Coverage-based Greybox Fuzzing as Markov Chain

Marcel Bohme, Van-Thuan Pham, Abhik Roychoudhury

School Of Computing, NUS, Singapore

FM Update 2018

Presented by - Raveendra Kumar M, Animesh Basak Chowdhury

TCS Research

July 27, 2018

Some of the slides are adapted from Author's presentation.

Introduction

Fuzz testing is an automated testing technique that uncovers software error by executing the target program with large number of *randomly* generated test inputs.

Three main approaches.

- Black-box fuzzing : Random testing[1].
- White-box fuzzing: SAGE [2].
- Grey-box fuzzing : American Fuzzy Lop [3].

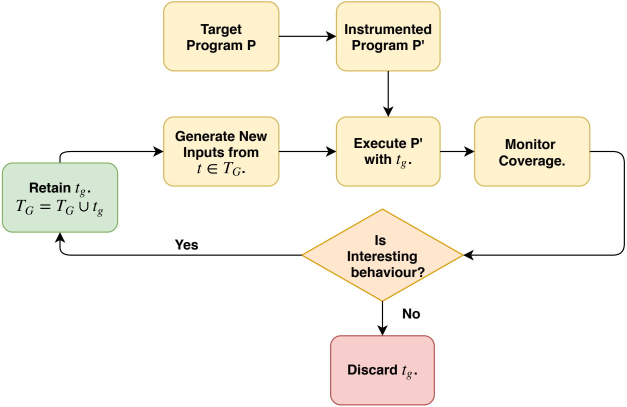
[1] Miller et al, An empirical study of Unix utilities, CACM, 1990.

[2] Goefroid et al, Automated whitebox fuzz testing, NDSS, 2008.

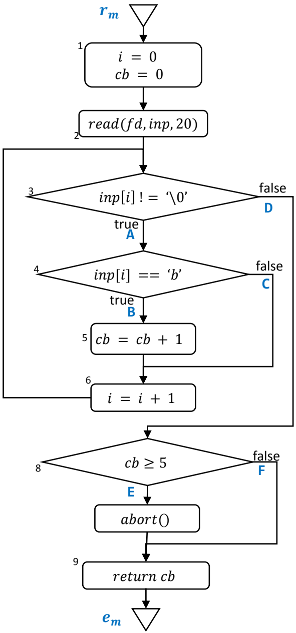
[3] Zalewski, http://lcamtuf.coredump.cx/afl/.

Grey-box fuzzing

Black-Box Fuzzing $\rightarrow$ Open Loop Control System.
GreyBox Fuzzing $\rightarrow$ Closed Loop Control System.
Feedback Function H(s) $\sim$ Branch-Pair Coverage (Pair of consecutive nodes in a CFG)



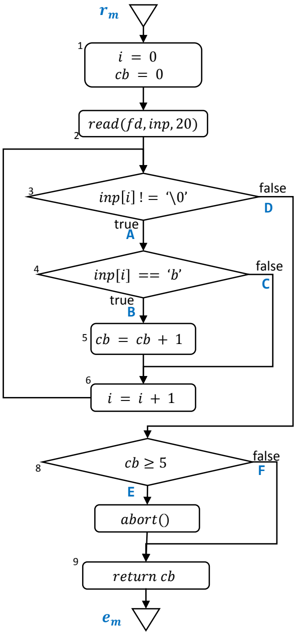
Grey-box fuzzing – Working example

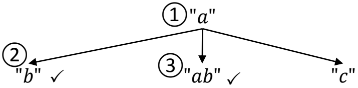


1 "a"

Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1

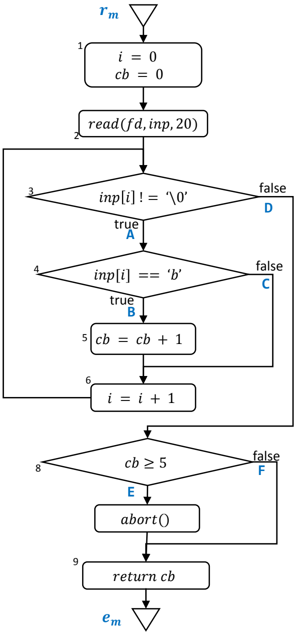
Grey-box fuzzing – Working example

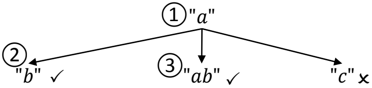




Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	“b”	1				1			1
3	“ab”	1	1		1	1			1
	“c”		1				1		1

