Software Reliability Coursework 1
Implement a SAT fuzzer

Yuzhou Zhuang Buzhen Ye Hanbin Qi

Spring 2020

Contents

1 Introduction **2**

2 How to Run **2**

3 Approaches of Fuzzing **2**
3.1 Undefined Behaviour Mode 2
3.1.1 Generating Inputs 2
3.1.2 Case Evaluation (Eviction Policy) 3
3.2 Functional Error Mode 4
3.2.1 Generating Follow-up Test Cases 4
3.2.2 Case Evaluation 5

4 Experimented design ideas **5**

5 Details of optimizations **5**

1 Introduction

In this project, we implemented a black-box DIMACS fuzzer with Python. This fuzzer has two modes, naming undefined behaviour (specified by `ub`) mode and functional error (specified as `func`) mode. In `ub` mode, the fuzzer will try to generate inputs (not necessarily valid) that cause undefined behaviours. In `func` mode, the fuzzer will try to generate follow-up test cases by using the metamorphic method, to find cases where the solver produces incorrect output.

2 How to Run

You need to put our files (`func_generator.py, ub_generator.py, ub_main.py, fuzz-sat`) into the same folder as the solver, and make sure the terminal is in the same folder as the solver. We do not have `build.sh` executable bash script that builds the fuzzer since we use Python. Our fuzzer can be simply invoked as follows:

```
./fuzzsat /path/to/SUT /path/to/inputs mode
```

Please note that `seed` is not needed for our fuzzer.

3 Approaches of Fuzzing

3.1 Undefined Behaviour Mode

3.1.1 Generating Inputs

In `ub` mode, we implemented 4 different methods to generate inputs (see below) and they are chosen randomly with different probabilities.

1. `create_trash_input`
2. `create_dimacs_input`
   (a) `first_line_mutaion`
   (b) `random_line_mutaion`
   (c) `generate_random_number_cnf`

For the first method `create_trash_input`, we generate 7 representative pre-defined trash inputs (dumb fuzzing). Those inputs are invalid files using random chars and overflowed integers that would essentially cause the solver to crash. They are listed below:

- Simply an empty file
- *p cnf* - valid prefix with no follow-up
- *p cnf MAX_INT+1 MAX_INT+1* - header with overflowed integers
- `string.printable` - simply the string of all the printable characters
- *p cnf 10 10* - valid header with no contents
- Valid header with `string.punctuation` as the follow-up
- Valid header with `string.printable` as the follow-up

During the first six iterations of the main loop, our fuzzer will call the `create_trash_input` function and feed the SUT with the above inputs respectively to crash the solver.

The second method `creat_dimacs_input` will be called after the first six iterations.It is a mutation-based smart fuzzing and it has three different strategies:

- The `first_line_mutaion` method (10% chance to be applied) takes the first line (header) of a well-formed DIMACS-format file (randomly selected from the `/path/to/inputs`) and randomly plus/minus 1 to the number of variables or the number of clauses.
- The `random_line_mutaion` method (60% chance to be applied) picks up a random non-header line (clause) from the randomly-selected well-formed DIMACS-format file. And choose (with equal chance) one from the following two rules for mutation:
  - Change each number except the ending zero to `sys.maxsize+1` or `num_of_variable+1` or a random punctuation (with equal probability).
  - Delete the ending zero or change it to a random punctuation (with equal probability).
- The `generate_random_number_cnf` method (30% chance to be applied) randomly picks a number of variables and a number of clauses to randomly generate a DIMACS-like inputs, it follows the format of DIMAC (for each variable in each clause, pick a arbitrary number within the declared range as its notation and add the minus sign randomly) but the content is essentially invalid (i.e The number of variables used is different from the number that declared in the header).

3.1.2 Case Evaluation (Eviction Policy)

After generating the input, our Python script will run the SUT with the generated input by using the `runsat.sh` script. Then the `stderr` and the `stdout` streams will be redirected and stored in two variables for evaluation. Since we can only store 20 cases for submission and we want our cases to include as many different types of Undefined Behaviors as possible and also focus on UBs that triggered by many distinct locations, we come up with the following steps for case replacement:

1. A buffer `ubts_buffer` of size 20 is maintained to store all the evaluation results of currently selected cases.
2. We make a dictionary of regular expressions to match all the potential errors detected by ASan and UBSan (given they are the only two being instrumented in the `ub` mode and their corresponding lists of detected issues can be easily found online).
3. After each iteration, the redirected `strerr` stream (containing issues reported by the sanitizers and raised during runtime) will be used for evaluation by matching its lines to each regular expressions we had respectively. To represent the evaluation result, we initialise a list `cur_status` of 14 (the number of checked errors/regular expressions) zeros, a counter `cur_counter` for counting the total matches (same errors counted multiple times here) and a boolean `dup_flag` for indicating whether it is a duplicate case. Whenever a regular expression is matched, the corresponding zero (index as the key of the dictionary) of the `cur_status` list will be flipped to 1 and the counter will plus one as well.
4. After getting the evaluation result of the current case, we will compare the result to all the stored results in the `ubts_buffer` and apply the following policies:
   - If the `cur_status` is unique among all the stored status, which means the current case is unique in respect of types of errors raised. Then we will select a duplicate case (`dup_flag` set) from the buffer and use the current case to replace it. However, if all the cases in our buffer are unique already, the current case will be dropped.
   - If the `cur_status` is not unique, we will compare the `cur_counter` to the counter of the same-status case in the buffer and the one with a larger counter (more errors, thought the types is same) will be stored.

