overview

More than providing a basic runtime environment to invoke target method and determining the execution result, AFL-based fuzzing tools require their driver class to change the instance status as well as method parameters to exercise more paths with the file generated by them. To reach this goal, we need to build method sequences that are able to change the instance status and make class instances for the method sequences with data resolved from the input file generated by the fuzzing tools.

The method sequences are designed to explore all the branches in the method under test. In Java, the branch statements may contain values derived from method parameters as well as fields. Thus, we need to get the knowledge of which fields are accessed by the method. The method sequences should contain methods to modify the accessed fields. We put forward **dependency analysis based method sequences generation**, which employs static analysis to extract dependency information (Section 3.2) and build method sequences according to the dependency analysis results (Section 3.3).

The method sequences require class instances to make them work. Instances can be generated from various sources: constructors and factory methods are the most common ways. However, some special classes are not easy to find proper methods to get instances, e.g., built-in classes that require additional helper methods. We put forward **knowledge assisted instance generation**, which builds knowledge through collecting method information in the SUT as well as the user's programming knowledge. A Helper Class is also generated to store the methods used for creating class instances (Section 3.4).

AFL-based fuzzing tools generate files to store input data. This requires our driver method to interpret the file and make instance with the interpreted data. We propose **input-file oriented driver class assembling**, which operates differently on methods processing files and ordinary methods not processing files. For ordinary methods, we assemble all the method parameters and make statements to recover typed values for different method parameters (Section 3.5).

3.2. Dependency Information Extraction

In Java, public member fields can be modified directly, while private/protected member fields can only be modified by member methods. In addition, some methods may access fields directly, while some methods access fields indirectly through invoking other methods. Thus, method calls should be taken into consideration for dependency information extraction. In our approach, we extend method

call graph and define two directed graphs, *Access Graph* and *Modify Graph*, to store the dependency information. The vertexes are either methods or fields, and the edges in the graph are either method calls or accessing/modifying operation. Specifically, in Access Graph $AG = (V, E)$, edge from method V_{m_a} to method V_{m_b} indicates method m_a invokes a call to method m_b, edge from method V_{m_a} to field V_{f_x} indicates method m_a accesses the field f_x. In Modify Graph $MG = (V, E)$, the edge from field V_{f_x} to method V_{m_a} indicates that it is modified by method m_a, and the edge from method V_{m_a} to method V_{m_b} indicates that method m_a is invoked by method m_b.

We use static analysis to extract dependency information. Algorithm 1 illustrates how `Access Graph` and `Modify Graph` are built. The algorithm begins with initializing `accessGraph` and `modifyGraph` with the methods and fields in the class (line 2–11). Afterwards, it loops over all the methods and walks over all the instructions in methods to build the graphs (line 12 to 29). If instruction `inst` is a method call instruction, we resolve its call target `callee`, and add an edge from `method` to `callee` in `accessGraph`. Differently, we add edge from `callee` to `method` in `modifyGraph`. If instruction `inst` is a field related instruction, we retrieve its target `field`. An edge from `field` to `method` is added to `modifyGraph` if it is a field-write instruction (lines 21–22) while an edge from `method` to `field` is added to `accessGraph` if it is a field-read instruction (line 24–25). In this way, both the method call relationship and the relationship between method and field are written to the two graphs.

Algorithm 1 Analyzing dependency

```
1:  procedure ANALYZE(classnode)                          ▷ classnode is class under test
2:      accessGraph ← new AccessGraph()
3:      modifyGraph ← new ModifyGraph()
4:      for field in classnode.fields do
5:          accessGraph.addVertex(field)
6:          modifyGraph.addVertex(field)
7:      end for
8:      for method in classnode.methods do
9:          accessGraph.addVertex(method)
10:         modifyGraph.addVertex(method)
11:     end for
12:     for method in classnode.methods do
13:         instrs ← method.instrs                ▷ instrs stores all the instructions in the method
14:         while instrs is not empty do
15:             inst ← instrs.get(0)
16:             if inst is method invoke then
17:                 callee ← inst.getCallee()                              ▷ resolve callee
18:                 accessGraph.addEdge(method, callee)
19:                 modifyGraph.addEdge(callee, method)
20:             else if inst is field-write operation then
21:                 field ← inst.getTarget()                    ▷ field is the target of inst
22:                 modifyGraph.addEdge(field, method)
23:             else if inst is field-read operation then
24:                 field ← inst.getTarget()
25:                 accessGraph.addEdge(method, field)
26:             end if
27:             instrs.remove(inst)                        ▷ remove inst from the instruction set
28:         end while
29:     end for
30: end procedure
```

Regarding the graph theory, we come to the following two theorems:

Theorem 1. *Method m_a accesses field f_x if and only if the two nodes V_{m_a} and V_{f_x} are connected in* `Access Graph`.