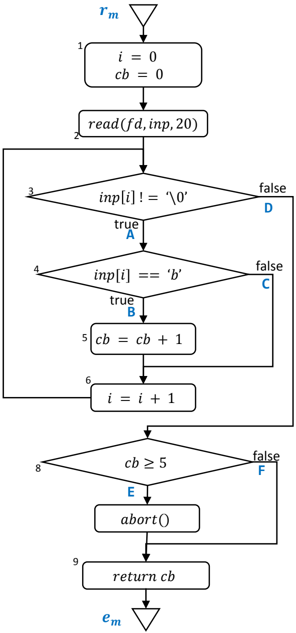
Grey-box fuzzing – Working example

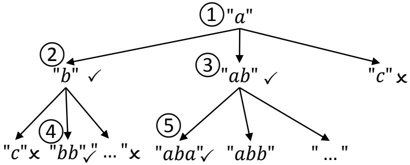




Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

Grey-box fuzzing – Working example





Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
4	"bb"	2		1		1			1
5	"aba"	1	2	1	1		1		1
	"abb"	2	1	1	1	1			1

Grey-box fuzzing algorithm

Algorithm 1 Grey-box fuzzing algorithm

Require: Program P, Initial non-crashing seeds I_s.
Ensure: Set of crashing inputs T_C and a tree of test inputs T_G for P.

$T_G = I_s$
Run P with I_s and observe visit counts of *branch pairs*.
repeat
  t = getNextInput() ▷ $t \in T_G$.
  N = assignEnergy(t)
  T_m = fuzzTestInput(t,N) ▷ $T_m : \{t_g | t_g \in MUTATE(t)\}$
  for all $t_g \in T_m$ **do**
    S_g = run(P,t_g)
    if $S_g = \perp$ **then** ▷ Did t_g caused a crash or hang ?
      $T_C.add(t_g)$
    else if isInterestingTestInput(t_g,S_g) **then**
      $T_G.add(t_g)$ ▷ Retain interesting test input
    end if
  end for
until User interrupt received.
return (T_G, T_C)

$N = assignEnergy(t)$

Let N=100.

Let N_1 be the $N * a$ factor inversely proportional to t_g's execution time.

(Ranging from 0.1 for higher execution time to 3 times for lower execution times)

Let N_2 be $N_1 * a$ factor based on number of branch pairs covered by t_g.

(Ranging from 0.25 for lower coverage to 3 times for higher coverage)

Let N_3 be $N_2 * a$ factor based on cycle of t_g's discovery and number of time t fuzzed.

(Low = 1 to high = 4)

Let N_4 be $N_3 * a$ factor based on depth of t_g's discovery.

(Low = 1 to high = 5)

return N_4

Problem Statement

```
void crashme (char *s) {

  if(s[0] == 'b')

    if(s[1] == 'a')

      if(s[2] == 'd')

        if(s[3] == '!')

            abort() ;
}
```

Listing 1: Program crashes when string s == "bad!"

BlackBox Fuzzing

- Assumption : 2^8 characters.
- Expected no. of testcase required to catch the bug : 2^{32}.

Coverage-based GreyBox Fuzzing (CGF)

- Markov Chain modeling of CGF gives the expectation that 2^{12} is minimum test required to catch the crash.
- Current CGF algorithms are independent of judicious energy assignment to interesting test vectors for further fuzzing.

Problem Statement

```
void crashme (char *s) {

  if(s[0] == 'b')

    if(s[1] == 'a')

      if(s[2] == 'd')

        if(s[3] == '!')

          abort() ;
}
```

Listing 2: Program crashes when string s == "bad!"

Objective

Tune energy assignment scheme close to ideal.

BlackBox Fuzzing

- Assumption : 2^8 characters.
- Expected no. of testcase required to catch the bug : 2^{32}.

Coverage-based GreyBox Fuzzing (CGF)

- Markov Chain modeling of CGF gives the expectation that 2^{12} tests are required to catch the crash.
- Current CGF algorithms are independent of judicious energy assignment to interesting test vectors for further fuzzing.

Some terminologies

Branch Pair Tuple BP_i : $<bp_i, C_i>$ where, bp_i - Branch Pair i, C_i - Visit Count.

Path: Sequence of branch pair tuples $[BP_i, BP_j \ldots]$ visited during the execution of the program P on a test vector t.