The main loop of our fuzzer (in `ub` mode) will keep running until being manually interrupted. For all the cases stored in the `ubts_buffer` buffer, the used input file will be stored in a folder called

`fuzzed-tests` (created by our fuzzer as required) under the `/path/to/SUT` (essentially the currently directory since we are using the `runsat.sh` script) and the corresponding error information (`stderr`) and solver output (`stdout`) will be stored in `fuzzed-tests-logs` and `fuzzed-tests-outpus` respectively for checking.

3.2 Functional Error Mode

3.2.1 Generating Follow-up Test Cases

Our `func` mode fuzzer mutates the existing input files to generate follow-up tests in a blind fashion. These generated tests are also valid inputs. We define following 5 basic mutation strategies:

(1) swap between clauses

(2) swap internal clauses

(3) add clauses

(4) add one trivial SAT clause

(5) add two trivial UNSAT clauses

And three expected effects could be resulted by mutations above:

A. SAT→SAT, UNSAT→UNSAT

B. SAT→UNKNOWN, UNSAT→UNSAT

C. SAT→UNSAT, UNSAT→UNSAT

In addition to basic mutation strategies, we also perform different combinations of these to generate more sophisticated mutations. The following chart demonstrates the final strategies we use:

Number	Strategy	Expectation
0-5	(1)	A
6-11	(2)	A
12-17	(3)	B
18	(4)	A
19	(5)	C
20-25	(6) = (2) + (1)	A
26-31	(7) = (3) + (1)	B
32-37	(8) = (3) + (2)	B
38-43	(9) = (1) + (2) + (3)	B
44-46	(10) = (1) + (2) + (4)	A
47-49	(11) = (1) + (2) + (5)	C

The first strategy randomly changes the order of clauses, which obviously would not change the satisfiability of the set of clauses.

The second strategy randomly swaps literals inside each clause and leaves the order of all clauses unchanged. This strategy would also maintain the satisfiability of the formula.

The third strategy randomly generates new clauses by only using existing variables, and adds them to the original input formula. The number of new clauses and the number of literals of each new clause are all between 1 to 100, which prevents an oversize new formula that the solver cannot process. This method would maintain UNSAT, but might not preserve SAT.

Strategy (4) introduces a new variable to the formula by adding a new clause which only consisting that variable itself. The new clause is SAT; therefore, it will maintain the satisfiability.

Strategy (5) also introduces a new variable to the formula but with two contradicting clauses. Thus, the formula would become UNSAT whatever it was before the mutation.

Strategy (6) combines (2) and (1), by shuffling the literals inside each clause and swapping the order of all clauses would still maintain the satisfiability of the formula.

Strategy (7) combines (3) and (1). This method adds new clauses first and then swaps the order of all clauses. Since (1) does not change the satisfiability while (3) does; therefore, strategy (7) has the same effect as (3).

Strategy (8) combines (3) and (2). Similar to (7), this strategy has the same effect as (3).

Strategy (9) combines (1), (2) and (3). It shuffles clauses order first, then swaps internal literals and add new clauses in the end. Since (1) and (2) maintains the satisfiability, thus, (9) has the same effect as (3).

Strategy (10) combines (1), (2) and (4). Since (1), (2) and (4) maintains the satisfiability, therefore, (10) also keeps the satisfiability unchanged.

Strategy (11) combines (1), (2) and (5). It shuffles clauses order first, then swaps internal literals and add two contradicting clauses in the end. Since (5) guarantees the UNSAT after the mutation, therefore, (11) has the same effect as (5).

All generated files, including DIMACS and expectation texts are in the folder called 'follow-up-tests', which is in the folder of original inputs.

3.2.2 Case Evaluation

After follow-up test cases for all formulae are generated by our strategies, we can run all test cases against the solver, invoked by the script as follows:

```
satfollowup.py /path/to/SUT /path/to/inputs /path/to/follow-up-tests
```

In our experiment, we reached executed lines percentages of 85.57% of 589 for solver 2, and 96.88% of 160 for solver 3.

4 Experimented design ideas

In `func` mode, we originally had one more basic strategies, called randomly deleting clauses. We assumed that deleting clauses could lead to SAT→SAT, UNSAT→UNKNOWN and put it into practice. However, in our experiment, we found out that it was not simple. Since we might reduce the number of variables when some clauses were deleted, then we have to calculate the new number of variables, which is extremely expensive to achieve. Therefore, we decided not to use this strategy in the end.

As to `ub` mode, we designed a special input `sys.max + 1`, which could lead to Integer overflow in our assumption. However, the sanitizer would catch this error and terminated SUT process. Hence, we decided not to consider this special input.

5 Details of optimizations

One valuable optimization part of our fuzzing project is finding the appropriate timeout value. A timeout should be small enough to improve efficiency, and also large enough to reach more code coverage. After several attempts and evaluation, we reached a final decision that 30s of the timeout. Although it takes quite a long time for fuzzing, it is necessary since the file for mutation might be quite large, which requires a long period for reading and processing.

Besides, we found the proper numbers of variables and clauses for `generate_random_number_cnf` method. Since this function has 30% chance to be applied, these numbers significantly impact the rate of generating input and the testing coverage of SUT. We decided finally on using a random number between 20 - 50 for variable and 40 - 100 for clauses.