Proof of Theorem 1. There are two situations in which method m_a accesses field f_x: direct and indirect access. In the direct situation, according to our definition of `Access Graph`, the direct access will be represented as an edge from V_{m_a} to V_{f_x} which means V_{m_a} and V_{f_x} are connected directly. In the indirect

situation, method m_a accesses field f_x indirectly through method calls, which means that there are call sequences from method m_a to method m_z and method m_z access field f_x. Method m_z accesses field f_x directly, so V_{m_z} and V_{f_x} are connected (1). The call sequences from m_a to m_z indicate that there is a path from V_{m_a} to V_{m_z} which indicates that V_{m_a} and V_{m_z} are connected (2). Combining (1) and (2), V_{m_a} and V_{f_x} are connected. Thus, if method m_a accesses field f_x, the two nodes are connected in the Access Graph. Reversely, if V_{m_a} and V_{f_x} are connected, there is a path between V_{m_a} and V_{f_x}. If the length of the path is 1, it means the method m_a access field f_x directly. If the length is bigger than 1, there are more than two vertexes in the path, namely V_{m_a}, V_{m_b}...V_{m_z}, V_{f_x}. The vertex V_{m_z} and V_{f_x} are connected directly meaning method m_z access field f_x directly (3). The path from V_{m_a} to V_{m_z} indicates that there are call sequences from method m_a to m_z(4). Combining (3) and (4), we get that method m_a accesses field f_x indirectly. Thus, if V_{m_a} and V_{f_x} are connected, method m_a access field f_x. □

Theorem 2. *Method V_{m_a} can modify field V_{f_x} if and only if the two nodes are connected in* `Modify Graph`.

Theorem 2 can be proved in the same convention of Theorem 1. Theorem 1 explains how we can find the member fields accessed by the given target. While Theorem 2 provides us with a way to find the member method that can modify target member fields. We use $AccesssSet_m$ to represent the set of fields that are accessed by method `m` and $ModifySet_f$ to be the set of methods that can modify field `f`. Actually, $AccesssSet_m$ is made up of all the field nodes that are connected with the specified method m and $ModifySet_f$ is made up of all the method nodes that are connected with field f.

3.3. Method Sequence Building

The method sequences are used to modify the status of the instance. Apart from invoking methods to change target fields, the public fields can also be changed by assigning values directly. Thus, we extend method sequences to include *field* to indicate that the field can be modified directly.

We build method sequences on dependency information. For the MUT, we can get its accessed fields set with Access Graph, and the Modify Graph assists us with retrieving a set of methods that can modify the target field. Algorithm 2 illustrates how we build method sequences with Access Graph and Modify Graph. An empty array `ms` is initialized to store method sequences. For the given method `mut`, we first resolve the `mut`'s accessSet $accessSet_{mut}$ (line 2). Then, we build the method sequence incrementally by iterating over the $accessSet_{mut}$ (line 4–10). For every field in the $accessSet_{mut}$, we retrieve its $modifySet_{field}$ and add chosen items to `ms` (line 5–9). The item is returned by the `select` procedure, which is used to define the policy of how we select methods to build method sequences.

Policy to select method. As static analysis is conservative, the extract dependency information may not be accurate. Thus, we need to implement different policies to get better performance. In our implementation, we design a policy to prioritize the field item and select the method whose method parameters are simplest to make. In the procedure `select`, we first check if the `field` is public and its type is primitive. If it is, we return it directly. If not, we examine whether the existing methods in the method sequences can modify the target field (lines 16–20). If such methods exist, null is returned to avoid duplicate modification. If not, we sort the method in the $modifySet_{field}$ (line 21) and return the first method (line 22). In our case, the methods are sorted by the simpleness of method parameters, which is measured by the number of primitive parameters in the method.

