ADDIS ABABA INSTITUTE OF TECHNOLOGY

SOFTWARE ENGINEERING II

ARTICLE REVIEW ON: AUTOMATED WHITEBOX FUZZ TESTING

Group MEMbers: Michael desalegn

: rihana abdela

: wakijira abdissa

: Zebiba hassen

Date : 25/04/2018

**Abstract**

Fuzz testing is a techniques used for finding security vulnerabilities in a software, previously this tool applied to random changes to a well-formed inputs of a program and test the corresponding result values, since this approach is random, its unlikely to discover the error, so a better approach to this would be whitebox fuzz testing which is inspired by symbolic execution and dynamic test generation. Here as well we will use well-formed input symbolically evaluating the recorded trace & gather constraints on inputs, whereby the each and every constraints are negated and solved with a constraint solver producing new inputs will take different path from the previous input sets. This process is repeated with the help of code coverage maximizing heuristic designed to find defects as fast as possible. This algorithm is implemented in SAGE which we will be discussed later.

**Keywords**

Fuzz testing : an effective technique for finding security vulnerabilities in software.

SAGE : (Scalable, Automated, Guided Execution), a whole-program whitebox file fuzzing tool for x86 Windows applications

Grammars : used to generate the well-formed inputs

Divergence : when an actual execution path does not match the program path predicted by symbolic execution for a given input vector.

**Introduction**

Fuzz testing is a quick and cost effective method for finding security defects in large application programs. It is a form of blackbox random testing. As the term says, this testing approach randomly mutates the well-formed inputs and tests the program on the resulting test data. This approach provides low code coverage providing serious security bugs. Better approach will be the whitebox fuzz testing. Here as well, we will be using well-formed inputs with a modified searching algorithm which will be discussed later. The collected constraints from conditional statements encountered along the way are then systematically negated and solved with a constraint solver, yielding new inputs that exercise different execution paths in the program. This process is repeated using a novel search algorithm with a coverage-maximizing heuristic designed to find defects as fast as possible. This approach has been remarkably effective in finding defects in applications that had been well tested. It has also addressed to the defect triage problem which is common in static program analysis and blackbox fuzzing. It also is applicable for larger applications than previously done in dynamic test generation. This approach has been implemented using SAGE( Scalable, Automated, Guided Execution). SAGE is capable of finding bugs that are beyond the reach of blackbox fuzzers.

Automated Whitebox Fuzz Testing

As mentioned earlier random testing unlikely to discover errors for large application programs, it is difficult to generate input values that will drive the program through all its possible execution paths. On the other hand, whitebox dynamic test generation consists in executing the program starting with some initial inputs, performing a dynamic symbolic execution to collect constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next executions of the program towards alternative program branches. This process is repeated until a given specific program statement or path is executed, or until all (or many) feasible program paths of the program are exercised. This also has two major limitations which is Path Explosion : systematically executing all feasible program paths does not scale to large, realistic programs,

& Imperfect Symbolic Execution: symbolic execution of large programs is bound to be imprecise due to complex program statements, causing divergence. For this reason it has been introduced a new search algorithm, the generational search, for dynamic test generation that tolerates divergences and better leverages expensive symbolic execution tasks. It is designed to systematically yet partially explore the state spaces of large applications executed with large inputs, maximizes the number of new tests generated from each symbolic execution, uses heuristics to maximize code coverage as quickly as possible.

Search(inputSeed){

inputSeed.bound = 0;

workList = {inputSeed}; //places the initial input inputSeed in a workList

Run&Check(inputSeed);//runs the program to check whether any bugs are detected during the first execution

while (workList not empty) {//new children ; inputs in the workList are then processed

input = PickFirstItem(workList);// selects an element

childInputs = ExpandExecution(input); //expands the element

while (childInputs not empty) {

newInput = PickOneItem(childInputs);

Run&Check(newInput); // the execution is checked for errors

Score(newInput); // assigns score

workList = workList + newInput; //gets added to the worklist

}

}

}

ExpandExecution(input) {

  childInputs = {};

  // symbolically execute (program,input)

PC = ComputePathConstraint(input);

  for (j=input.bound; j < |PC|; j++) {

if((PC[0..(j-1)] and not(PC[j])) has a solution I){

newInput = input + I;

newInput.bound = j;

childInputs = childInputs + newInput;

}

return childInputs;

}

The main originality of this search algorithm is in the way children are expanded, starting with an initial input inputSeed and initial path constraint PC, the new search algorithm will attempt to expand all |PC| constraints in PC, instead of just the last one with a depth-first search, or the first one with a breadth-first search. To prevent these child sub-searches from redundantly exploring overlapping parts of the search space, a parameter bound is used to limit the backtracking of each sub-search above the branch where the sub-search started off its parent. Because each execution is typically expanded with many children, such a search order a generational search. By repeating this process, all feasible execution paths are eventually generated exactly once, thus maximizing the number of new test inputs generated from each symbolic execution.

