定向覆盖模糊测试工具的设计与实现

毕业设计检查

雷尚远

南京邮电大学计算机学院

2023 年 6 月

1 Background
- Pre-Knowledge
- Motivation
- Research Status

2 Theory

3 Work

4 References

What Fuzzing is?

Defination[1]

- **Fuzzing** Fuzzing is the execution of the PUT using input(s) sampled from an input space (the “fuzz input space”) that protrudes the expected input space of the PUT.
  - PUT: Program Under Test
- **Fuzz testing** Fuzz testing is the use of fuzzing to test if a PUT violates a correctness policy.
- **Fuzzer** A fuzzer is a program that performs fuzz testing on a PUT.
- **Bug Oracle** A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific correctness policy.
- **Fuzz Configuration** A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.
- **Seed** A seed is a (commonly well-structured) input to the PUT, used to generate test cases by modifying it.

What Fuzzing is?

Fuzz Testing

Input: $\mathbb{C}$, $\mathsf{t}_{\text{limit}}$

Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$

$\mathbb{C} \leftarrow \texttt{Preprocess}(\mathbb{C})$

while $\mathsf{t}_{\texttt{elapsed}} < \mathsf{t}_{\texttt{limit}} \wedge \texttt{Continue}(\mathbb{C})$ **do**

  $\texttt{conf} \leftarrow \texttt{Schedule}(\mathbb{C}, \mathsf{t}_{elapsed}, \mathsf{t}_{limit})$

  $\texttt{tcs} \leftarrow \texttt{InputGen}(conf)$

  // O_{bug} is embedded in a fuzzer

  $\mathbb{B}', \texttt{execinfos} \leftarrow \texttt{InputEval}(conf, tcs, \mathsf{O}_{bug})$

  $\mathbb{C} \leftarrow \texttt{ConfUpdate}(\mathbb{C}, conf, execinfos)$

  $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$

return $\mathbb{B}$

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
𝔹 ← ∅
ℂ ← Preprocess(ℂ)
while t_elapsed < t_limit ∧ Continue(ℂ) do
    conf ← Schedule(ℂ, t_elapsed, t_limit)
    tcs ← InputGen(conf)
    // O_bug is embedded in a fuzzer
    𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
    ℂ ← ConfUpdate(ℂ, conf, execinfos)
    𝔹 ← 𝔹 ∪ 𝔹′
return 𝔹
```

- $\mathbb{C}$:a set of fuzz configurations
- t_{limit}: timeout
- $\mathbb{B}$: a set of discovered bugs

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{elapsed} < t_{limit} \land$ Continue($\mathbb{C}$) **do**
    conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, t_{limit})
    tcs $\leftarrow$ InputGen(*conf*)
    // O_{bug} is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ InputEval(*conf*, *tcs*, O_{bug})
    $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, *conf*, *execinfos*)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

PREPROCESS ($\mathbb{C}$) $\rightarrow \mathbb{C}$

- **Instrumentation**
  - grey-box and white-box fuzzers
  - static/dynamic(INPUTEVAL)
- **Seed Selection**
  - weed out potentially redundant configurations
- **Seed Trimming**
  - reduce the size of seeds
- **Preparing a Driver Application**
  - library Fuzzing, kernal Fuzzing

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) **do**
    conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, t_{limit})
    tcs $\leftarrow$ InputGen($conf$)
    // O_{bug} is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ InputEval($conf$, tcs, O_{bug})
    $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, $conf$, $execinfos$)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

Stop Condition

- $t_{\text{elapsed}} < t_{\text{limit}}$
- CONTINUE ($\mathbb{C}$)$\rightarrow$ {True, False}
  - Determine whether a new fuzz iteration should occur

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) **do**
  conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, t_{limit})
  tcs $\leftarrow$ InputGen(*conf*)
  // O_{bug} is embedded in a fuzzer
  $\mathbb{B}'$, execinfos $\leftarrow$ InputEval(*conf*, *tcs*, O_{bug})
  $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, *conf*, *execinfos*)
  $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

SCHEDULE ($\mathbb{C}$, t_{elapsed}, t_{limit}) $\rightarrow$ conf

- **Function**
  - Pick important information(conf)
- **FCS Problem**
  - *exploration*:Spent time on gathering more accurate information on each configuration to inform future decisions
  - *exploitation*:Spent time on fuzzing the configurations that are currently believed to lead to more favorable outcomes

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs
$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) **do**
    conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, t_{limit})
    tcs $\leftarrow$ InputGen(*conf*)
    // O_{bug} is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ InputEval(*conf*, *tcs*, O_{bug})
    $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, *conf*, *execinfos*)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

INPUTGEN (conf)$\longrightarrow$ tcs

- **function**
  - Generate testcases
- **classification**
  - Generation-based(*Model-based*)
  - Mutation-based(*Model-less*)
  - White-box Fuzzers: symbolic execution

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
𝔹 ← ∅
ℂ ← Preprocess(ℂ)
while t_elapsed < t_limit ∧ Continue(ℂ) do
    conf ← Schedule(ℂ, t_elapsed, t_limit)
    tcs ← InputGen(conf)
    // O_bug is embedded in a fuzzer
    𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
    ℂ ← ConfUpdate(ℂ, conf, execinfos)
    𝔹 ← 𝔹 ∪ 𝔹′
return 𝔹
```

InputEval (conf, tcs, O_{bug})
$\rightarrow \mathbb{B}'$, execinfos

- **Fuzzing PUT**
  - tcs
  - $\mathbb{B}'$
- **Feedback Information**
  - conf, tcs
  - execinfos (tcs,crashes,stack backtrace hash,edge coverage,etc.)