3.4. Knowledge Assisted Instance Generation

Normally, instances are created by the constructor of the specified class. In addition, factory methods that make instance as its return value can also be used to generate instances. However, for built-in classes provided by The Java Platform, Standard Edition (Java SE), e.g., `String`, it is not easy to find proper constructors or factory methods, they require additional methods to make instances. We name these methods to create instances as *knowledge*. Our knowledge assisted instance generation method builds knowledge through collecting related methods for the SUT as well as making methods

from the users' knowledge. As instance are frequently used in driver class, we build a `Helper Class` to store all the instance generation methods.

Algorithm 2 Building method sequences

```
1:  procedure BUILDMETHODSEQUENCE(mut,accessGraph, modifyGrph)▷ mut is method under test,
    accessGraph and modifyGraph are used to store dependency information
2:      ms ← newArrayList()
3:      accessSet_mut ← accessGraph.getAccessSet(mut)
4:      for field in accessSet_mut do
5:          modifySet_field ← modifyGraph.getModifySet()
6:          item ← select(field, modifySet_field, ms)
7:          if item is not null then
8:              ms.add(item)
9:          end if
10:     end for
11:     return ms
12: end procedure
13: procedure SELECT(field, modifySet, ms)
14:     if field is public and primitive typed then return field
15:     end if
16:     for method in ms do
17:         if modifySet.contains(method) then
18:             return null
19:         end if
20:     end for
21:     modifySet ← sort(modifySet)
22:     return modifySet.get(0)
23: end procedure
```

Collecting instance generation methods. We define a *type table* to store factory methods and class constructors for the SUT. The type table uses class type as the key, and the value of the key is a set of methods. We build type table by walking all the methods in the SUT as illustrated in Algorithm 3. For each method in the SUT, we first resolve its return type `returnType` (line 5). Afterwards, we decide if `returnType` has already existed in the type table. If it has, we add it to the corresponding method set (line 7). If it has not, we create a new set, and put it into the new set, and add an item to the type table (lines 9–11). Apart from SUT, we also build type table for its dependent libraries.

Algorithm 3 Building type table

```
1:  procedure BUILDTYPETABLE(sut)                    ▷ sut is the software under test
2:      typeTalbe ← new TypeTable()
3:      for class in sut.classes do
4:          for method in class.methods do
5:              returnType ← getReturnType(method)
6:              if typeTable.keySet().contains(returnType) then
7:                  typeTable.get(returnType).add(method)
8:              else
9:                  newSet ← new Set()
10:                 newSet.add(method)
11:                 typeTable.put(returnType, newSet)
12:             end if
13:         end for
14:     end for
15: end procedure
```

Knowledge for built-in classes. We add knowledge for classes provided by the Java Platform. Specifically, we cover most of the classes defined in the `java.util` package, which contains the container classes such as `Set`, the `java.lang` package, which defines classes that are fundamental to the design of the Java programming language such as `String`, and the classes in `java.io` package, which contains classes to handle system input/output [26]. Figure 1 shows two sample knowledge

methods. Method `get_String` returns a `String` instance which comes from the input parameter `arg0`. Method `get_File` returns a `File` instance, which is created by the `new` expression with the method parameter`arg0`.

```
1  public static String get_String(String arg0) throws Exception {
2    return arg0;
3  }
4  public static File get_File(String arg0) throws Exception {
5    return new File(arg0);
6  }
7
```

Figure 1. Sample knowledge method.

Building Instance Helper Class. Instantiating class instances are frequently used during testing. To avoid generating it repeatedly, we build an Instance Helper Class to handle the generation of instances. Algorithm 4 shows the building process for Instance Helper. It starts with initializing `typeSet` to include all the buildable classes (lines 2–7). Afterwards, it builds instance helper methods with `buildInstanceHelper` (line 8). When the processing finishes, `InstanceHelperClasses` assembles all the methods, and adds miscellaneous codes to build a compilable Helper Class (line 9). In the `buildInstanceHelper` method, it initializes `unprocessed` as a copy of `typeSet`, and then it walks over all the types in `typeSet` to build helper methods (lines 13–18) with `buildType`. Procedure `buildType` builds helper methods for each type and returns the number of generated helper methods. If `buildType` builds more than one helper method successfully, then the `type` is removed from `unprocessed` (line 16). Since some class constructors rely on other classes, it is necessary to build helper class recursively to cover these classes (lines 19–20). In `buildType`, it firstly resolves the `methodSet` for the given type `type`. Then, it iterates over all the methods in the `methodSet` to build helper method (lines 28–34). Each time a helper method is generated, it is appended to `InstanceHelperClasses`. Note that, in order to simplify the method inputs, our helper method only employs primitives or `String` as method parameters.

