DEPARTMENT OF COMPUTING
IMPERIAL COLLEGE OF LONDON

---

440 Software Reliability

Fuzzer

Eduardo Janicas, Luca Battistelli, Luca Campese

Contents

November 2017

1 Introduction

We have implemented a black-box fuzzer using the Python programming language. It can be run in two different modes, which will be referred to as `ub` mode and `func` mode in the following sections. It also makes use of a seed provided as an argument which allows our fuzzer to exhibit a deterministic behaviour.

2 `ub` mode

In `ub` mode, our fuzzer first starts by feeding the SUT a certain number of formulae that proved to trigger undefined behaviours during our tests. From then on, it operates both in a dumb and smart fashion, generating alternatively random inputs and correct formulae to feed the SUT.

The random input generator, contained in `create_garbage()` is very simple. Starting with a string that just contains a valid header for a DIMACS formula, the function loops indefinitely choosing one of the five following actions at random:

1. If the generated string is at least 20-character long, it exits returning the string;
2. Adds a punctuation character to the string;
3. Adds a printable character to the string;
4. Adds a digit to the string;
5. Adds a 0;
6. Adds a new-line character.

On the other hand, when generating valid formulae, the algorithm is parameterised with the number of clauses, the number of literals and the formula width. Under these constraints, it appends literals, 0s and newlines to a string to form a correct DIMACS CNF formula.

After generating an input of any kind, the fuzzer saves it in a temporary file and use that file as an argument for `runsat.sh`.

The standard input and the standard error of the process spawned by executing `runsat.sh` are then piped to our Python script. The latter will intercept and, in particular, watch the standard error for the sanitizer output. By using regular expressions tailored to detect this kind of output, our fuzzer can detect:

- Negation of `INT_MIN`;
- Null pointer errors;
- Bitshift errors;
- Use after free;
- Heap and stack buffer overflows;
- Signed integer overflows.

At this stage, one of the following three situations arises:

1. The solver completes execution correctly;
2. The solver takes more than 20 seconds to execute and gets killed;
3. An undefined behaviour is detected, thus the input is saved in a text file named `interesting_input{number}` in the folder `path/to/sut/fuzzed-tests/`. After twenty inputs are found this way, they are overwritten only if an input triggering a kind of undefined behaviour that was not encountered before is found.

2.1 Evaluating `ub` mode

In order to test `ub` mode, the script `run_fuzzer.py`—which also works for `func` mode—was developed so that, for all the SUTs provided in a given folder:

1. Compiles them by running `make [mode]`;
2. Runs our fuzzer against them in the selected mode;
3. Runs `make clean`.

We have discovered by doing this that our fuzzer in `ub` mode manages to find inputs which make all the three SUTs provided exhibit undefined behaviour.

3 `func` mode

When using this mode, the fuzzer behaves in a *blind* fashion, generating new inputs by mutating pre-existing formulae. Only valid inputs will be generated in this phase.

We have developed a set of rules to create new formulae, which can be chained together to generate more complex transformations of existing inputs.

These rules are outlined below:

Transformation number	Rule
0 - 4	Swap clauses (1)
5 - 9	Swap literals inside a clause (2)
10 - 14	Add clauses (3)
15 - 19	Remove clauses (4)
20 - 24	(1) and (2)
25 - 29	(1) and (3)
30 - 34	(1) and (4)
35 - 39	(2) and (3)
40 - 44	(1) and (2) and (3)
45 - 49	(1) and (2) and (4)

The first rule (1) swaps the clauses of a formula randomly—this guarantees that the satisfiability does not change. However, it may cause some solvers to take additional time and to use different functions to compute the result.

The second rule (2) maintains the order of the clauses, but swaps the literals inside each clause randomly. This transformation also leaves the satisfiability of the formula unchanged, but it may cause the propagation mechanism to behave differently.

The third rule (3) adds a random number of clauses to the formula. We keep this number between 0 and N, where N is the total number of clauses in the original formula. This prevents a test case from becoming a much larger formula that the solvers cannot handle. For the same reason, each added clause is limited to having 10 literals, whose identifiers are picked amongst the ones in the original formula.

This technique maintains the satisfiability of an UNSAT clause—i.e. if there was a conflict, that conflict still exists—but now we cannot infer anything about the new formula being SAT. The added clauses may create a new conflict that makes the original SAT formula now UNSAT.

On the contrary, in rule (4), we remove between 0 and N clauses, where N is the total number of clauses in the original formula. If the formula was SAT, removing some clauses will not introduce conflicts, making the new formula SAT. On the other hand, if the formula was UNSAT, we might have removed the clause(s) that caused the conflict, thus making the new formula SAT. However, we cannot assume that this is always the case.

The following rules are obtained by combining the previous ones.

By combining (1) and (2), the satisfiability of the formula remains unchanged as the clauses and the literals inside a clause are just shuffled.

When combining (1) and (3), we know that (1) does not alter satisfiability, but (3) does. Therefore, the satisfiability implications of this rule will be the same as the ones in (3).

When combining (1) and (4), we know that (1) does not alter satisfiability, but (4) does. Therefore, the satisfiability implications of this rule will be the same as the ones in (4).

When combining (2) and (3), we know that (2) does not alter satisfiability, but (3) does. Therefore, the satisfiability implications of this rule will be the same as the ones in (3).

Finally, we can either combine (1), (2) and (3)—which has the same result in satisfiability as just (3) due to the reasons stated above—or we can combine (1), (2) and (4).

It should be noted that we cannot combine (3) and (4), because adding random clauses and then deleting random clauses would leave us with a formula we do not know anything about.

When implementing these transformations, we had to be careful to generate files complying with the DIMACS format. For instance, when adding (resp. deleting) clauses we had to keep track of the number of new (resp. removed) clauses to update the first line of the file. Also, the trailing 0 at the end of each line was kept when swapping literals.

3.1 Evaluating `func` mode

We began by only applying transformations (1) and (2) to generate formulae as inputs to the SUTs. By running the script `sat-follow-up`, we obtained a coverage of around 30%. Then, we added the transformations described above and checked the results with `sat-follow-up` again, and we noticed that the code coverage went up to 90% in the second and the third solvers.

In order to get this result, we also needed to add a timeout in `sat-follow-up` (which has not been committed alongside the rest of the code), as some of the largest inputs caused infinite loops on solver 1 and 3.

4 Next steps

Our fuzzer is not feedback-directed. It would be good to take this into account, by for instance keeping the inputs that increase code coverage.

In the `func` mode, the fuzzer could generate a new formula, call the SUT, and then generate more formulae using the same transformation method if the code coverage increases. On the contrary, if code coverage does not increase, then that transformation can be discarded and the fuzzer would proceed to explore different transformations to exercise different parts of the SUT.