Basic Concepts : Probabilistic Modeling

Random Variable

Maps possible outcomes from Sample Space to a real valued number.

$$X : \Omega \rightarrow \mathbb{R}$$

Conditional Probability

Calculates probability of an event happening, given a partial information.

$$P(B|A) = P(B \cap A)/P(A)$$

Stochastic Process

Collection of Random Variables indexed by time.

Discrete Time Stochastic Process (DTSP)

Sequence of random variables X_0, X_1, X_2, . . Denoted by $\{ X_n \}$.

Time: n = 0, 1, 2, . . .

State Space: m-dimensional vector, s = $(s_1, s_2, \ldots, s_m)$
Set of all values that the X_n's can take.

Also, X_n takes one of m values, so $X_n \leftrightarrow s$.

Discrete Time Markov Chain (DTMC)

DTSP $\rightarrow$ Discrete time Markov Chain (DTMC) iff

$P[X_{n+1} = j \mid X_n = i_n, ..., X_0 = i_0] = P[X_{n+1} = j \mid X_n = i_n] = P_{ij}(n)$ (Markovian Property)

Markov Property

Future state is independent of the past given the present state is fully known/observable.

$P_{ij}(n)$: Probability of transition from state i to state j, at time n.

This is also referred as one-step transition probability.

Rat Maze Problem as DTMC

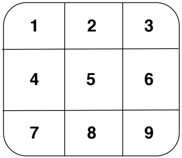


Figure : A rat maze. Allowed transitions are horizontal and vertical neighbors.

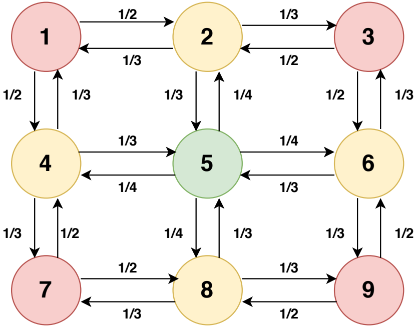


Figure : Markov Chain Modeling of Rat Maze Problem

Homogeneous DTMC

DTMC $\rightarrow$ Homogeneous iff transition probabilities do not depend on the time n, i.e.

$P[X_{n+1} = j | Xn = i] = P[X_1 = j | X_0 = i] = P_{ij}$.

Transition matrix of Homogeneous DTMC $P = [P_{ij}]_{i,j \in E}$

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} \\ p_{4,1} & p_{4,2} & p_{4,3} & p_{4,4} \end{pmatrix}$$

Coverage-Based Fuzzing as Homogeneous DTMC

Coverage-based Greybox fuzzing can modeled as Timed homogeneous DTMC.

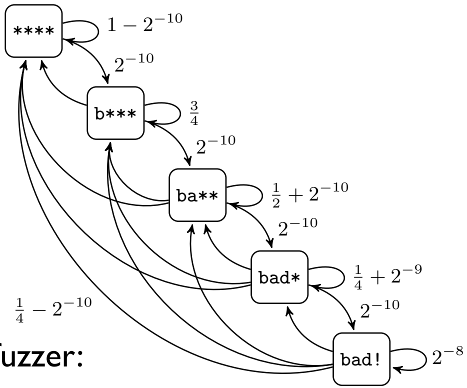
State Space $S = S^+ + S^-$.

S^+ - Paths already explored by seeds T_G.
S^- - Paths yet to be discovered by fuzzing $t \in T_G$.

Assumptions :
Probability of exercising path i(undiscovered) from already generated input t_j, is same as probability of creating test input t_j from test vectors t_i.

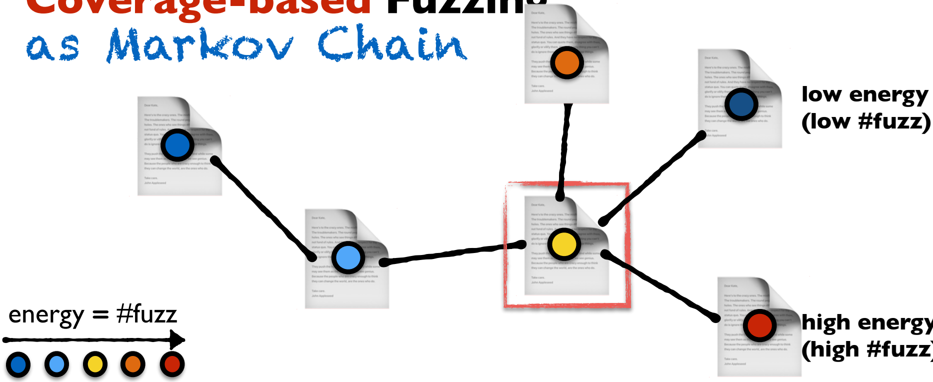
Example

```
void crashme (char* s) {
  if (s[0] == 'b')
    if (s[1] == 'a')
      if (s[2] == 'd')
        if (s[3] == '!')
          abort();
}
```