3.5. Input-file Oriented Driver Class Assembling

For a class-under-test (CUT), we build driver method separately for each public method and assemble the driver methods into a driver class. AFL-based fuzzing tools generate files to store the input data. If the MUT processes files directly, we can pass the file directly as a method parameter. However, in most cases, the methods don't do so. For these methods, their driver methods need to process the input file and present the data to make variables for the methods. Our input-file oriented driver class assembling method works differently on ordinary methods not processing file and methods processing file.

Testing if method processes file. As file processing methods use built-in classes such as `File` to handle files, we design the following heuristic to determine whether the method processes files directly: (1) the method parameters contain file related class instances such as `File`; and (2) there is a `String` typed method parameter which flows to a file opening method.

Ordinary methods not processing files. Our driver method starts with extracting the input file to a byte array, and then it resolves the values for the method parameters sequentially from the byte array. Algorithm 5 shows the building process. Method `makeStatements` begins with declaring a variable `position` to mark the position in the byte array, and it iterates over the items in the method sequences to make statements. If the item is a field, it makes an assigning statement directly with `makeField` (lines 35–40). If it is a method, it makes statements to declare variables as well as the statement to invoke the method with `makeMethod` (lines 11–34). For each method parameter, method `makeMethod` applies different rules according to their types. There are two categories of types in Java: primitive types and reference types. **(1) Primitive types**. Primitive typed data has fixed sizes, and we can make it directly from the input bytes. Method `makeVariableStatement` generates statements like

this: `int a = Helper.getInt(inputs, position)`. **(2) Reference types.** Class types and array types are two reference types. For class types, we firstly get a helper method from `InstanceHelperClasses` (line 18). If `helperMethod` is not null, we make statements for the helper method (line 20). For array types, if it is primitive array, we resolve its element type (line 25), and its array length (line 26). If the length is not specified, we will use a random number to replace it. We make statements with its element type `etype` (line 27). If we can't get a proper *helperMethod* or the array is not a primitive array, we make statements with built-in knowledge (lines 21, 28). Method `makeMethod` ends with making statements to invoke the *method* (line 32), and returning the *position* to avoid retrieving bytes from the same position (line 33). Method `makeField` works similarly as the primitive type in `makeMethod`.

Algorithm 4 Building instance helper

```
1:  procedure BUILD(typeTable)                          ▷ typeTable is the type table generated for SUT
2:      typeSet ← new Set()
3:      for type in typeTable.keySet() do
4:          if isTypeBuildable(type) then                          ▷ test if the type buildable
5:              typeSet.add(type)
6:          end if
7:      end for
8:      buildInstanceHelper(typeTable, typeSet)
9:      InstanceHelperClasses.write()                  ▷ save the generated helper methods to file
10: end procedure
11: procedure BUILDINSTANCEHELPER(typeTable, typeSet)   ▷ typeSet is the set of class to build
12:     unprocessed ← typeSet.copy()
13:     for type in typeSet do
14:         result ← buildType(type, typeTable)
15:         if result > 0 then
16:             unprocessed.remove(type)
17:         end if
18:     end for
19:     if repeatTest(typeSet, typeSetCopy) then                ▷ test if future test is necessary
20:         unprocessed ← buildInstanceHelper(typeTable, unprocessed) ▷ building unprocessed types
21:     else
22:         return typeSet
23:     end if
24: end procedure
25: procedure BUILDTYPE(type, typeTable)                ▷ type is the type for building
26:     methodSet ← typeTable.get(type)
27:     rtn ← 0
28:     for method in methodSet do
29:         helperMethod ← buildHelperMethod(type, method)      ▷ build a helper method for type
30:         if instanceMethod not null then
31:             InstanceHelperClasses.add(helperMethod) ▷ save helperMethod to InstanceHelperClass
32:             rtn += 1
33:         end if
34:     end for
35:     return rtn
36: end procedure
```

Method processing files. For these methods, we identify which method parameter is used to specify the filename, and then present the file path to the method directly. If there exist other method parameters, we use random generators to generate values.

