Assisting Fuzzing with Symbolic Execution

Søren Lund Jensen

15. maj 2017

Indhold

1 Introduction and concept **2**

1.1 Problem statement 2

1.2 Fuzzing 3

1.3 Symbolic Execution 4

1.4 Symbolic Execution-Assisted Fuzzing 6

2 Implementation **6**

2.1 Python 6

2.2 The basic algorithm 6

2.3 American Fuzzy Lop 6

2.4 Other Implementation traits 6

3 Testing **6**

3.1 Basis 6

3.2 Results 6

4 Conclusion **6**

4.1 Discussion 6

4.2 Future Work 6

1 Introduction and concept

An ever-present danger in today's society is memory corruption vulnerabilities in software, be they using of uninitialized memory, using dangling null-pointers, buffer overflow, memory leaks, or a fifth, sixth, or seventh vulnerabilities. An attacker could, did he know of these vulnerabilities, exploit them in order to access confidential informations, and as computer processing- and connecting continues to be on the rise, playing a major role in present day, patching these vulnerabilities has to be a priority. This, of course, cannot be done without first discovering the bugs.

Memory corruption bugs are often-case virtually untraceable, as only specific input combinations may trigger them, or the fact that they may appear under very unusual conditions, which makes it very hard to discover, or in some cases, even reproduce them. Add thereto, the fact, that the memory corruption's effect may manifest itself far away from it's source, it can also be hard to even correlate these two, once a bug has been discovered.

A variety of tools exists, with the purpose of bug-discovery, but as the bugs are often very specific, and/or wide-spread, creating a silver bullet is hard, if not impossible. Many vulnerabilities are discovered manually, however, this solution is not scalable, as software applications generally increase in size and complexity. A handful of tools exist, including fuzzers and symbolic execution engines. These do, however have, in the worst cases, deal-breaking flaws, working against them, and their usefulness.

In the following paper, I will combine the AFL fuzzer with a symbolic execution engine, in order to both make use of both of their strengths, as well as mitigate their, not insignificant, weaknesses.

1.1 Problem statement

How can American Fuzzy Lop be used, along with symbolic execution, in order to verify known security vulnerabilities in software?
Does this experiment display a significant difference, in terms of running time and bugs/vulnerabilities found, when compared to "regular"fuzzing?

1.2 Fuzzing

Stemming from the early years of punch-card-programming, a technique, known as fuzzing exists. This technique works by feeding random input to a program, at a very high rate, some of which will hit specific vulnerabilities in said program. Upon vulnerability-hit, a fuzzer logs the vulnerability, along with information about where the vulnerability occurred, and which input triggered it.

An advantage, as well as a drawback of most fuzzers is their execution method. They are as little invasive as possible, as to prioritize speed. This means that a typical fuzzer does not analyse a fuzzed application - instead directly executing the application with random input, which is immensely faster than finding qualified input variables, based on an application analysis.

Features of Fuzzing

Modern fuzzers implement a variety of features, to enhance their efficiency. In this section, I will list some of the key features, offered by fuzzing.
Genetic Fuzzing
When stating that the AFL fuzzing engine relies on executing applications with inputs at absolute random, one is not totally correct. This is due to the technique known as 'Genetic Fuzzing'. Genetic fuzzing means that the engine generates - *unique* - inputs at total random. Simplified, this means that the current input, that AFL is generating cannot be the same as a previously generated input.
Stable Transition Tracking
AFL views the union of source and destination as a tuple of it's destination blocks. These tuples are prioritised, meaning that tuples cause the most different execution are chosen first for future input generation. **Loop Bucketization**
For a symbolic execution engines and fuzzers alike, loops are complicated to handle, as looping potentially offers an added layer of complexity. The AFL fuzzer makes the following contortions, in order to avoid looping's added complexity, and path space requirements: When AFL detects that a triggered path contains a loop, it logs the executed loop iterations and compares this with previous inputs. The paths are grouped, based on the amount of iterations, and hereafter only *one* path in a group is fuzzed XXXXX upon. Using this technique, $O(N)$ of the slow loop-including paths are executed, as opposed to $O(N)$ paths.
Derandomization
TODO