The generational search algorithm has been implemented in a new tool named SAGE, which stands for Scalable, Automated, Guided Execution. SAGE can test any file-reading program running on Windows by treating bytes read from files as symbolic inputs. SAGE performs a generational search by repeating four different types of tasks.

1. Tester task: implements the function Run&Check by executing a program under test on a test input and looking for unusual events such as access violation exceptions and extreme memory consumption.

2. Tracer task: runs the target program on the same input file again, this time recording a log of the run which will be used by the following tasks to replay the program execution offline.

3. Coverage Collector task: replays the recorded execution to compute which basic blocks were executed during the run.

4. Symbolic Executor 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’s constraint generation differs from previous dynamic test generation implementations in two main ways. First, instead of a source-based instrumentation, SAGE adopts a machine-code-based approach for three main reasons: Multitude of languages and build processes, Compiler and post-build transformations, Unavailability of source. Second, instead of an online instrumentation, SAGE adopts an offline trace-based constraint generation for two reasons. First, a single program may involve a large number of bi- nary components some of which may be protected by the operating system or obfuscated, making it hard to replace them with instrumented versions. Second, inherent non determinism in large target programs makes debugging online constraint generation difficult.

**Existing Work: Blackbox Fuzz Testing**

Dynamic, randomized-input functional testing, or *black-box fuzz testing*, is an effective technique for finding security vulnerabilities in software applications. Parameters for an invocation of blackbox fuzz testing generally include known-good input to use as a basis for randomization (i.e., a seed file) and a specification of how much of the seed file to randomize (i.e., the range). A fuzzer will input massive amount of random data into another program to see how it responds, then reports back with details on how the program responded to the fuzz test. Most often the result of a fuzz test is only a crash, where it is the system, or a process spawned on the system. This technique is conceptually simple yet effective.

**Whitebox Fuzz Testing Vs. BlackBox Fuzz Testing**

Both of these approaches have their own merit as well as demerits. Fuzz technique in general is a technique used for finding security vulnerabilities in software. Software security bugs can be very expensive, which is why security testing is a huge issue. Blackbox fuzz testing randomly mutates well-formed inputs and tests the program on the resulting data, thus usually provides low code coverage. In the security context, these limitations mean that potentially serious security bugs, such as buffer overflows, may be missed because the code that contains the bug is not even exercised, on the other hand whitebox fuzz testing, which is inspired by systematic dynamic test generation, which in theory can lead to a full path coverage. The defect triage problem which is common in static program analysis and blackbox fuzzing, but has not been faced until now in the context of dynamic test generation has been also addressed by whitebox fuzz testing approach. Without any format-specific knowledge, SAGE detects the critical MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis. In Blackbox fuzz testing it is difficult to generate input values that will drive the program through all its possible execution paths, in contrast, whitebox dynamic test generation can easily find the errors. Whitebox fuzz testing is perhaps not significant when exhaustively exploring all execution paths of small programs, it is important when symbolic execution takes a long time, as is the case for large applications where exercising all execution paths is virtually hopeless anyway, thus Blackbox fuzz testing is applicable for smaller applications.

To summarize, Blackbox is lightweight, easy and fast which is good for smaller application programs, but poor coverage, while Whitebox is smarter, but complex and slower which is good for larger applications. A whitebox fuzzer is slower than a blackbox fuzzer, Whitebox needs more time to generate new tests (since smarter). But over time, blackbox is more expensive than whitebox, Blackbox does not know when to (soundly) stop!

Example:

int magic(int input){

if (input==645723) error();

return 0;

}

Coverage(bugs)

2 Whitebox

Blackbox

1

Time(money)

1. 2

**Conclusion**

Fuzz testing is an effective technique for finding security vulnerabilities in software. Whitebox fuzz testing approach inspired by recent advances in symbolic execution and dynamic test generation. It has implemented this approach in SAGE, short for Scalable, Automated, Guided Execution, a whole-program whitebox file fuzzing tool for x86 Windows applications. This approach tests larger applications than previously done in dynamic test generation. Whitebox dynamic test generation can easily find the error in large programs. It introduced a new search algorithm, the generational search, for dynamic test generation that tolerates divergences and better leverages expensive symbolic execution tasks. This search strategy works best if that initial input is well formed. SAGE can test any file-reading program running on Windows by treating bytes read from files as symbolic inputs. Another key novelty of SAGE is that it performs symbolic execution of program traces at the x86 binary level. SAGE has found several bugs that were missed by blackbox fuzzing efforts, can successfully reason about programs on a large scale & uncover serious bugs that do not immediately lead to crashes. Blackbox fuzzing is simple yet also effective. Many apps are so buggy, any form of bugging finds bugs in those, once low hanging bugs are gone, fuzzing must become smarter, use whitebox fuzz testing then. To conclude use both fuzzing techniques.

**References**

* <https://patricegodefroid.github.io>