Algorithm 5 Making statements to recover method parameters

```
1:  procedure MAKESTATEMENTS(ms)                    ▷ ms are the generated method sequences.
2:      position ← 0                                ▷ position is used to mark the position the byte array
3:      for item in ms do
4:          if item is method then
5:              position = makeMethod(item, poistion)
6:          else
7:              position = makeField(item, position)
8:          end if
9:      end for
10: end procedure
11: procedure MAKEMETHOD(method, position)
12:     inputTypes ← method.getInputs()
13:     for i in inputTypes do
14:         if i is primitive then
15:             makeVariableStatement(i, position)
16:             position += Type.getSize(i)
17:         else if i is class type then
18:             helpMethod ← InstanceHelperClasses.getMethod(i)
19:             if helpMethod is not null then
20:                 position = makeMethod(helpMethod, poistion)    ▷ make statements for method
21:             else makeStatementWithKnowledge(i);
22:             end if
23:         else if i is array type then
24:             if i is primitive array then
25:                 etype ← i.getElementType            ▷ etype is the element type of the array
26:                 size ← getArrayLength(i)
27:                 position = makeArray(etype, size, poistion)
28:             else makeStatementWithKnowledge(i);
29:             end if
30:         end if
31:     end for
32:     makeMethodStatement()
33:     return position
34: end procedure
35: procedure MAKEFIELD(field, position)
36:     type ← field.getType()
37:     makeVariableStatement(type, position)
38:     position += Type.getSize(type)
39:     return position
40: end procedure
```

4. Implementation

Our automatic driver class generation approach has been implemented into JDriver with around 5000 lines of Java code. It uses the general purpose Java bytecode analysis framework ASM [27] to perform dependency analysis as well as extract class information. JDriver takes Java program as input, and produces driver classes in the following steps:

(1) **preprocessing**. JDriver starts with analyzing the SUT to collect methods used for instance generating and analyzing the attributes of the classes.
(2) **extracting dependency information**. For each CUT, we extract dependency information, *Access Graph* and *Modify Graph* according to the algorithm described in Section 3.2.
(3) **building method sequence**. For each MUT, we resolve its $accessSet_{mut}$ and build method sequences that are able to change the values in the $accessSet_{mut}$.
(4) **building instance helper class for instance generation**. For the SUT, we build knowledge by collecting constructors and factory methods in the SUT. For built-in classes, we design additional helper methods. With the knowledge, we produce a `Helper Class` to save methods for class instantiating.

(5) **assembling driver class**. Finally, with the method sequences and instance generation helper methods, we build statements to operate on the input file and declare variables for the method sequences.

5. Evaluation

We first demonstrate our driver class generation with a simple example in Section 5.1. Then, we work on the widely used image processing library `commons-imaging` [28] to evaluate instance generation (Section 5.2) and driver class generation (Section 5.3).

5.1. Simple Example

The example CUT. We use the simple example to illustrate how our approach works. Figure 2 shows the source code of the CUT. The class `AClass` has three member fields: `a`, `b` and `c`. Fields `a` and `c` are public while `b` is private which can only be modified through public method `setB` (lines 15–18). In method `foo` (lines 18–25), both field `a` and `b` are involved in the branch statement in line 19. To exercise all the input space, the driver code should be able to change the values of fields `a` and `b` as well the method parameter `x`.

```
1  public class AClass {
2    public int a;
3    private int b;
4    public final int c = 10;
5    public AClass() {
6        a = b = 0;
7    }
8    public AClass(int a, int b) {
9        this.a = a;
10       this.b = b;
11   }
12   private void setA(int x ) {
13       a = x;
14   }
15   public void setB(int y) {
16       b = y;
17   }
18   public int foo(int x) {
19       if(a+b > 0)
20           return 10;
21       else {
22           setA(x);
23           return 20 + a;
24       }
25   }
26 }
```

Figure 2. Sample class under test.