Limitations of Fuzzing

Because of the union of the above techniques and it's general nature, AFL is able to quickly discover a wide selection of general vulnerabilities, meaning vulnerabilities, that are triggered by some *kind* of input. When vulnerability-triggers move past general input, and into the territory of general input AFL can potentially fall seriously behind.

```
int main(void)
{
    int x;
    read(0, &x, sizeof(x));

    if (x == 0x12345678){
        vulnerability();
    }else{
         ...
    }
}
```

Listing 1: A program that is difficult to fuzz

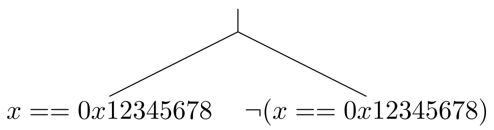
A generic example of this can be seen in Listing 1. This describes a program, that takes an input x from a user. If, and only if, x evaluates to $0x12345678$ the program will fail, as a vulnerability has been triggered, and as so, at each command, executed by the fuzzer, the frequency, and by extension, the chance of discovering the bug, is 1 in 2^{32}. Furthermore, as the AFL lacks the ability to produce new paths within this specific program lacks, it's instrumentation falls short, and AFL is reduced to randomly mutating non-instrumented input.

1.3 Symbolic Execution

Symbolic execution, or symbolic evaluation, is a way of analysing programs, in order to determine the different ways said program can be executed, and which type of input causes it. Instead of executing actual values, to determine this, an interpreter assigns symbolic values to the input variables. The symbolic used to visualise how the program will execute, based on what *kind* of input it will be fed.

Features of Symbolic Execution

As symbolic execution relies on analysing input, instead of mindlessly executing input, it is able to detect specific inputs which, in this case, cause the application to crash. See Listing 1 as an example. A symbolic execution engine will analyse it's functions, and generate the following tree:



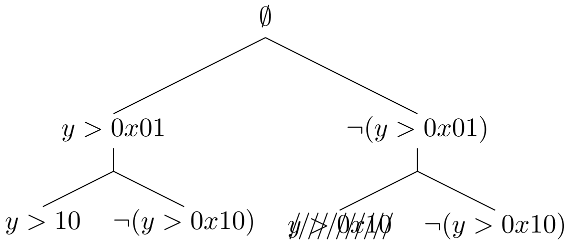
Another advantage of symbolic execution is the ability to invalidate sections input.

```c
int main(void)
{
    int y;
    read(0, &y, sizeof(y));

    if (y > 0x01){
        ...
    }else{
        ...
    }
    if (y > 0x10){
        ...
    }else{
        ...
    }
}
```

Listing 2: Example of Symbolic Execution

In Listing 2 above, for example, the input y is evaluated. A symbolic execution engine will analyse Listing 2's formulae, to find that y will either assume a value greater than $0x01$ or not greater than $0x01$. Furthermore y will either assume a value greater, or not greater than $0x10$, however greater than $0x10$ cannot occur, if y, at the same time was not greater than $0x01$. This will produce the following tree:



Because of this trait, symbolic execution's relevance to the experiment is further heightened, as this allows for AFL to exclude certain value-ranges, when mutating input.

Limitations of Symbolic Execution

Program-Dependent Efficiency
The advantage of having a symbolic execution-engine analyse paths, as opposed to input, is not present in all programs. If some inputs take the same paths in a program, the difference in analysing the inputs instead of paths, is vanishingly small.

Environment Interactions
Often programs interact with their environment, such as executing system calls and/or receiving signals. This becomes a problem when these environment factors are not under the control of the symbolic execution tool, as the tool is not

able to determine branching of a program, when it has insufficient data concerning input.

Path Explosion

The last, and possibly biggest drawback in symbolic execution engines is the path explosion problem. This problem is caused by a program that is containing loops, which causes the amount of path to grow exponentially. In theory the path amount can even become infinite, because of unbound loops. This problem is near-not-existing in small programs, however it often scales faster than the symbolically executed program scales, rendering symbolic execution virtually useless for testing medium to large applications.

1.4 Symbolic Execution-Assisted Fuzzing

Expected Strengths

Expected Weaknesses

2 Implementation

2.1 Python

2.2 The basic algorithm

2.3 American Fuzzy Lop

2.4 Other Implementation traits

3 Testing

3.1 Basis

3.2 Results

Comparable to "Dumb Fuzzing”

Comparable to Symbolic Execution

4 Conclusion

4.1 Discussion

4.2 Future Work