- Defining the coverage-based fuzzer:
  - Start with seed that is a random 4-letter word.
  - Given a seed, the fuzzer chooses a letter and substitutes it.

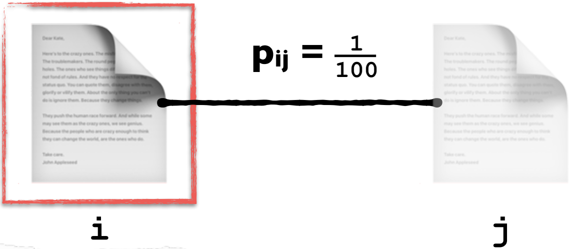
Greybox Coverage-based Fuzzing as Markov Chain



Markov chain describes the probability $\mathtt{p_{ij}}$ that fuzzing the input exercising path $\mathtt{i}$ generates an input exercising path $\mathtt{j}$

Coverage-based Fuzzing as Markov Chain

Greybox



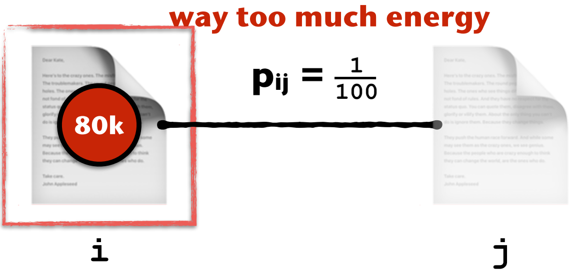
$$p_{ij} = \frac{1}{100}$$

= What is the minimum energy required to expect discovery of new path j?

Greybox

Challenges of Coverage-based Fuzzing

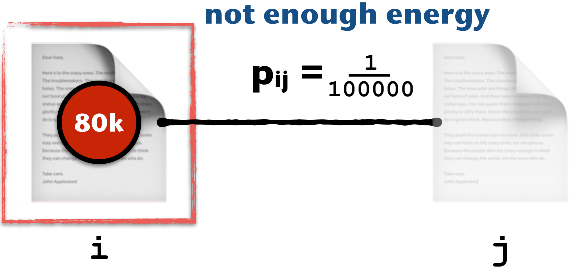
- AFL's power schedule is *constant* in the number of times s(i) the seed has been chosen for fuzzing.



Greybox

Challenges of Coverage-based Fuzzing

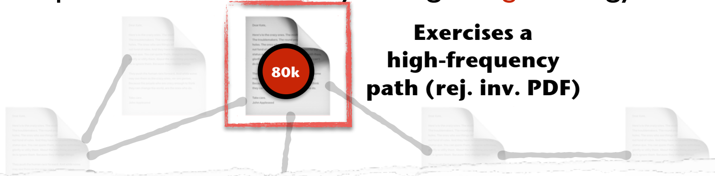
- AFL's power schedule is *constant* in the number of times s(i) the seed has been chosen for fuzzing.



Greybox

Challenges of **Coverage-based** Fuzzing

- AFL's power schedule is *constant* in the number of times s(i) the seed has been chosen for fuzzing.
- AFL's power schedule always assigns *high* energy



Too much energy assigned to high-frequency paths!

Stationary Distribution and Neighborhood Density

For a time homogeneous DTMC, the vector π is called stationary distribution of MC.

$$\forall j \in S,\ 0 \leq \pi_j \leq 1.$$
$$1 = \sum_{i \in S} \pi_i.$$
$$\pi_j = \sum_{i \in S} \pi_i * p_{ij}$$

Neighborhood Density of π

- High Density Region :- Set of neighborhood of paths I , where $\mu_{i \in I}(\pi_i) > \mu_{t_g \in T_G}(\pi_g)$.
- Low Density Region :- Set of neighborhood of paths I , where $\mu_{i \in I}(\pi_i) < \mu_{t_g \in T_G}(\pi_g)$.