The driver class. As shown in Figure A1, JDriver generates a driver class `AClassTest` for the CUT which is made up the `main` method to control the testing and driver method `foo_test` to test method `foo`. The entry method `main` (lines 43–48) employs an array of `String` as input and `args[0]` is set to be the filename of the input. Method `foo_test` (lines 13–42) takes a `String` to represent filename. The file is extracted to a byte array `data` (lines 14–18), which is divided into `fields` and `inputs` array for building instances for method sequences and inputs of the target method separately (lines 19–20). Afterwards, the parameters for building `AClass` instance are prepared (lines 25–28), and an instance `cut` is generated (line 29) by the constructor. For public member field `a`, we modify it through assigning operation (line 30). The method parameter is recovered in lines 34–35, and the target method `foo` is tested in `try-catch` block to catch runtime exceptions (lines 37–41). Note that the generated method sequence for `foo` are lines 29 and 30. Our method sequences contain the statement (line 30) to change `a`, but don't contain separate statements to change `b`. This is because we implement a selecting method policy to ignore the fields that are changed by the constructor if the fields could not be assigned values directly.

5.2. Evaluating Instance Generation

JDriver builds a Helper Class named `InstanceHelper.java` to save the helper methods for instance generation. JDriver successfully generates 149 helper methods for 110 classes. Figure 3 illustrates the Instance Helper Method for class `ByteSourceFile`, whose constructor uses a `File` object. In line 1, a `File` instance is generated by the helper method `get_File`, which makes a `File` instance from a `String` instance.

```
1  public static ByteSourceFile get_ByteSourceFile(String arg0_tmp0)  throws Exception {
2    File arg0 = StandardLibraryHelper.get_File(arg0_tmp0);
3    return new ByteSourceFile(arg0);
4  }
5
```

Figure 3. Sample instance helper method.

5.3. Evaluating Driver Class Generation

JDriver successfully builds 99 driver classes for 422 methods, 60 of which are methods processing file. Figure 4 shows the driver method generated for method `convertYCbCrtoRGB`, which takes three integers as the method parameter. Our dependency analysis detects there are no fields involved in the `convertYCbCrtoRGB`, so we only need declare variables for method parameters. Similar to the driver class in Figure A1, our driver method begins with reading data in the file and forming the byte array `data` (lines 2–5). The `input` array is later resolved to make variables (line 8). Afterwards, we declare three int variables and assign them with the values resolved from the `inputs` (lines 13–19). To avoid naming conflicts, each variable is named after the name of its method. These variables are later used as parameters to the `convertYCbCrtoRGB`. In this way, the input file generated by the AFL-based fuzzing tool is converted to method parameter of ordinary methods. The more input files it generates, the higher possibility for the driver method to exercise all paths.

```
1     public static void convertYCbCrtoRGB_test(String filename) {
2        byte[] data = Helper.readBytes(filename);
3        if(data == null) {
4            System.out.println("Fail to read bytes from file!, quit!");
5            return;
6        }
7        byte[] fields = Helper.splitFields(data, 0);
8        byte[] inputs = Helper.splitInput(data, 0,12);
9        int position = 0;
10       byte[] tmp;
11
12       //calling the mut convertYCbCrtoRGB
13       position = 0;
14       int convertYCbCrtoRGB_0 = Helper.get_int(inputs, position);
15       position += 4;
16       int convertYCbCrtoRGB_1 = Helper.get_int(inputs, position);
17       position += 4;
18       int convertYCbCrtoRGB_2 = Helper.get_int(inputs, position);
19       position += 4;
20       try{
21           PhotometricInterpreterYCbCr.convertYCbCrtoRGB(
22                   convertYCbCrtoRGB_0, \
23                   convertYCbCrtoRGB_1, \
24                   convertYCbCrtoRGB_2);
25       } catch (Exception e) {
26           e.printStackTrace();
27       }
28    }
29
```

Figure 4. Sample driver method for the processing file method.

