Automated Whitebox Fuzz Testing

by - Patrice Godefroid,
- Michael Y. Levin and
- David Molnar

OUTLINE

- Introduction
- Methods
- Experiments
- Results
- Conclusion

Introduction

- Fuzz testing is an effective Software testing techniques for finding security vulnerabilities in software mostly automated or semi automated
- Apply invalid, unexpected or random data to the inputs of a program
- If the program fails (crashing, access violation exception), defects can be noted
- Cost effective and can find most of known bugs

Whitebox Fuzzing

- Combines fuzz testing with dynamic test generation
  - Run the code with some initial well formed input
  - Collect constraints on inputs with symbolic execution
  - Generate new constraints (By negating one by one)
  - Solve constraints with constraint solver
  - Synthesize new inputs

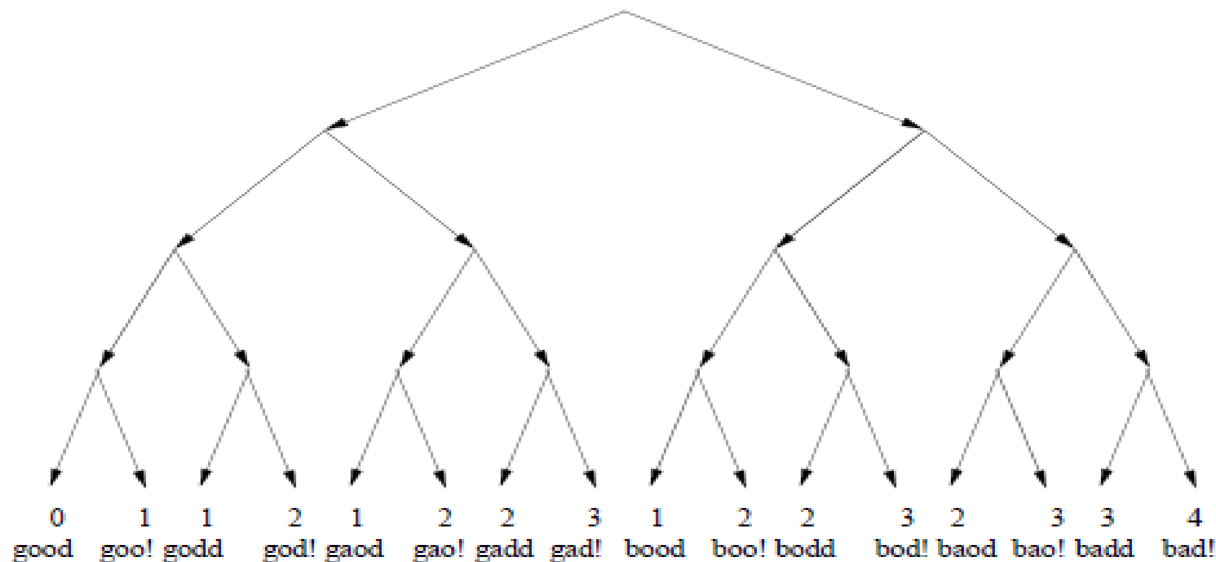
Search Algorithm

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```

- Black box will do poorly in this case
- Dynamic test could do better

Dynamic Test Generation

- For eg. Take input : 'good'
- Collect constrain from trace
- Create a new path constraint
- Solve new constraints, giving new inputs



Limitations of Dynamic Testing

- Path Explosion
  - Path does not scale for large applications with hundreds of instruction
  - Can be corrected by modifying the search algorithm

- Imperfect Symbolic Execution
  - Bound to be imprecise due to complex program statements (pointer manipulation, arithmetic, etc.)
  - Calls to OS and library functions have to be expensive in order to be precise

Generation Search Algorithm

- Designed to systematically explore large applications execution with large input and deep paths
- It uses heuristics to maximize code coverage quickly in order to find bugs faster
- Resilient to divergences. Whenever it occurs, search is able to recover and continue

Search Algorithm

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) {//new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                       has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

Algorithm Summary

- **Part 1**
  - Push input to the list
  - Run&Check(input) check bugs in that input
  - Traverse the list by selecting from the list base in score
  - Expanded child paths and adding to the childlist
  - Traverse childlist Run&Check, assigned score and add to list
- **Expand Execution**
  - Generates Path constrain
  - Attempt to expand path constraints and save them
  - Input.bound is bound is used to limit the backtracking of each sub-search above the branch.

SAGE (Scalable, Automated, Guided Execution)

- Can test any file-reading program running on Windows by treating bytes read from files as symbolic input.
- Another key novelty of SAGE is that it performs symbolic execution of program traces at the x86 binary level

SAGE

- Performs generation search repeating four tasks
- **Tester** task implements function Run & Check
- **Tracer** task runs target program on same file again, this time recording log of run
- **CoverageCollector** task replays the recorded execution to compute which basic blocks were executed during the run.
- **SymbolicExecution** task implements the function ExpandExecution by replaying the recorded execution once again, this time to collect input related constraints and generate new inputs using the constraint solver Disolver

Sage advantages

- Not source-based, SAGE is a machine-code-based, so it can run different languages.
- Expensive to build at the beginning, but less expensive over time
- Test after shipping
  - Since is based in symbolic execution on binary code, SAGE can detects bugs after the production phase
- Not source is needed like in another systems
  - SAGE doesn't even need specific data types or structures not easy visible in machine code

Experiments

- Since 1st MS internal release in April'07: dozens of new security bugs found (most missed by blackbox fuzzers, static analysis)
- Test in different Apps such as image processors, media players, file decoders.
- Many bugs found rated as "security critical, severity 1, priority 1"
- Now used by several teams regularly as part of QA process.