μ : Arithmetic Mean

Greybox

Challenges of Coverage-based Fuzzing

- AFL spends too much energy on high-frequency paths.
- We suggest to spend more energy on low-frequency paths and less energy on high-frequency paths.
- We suggest to spend the minimum energy required to discover a new state.

A power schedule manages the energy spent on each state.

Power Schedules

- Constant: $p(i) = \alpha(i)$
  - AFL uses this schedule (fuzzing ~1 minute)
  - $\alpha(i)$ .. how AFL judges fuzzing time for the test exercising path i
- Cut-off Exponential: $p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases}$
  - energy increases exponentially
  - but spend no energy on states in high-density region
  - $\beta > 1$ .. is a constant
  - $s(i)$ .. #times the input exercising path i has been chosen for fuzzing
  - $f(i)$ .. #fuzz exercising path i (path-frequency)
  - μ .. mean #fuzz exercising a discovered path (avg. path-frequency)
  - M .. maximum energy expendable on a state

Power Schedules

- Exponential: $p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right)$
  - Instead of spending *no* energy on states in high-density region,
  - spend energy *proportional* to the density for the state's region
- Cut-off Exponential: $p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases}$
  - energy increases exponentially
  - but spend no energy on states in high-density region

  - $\beta > 1$ .. is a constant
  - $s(i)$ .. #times the input exercising path i has been chosen for fuzzing
  - $f(i)$ .. #fuzz exercising path i (approx. the page rank of i)
  - μ .. mean #fuzz exercising a discovered path
  - M .. maximum energy expendable on a state

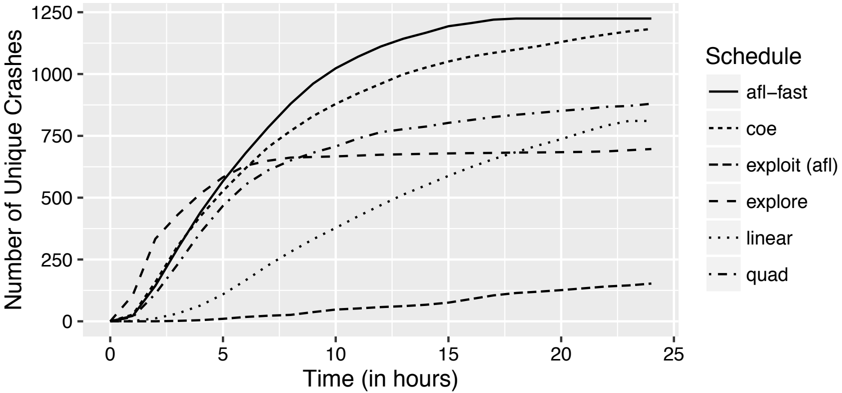
Experiments

- Binutils (nm, objdump, strings, size, cxxfilt)
  - it is a difficult subject because it takes program binaries as input.
  - vulnerabilities exist in GDB, Valgrind, Gcov and other libbfd-based tools.
  - attacker might modify a binary such that it becomes malicious upon analysis!
    - e.g., during *scan* for malicious software or during reverse engineering.

Vulnerability	Type
CVE-2016-2226	Exploitable Buffer Overflow
CVE-2016-4487	Invalid Write due to a Use-After-Free
CVE-2016-4488	Invalid Write due to a Use-After-Free
CVE-2016-4489	Invalid Write due to Integer Overflow
CVE-2016-4490	Write Access Violation
CVE-2016-4491	Various Stack Corruptions
CVE-2016-4492	Write Access Violation
CVE-2016-4493	Write Access Violation
CVE Requested	Stack Corruption
Bug 1	Buffer Overflow (Invalid Read)
Bug 2	Buffer Overflow (Invalid Read)
Bug 3	Buffer Overflow (Invalid Read)

We found and reported these vulns. AND use them for our evaluation.

Power Schedules



AFLFast @ DARPA Cyber Grand Challenge

- An independent evaluation by team Codejitsu found that AFLFast exposes errors in the benchmark binaries of the DARPA Cyber Grand Challenge 19x faster than AFL.
- In the CGC finals, team Codejitsu placed 5th overall but placed 2nd in terms of Vulnerability Detection (i.e., 2nd highest evaluation score).

Questions ?

Thank You !