6. Discussion and Future Work

Instance Generation. Although JDriver has generated hundreds of driver methods for `commons-imaging`, it fails to make correct instances for the following classes: (1) interface classes. Class `ImageFormat` is an interface, which should be initialized through classes that has implemented this interface. However, JDriver has no knowledge for generating class instance for interface, thus it fails on interface classes. (2) classes containing types not covered in knowledge base. The constructor of `ByteSourceArray` has `String` and `byte[]` as method parameters, but we missed the `byte[]` type in our helper class. This makes JDriver fail to generate instance for class `ByteSourceArray`. (3) classes whose constructor involves multiple `String` objects. The constructor method of class `ByteSourceInputStream` takes an `InputStream` and a `String` typed parameters. Our algorithm detects that the helper method `get_InputStream` can be used to make instance for class `InputStream` and method `get_String` can be used to make `String` instance. However, it skips the methods that use multiple `String` objects because the driver code only has one String object as input, so it fails to make `ByteSourceInputStream` instances. In addition, some instances we make are meaningless. For example, method `getBufferedImage` in class verb|JpegImageParser| accepts a `HashMap` instance and uses it to store items. Normally, we need to initialize an non-empty `HashMap`. However, if we assign a `null` to that parameter, it may continue the execution but won't reach our target branch. To summarize, we need more smart knowledge to build correct instance. In the future, we will continue working on: covering advanced Java features like subclassing, interfaces into knowledge base, covering more built-in classes, restructuring helper methods, and learning knowledge from the source code of the SUT.

Fuzzing Scheduling. Actually, fuzzing every method in Java programs is neither plausible nor necessary. (1) indirectly accessible methods should be skipped. Java puts access control attributes in every method, and methods declared with `public` and `private` can't be accessed directly. In addition, abstract classes can't be instantiated. These codes should be skipped. (2) methods that don't produce errors should be excluded. In Java, some methods are used to do simple work, they have single branches and never throw exceptions. For example, the `getter` methods only contain a single statement to return a field value; they can never produce exceptions. (3) methods that have been exercised are not necessary to test alone. Some methods are repeatedly implemented when fuzzing other methods; testing these methods is useless and wastes a lot of time. A proper way to schedule methods for fuzzing is to use program analysis techniques to identify the methods listed above. Static analysis can help identify the methods that have no branches and won't produce exceptions. Dynamic analysis techniques could track the execution of the methods and find out what methods have already been used.

7. Conclusions

Building sound driver classes for AFL-based Java fuzzing tools can improve their fuzzing ability and efficiency. In this paper, we study the automatic driver class generation problem for these tools. Our automatic driver class generation method employs dependency analysis to analyze what fields are accessed by the method, and then construct method sequences to mutate the values of the accessed fields so that the generated method sequences are able to mutate the status of the class instances. We design a knowledge assisted instance generation method to make instances for classes from various sources. In order to allow fuzzing for ordinary methods, our input-file oriented driver class generation method rebuilds method interfaces and generates statements to declare variables for parameters used in the method sequences. We implement our approach in JDriver. To the best of our knowledge, we are the first to study this problem, and JDriver is the earliest automatic driver class generation framework for AFL-based Java fuzzing tools. We evaluate JDriver on real-world library commons-imaging, and it has successfully generated 99 driver classes to cover 422 methods; this proves that JDriver is effective in driver class generation for AFL-based fuzzing tools.

Author Contributions: Methodology, Software, Writing, Z.H.; Methodology, Supervision, Review & Editing, Y.W.

Funding: This research is supported by NSFC No.61472439, the National Natural Science Foundation of China.

Acknowledgments: We'd like to express our appreciation to anonymous editors.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. The Generated Sample Driver Class

Figure A1 illustrates the driver class we generated for sample class in Figure 2.

```
import java.util.Arrays;
import java.nio.ByteBuffer;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;
import java.lang.reflect.Method;
//Driver code for Class AClass;
public class AClassTest {
    public static void foo_test(String filename) {
        byte[] data = Helper.readBytes(filename);
        if(data == null) {
            System.out.println("Fail to read bytes from file!, quit!");
            return;
        }
        byte[] fields = Helper.splitFields(data, 12);
        byte[] inputs = Helper.splitInput(data, 12,16);
        int position = 0;
        byte[] tmp;

        //recovery parameters for AClass
        int AClass_0 = Helper.get_int(fields, position);
        position += 4;
        int AClass_1 = Helper.get_int(fields, position);
        position += 4;
        AClass cut = new AClass(AClass_0, AClass_1);
        cut.a = Helper.get_int(fields, position);
        position += 4;

        //calling the mut foo
        position = 0;
        int foo_0 = Helper.get_int(inputs, position);
        position += 4;
        try{
            cut.foo(foo_0);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        if(args.length == 1) {
            foo_test(args[0]);
        }
    }
}
```

Figure A1. Driver class for class defined in Figure 2.