Results

- Statistics from 10hour searches on seven test applications, each seeded with a well formed input file.

Test	# SymExec	SymExecT	Init. |PC|	# Tests	Mean Depth	Mean # Instr.	Mean Size
ANI	808	19099	341	11468	178	2066087	5400
Media 1	564	5625	71	6890	73	3409376	65536
Media 2	3	3457	3202	1045	1100	271432489	27335
Media 3	17	3117	1666	2266	608	54644652	30833
Media 4	7	3108	1598	909	883	133685240	22209
Compressed File	47	1495	111	1527	65	480435	634
OfficeApp	1	3108	15745	3008	6502	923731248	45064

Reported the number of SymbolicExecutor tasks during the search, the total time spent in all SymbolicExecutor tasks in seconds, the number of constraints generated from the seed file, the total number of test cases generated, the mean depth per test case in number of constraints, the mean number of instructions executed after reading the input file, and the mean size of the symbolic input in bytes.

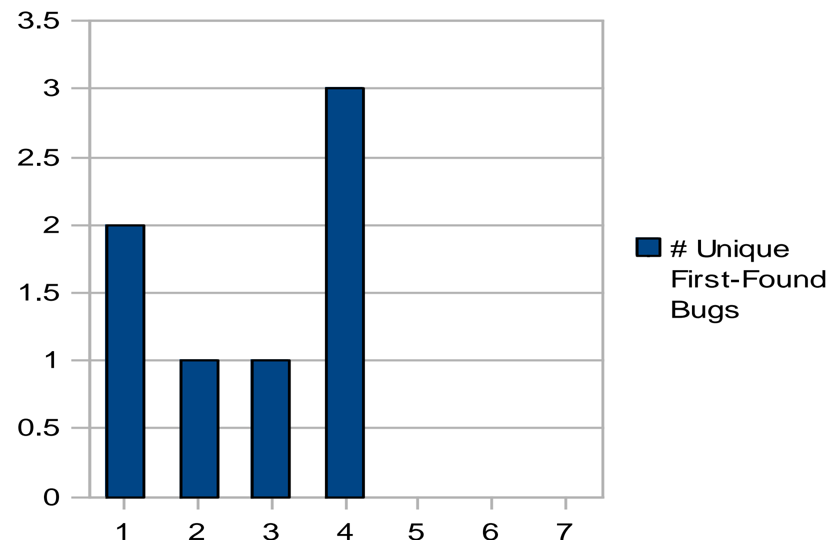
Results

- Focused on the Media 1 and Media 2 parsers.
- Ran a SAGE search for the Media 1 parser with five “well-formed” media files, and five bogus files.

stack hash	wff-1	wff-2	wff-3	wff-4	wff-5	bogus-1
1867196225	×	×	×	×	×	
2031962117	×	×	×	×	×	
612334691		×	×			
1061959981			×	×		
1212954973			×			×
1011628381			×	×		×
842674295				×		
1246509355			×	×		×
1527393075				×		
1277839407					×	
1392730167					×	
1951025690			×			

Results

- Compared with Depth-First Search Method
  - DFS runs for 10 hours for Media 2 with wff-2 and wff-3, didn't find anything while generational search found 15 crashes
- Symbolic Execution is slow
- Well formed input are better than Bogus files
- The heuristic method didn't have too much impact
- Divergences are common
- Most bugs are shallow



Strengths

- Novel approach to do fuzz testing
  - Introduced new search algorithm that use code-coverage maximizing heuristic
- Applied as a black box
  - Not source code was needed
  - symbolic execution of program at the x86 binary level
- Shows results comparing previous results
  - Test large applications previously tested found more bugs.
- Introduced a full system and applied the novel ideas in this paper

Weakness

- The results were non-determinism
  - Same input, program and idea different results.
- Only focus in specific areas
  - X86 windows applications
  - File manipulation applications
- SAGE needs help from different tools
- Thus paper extends too much in the implementation of SAGE, and the system could of be too specific to Microsoft

Conclusions

- Blackbox is lightweight, easy and fast, but poor coverage
- Whitebox is smarter, but complex and slower
- Many apps are so buggy, any form of fuzzing finds bugs can be used
- Once they are eliminated, we can use whitebox, user-provided guidance (grammars), etc.
- Conclusion use both.

Further Work

- Currently work is going on Whitebox fuzzing
- Recently it has found one third of all file fuzzing bugs during development of Windows 7
- Which in turn saved millions of dollar in potential security vulnerabilities
- Two new systems are build on these namely SAGAN and JobCentre
- As per researcher in Microsoft end goal of these is a "testing cloud" that can accept applications, explore them, and report the results to developers with minimum manual work.

- **Questions**

References

- http://research.microsoft.com/en-us/um/people/pg/public_psfiles/icse2013.pdf
- http://en.wikipedia.org/wiki/Fuzz_testing
- http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ndss2008.pdf