References

1. American Fuzzy Lop. Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 21 August 2018).
2. Kirsten, R.; Luckow, K.; Păsăreanu, C.S. POSTER: AFL-based Fuzzing for Java with Kelinci. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; ACM: New York, NY, USA, 2017; pp. 2511–2513. [CrossRef]
3. JQF. Available online: https://github.com/rohanpadhye/jqf (accessed on 21 August 2018).
4. JDriver. Available online: https://github.com/qorost/jdriver.git (accessed on 1 October 2018).

5. Li, J.; Zhao, B.; Zhang, C. Fuzzing: A survey. *Cybersecurity* **2018**, *1*, 6. [CrossRef]
6. Miller, B.P.; Fredriksen, L.; So, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* **1990**, *33*, 32–44. [CrossRef]
7. The Sulley Fuzzer. Available online: https://github.com/OpenRCE/sulley (accessed on 21 September 2018).
8. The Peach Platform. Available online: https://www.peach.tech/products/peach-fuzzer/peach-platform/ (accessed on 21 September 2018).
9. The BFuzz Platform. Available online: https://github.com/RootUp/BFuzz (accessed on 21 September 2018).
10. Godefroid, P.; Peleg, H.; Singh, R. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana-Champaign, IL, USA, 30 October–3 November 2017; IEEE Press: Piscataway, NJ, USA, 2017; pp. 50–59.
11. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
12. Li, Y.; Chen, B.; Chandramohan, M.; Lin, S.W.; Liu, Y.; Tiu, A. Steelix: Program-state based binary fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; ACM: New York, NY, USA, 2017; pp. 627–637.
13. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**. [CrossRef]
14. Chen, P.; Chen, H. Angora: Efficient Fuzzing by Principled Search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 21–23 May 2018; pp. 711–725. [CrossRef]
15. Jayaraman, K.; Harvison, D.; Ganesh, V.; Kiezun, A. jFuzz: A Concolic Whitebox Fuzzer for Java. In Proceedings of the First NASA Formal Methods Symposium, Moffett Field, CA, USA, 6–8 April 2009; pp. 121–125.
16. NASA Java PathFinder. Available online: http://javapathfinder.sourceforge.net (accessed on 21 August 2018).
17. Zhu, H. JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods. In Proceedings of the 2nd International Conference on Trustworthy Systems and Their Applications, Hualien, Taiwan, 8–9 July 2015; pp. 8–15.
18. Csallner, C.; Smaragdakis, Y. J. Crasher: An automatic robustness tester for Java. *Softw. Pract. Exp.* **2004**, *34*, 1025–1050. [CrossRef]
19. Pacheco, C.; Ernst, M.D. Randoop: Feedback-directed Random Testing for Java. In Proceedings of the Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, Montreal, QC, Canada, 21–25 October 2007; ACM: New York, NY, USA, 2007; pp. 815–816. [CrossRef]
20. Godefroid, P.; Klarlund, N.; Sen, K. DART: Directed Automated Random Testing. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; ACM: New York, NY, USA, 2005; pp. 213–223. [CrossRef]
21. Ma, L.; Artho, C.; Zhang, C.; Sato, H.; Gmeiner, J.; Ramler, R. Grt: Program-analysis-guided random testing (t). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 212–223.
22. 10th International Workshop on Search-Based Software Testing. Available online: http://sbst2017.lafhis.dc.uba.ar (accessed on 21 August 2018).
23. Fraser, G.; Arcuri, A. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; ACM: New York, NY, USA, 2011; pp. 416–419. [CrossRef]
24. Sakti, A.; Pesant, G.; Guéhéneuc, Y.G. Instance generator and problem representation to improve object oriented code coverage. *IEEE Trans. Softw. Eng.* **2015**, *41*, 294–313. [CrossRef]
25. Shamshiri, S.; Rojas, M.; Fraser, G.; Mcminn, P.; Arcuri, A. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015.
26. Java Platform, Standard Edition & Java Development Kit Version 9 API Specification. Available online: https://docs.oracle.com/javase/9/docs/api/index.html?overview-summary.html (accessed on 21 August 2018).

27. ASM. Available online: http://https://asm.ow2.io (accessed on 21 August 2018).
28. commons-imaging. Available online: https://commons.apache.org/proper/commons-imaging (accessed on 21 August 2018).