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}

Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$

$\mathbb{C} \leftarrow \texttt{Preprocess}(\mathbb{C})$

while $t_{elapsed} < t_{limit} \wedge \texttt{Continue}(\mathbb{C})$ **do**

    $conf \leftarrow \texttt{Schedule}(\mathbb{C}, t_{elapsed}, t_{limit})$

    $tcs \leftarrow \texttt{InputGen}(conf)$

    // O_{bug} is embedded in a fuzzer

    $\mathbb{B}', execinfos \leftarrow \texttt{InputEval}(conf, tcs, O_{bug})$

    $\mathbb{C} \leftarrow \texttt{ConfUpdate}(\mathbb{C}, conf, execinfos)$

    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$

return $\mathbb{B}$

- CONFUPDATE ($\mathbb{C}$, conf, execinfos) $\rightarrow \mathbb{C}$
  - Update Fuzz Configuration(distinguishablity)
  - Seed Pool Update
- $\mathbb{B} \cup \mathbb{B}' \rightarrow \mathbb{B}$
  - Update Bugs Set

Fuzzing Algorithm

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ Preprocess($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ Continue($\mathbb{C}$) **do**
  conf $\leftarrow$ Schedule($\mathbb{C}$, $t_{elapsed}$, t_{limit})
  tcs $\leftarrow$ InputGen($conf$)
  // O_{bug} is embedded in a fuzzer
  $\mathbb{B}'$, execinfos $\leftarrow$ InputEval($conf$, tcs, O_{bug})
  $\mathbb{C} \leftarrow$ ConfUpdate($\mathbb{C}$, $conf$, $execinfos$)
  $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

stop condition

- $t_{\text{elapsed}} < t_{\text{limit}}$
- CONTINUE ($\mathbb{C}$)$\rightarrow$ {True, False}
  - Determine whether a new fuzz iteration should occur

Classification

The amount of collected information defines the color of a fuzzer[1].

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
𝔹 ← ∅
ℂ ← Preprocess(ℂ)
while t_elapsed < t_limit ∧ Continue(ℂ) do
    conf ← Schedule(ℂ, t_elapsed, t_limit)
    tcs ← InputGen(conf)
    // O_bug is embedded in a fuzzer
    𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
    ℂ ← ConfUpdate(ℂ, conf, execinfos)
    𝔹 ← 𝔹 ∪ 𝔹′
return 𝔹
```

- **program instrumentation**
  - static
  - dynamic
- processor traces
- system call usage
- etc.

Classification

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
𝔹 ← ∅
ℂ ← Preprocess(ℂ)
while t_elapsed < t_limit ∧ Continue(ℂ) do
    conf ← Schedule(ℂ, t_elapsed, t_limit)
    tcs ← InputGen(conf)
    // O_bug is embedded in a fuzzer
    𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
    ℂ ← ConfUpdate(ℂ, conf, execinfos)
    𝔹 ← 𝔹 ∪ 𝔹′
return 𝔹
```

Program Instrumentation

- Static
  - source code
  - intermediate code
  - binary-level
- Dynamic

Classification

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
𝔹 ← ∅
ℂ ← Preprocess(ℂ)
while t_elapsed < t_limit ∧ Continue(ℂ) do
    conf ← Schedule(ℂ, t_elapsed, t_limit)
    tcs ← InputGen(conf)
    // O_bug is embedded in a fuzzer
    𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
    ℂ ← ConfUpdate(ℂ, conf, execinfos)
    𝔹 ← 𝔹 ∪ 𝔹′
return 𝔹
```

Program Instrumentation

- Static
- Dynamic
  - dynamically-linked libraries
  - execution feedback: branch coverage, new path, etc.
  - race condition bugs: thread scheduling

Classification

Classification of Fuzzing

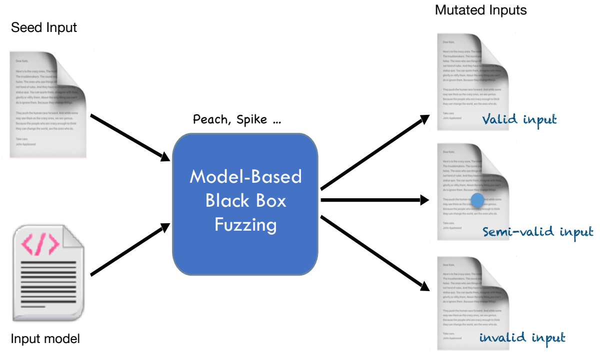
- **Black-box Fuzzing**
  - no program analysis, no feedback
- **White-box Fuzzing**
  - mostly program analysis
- **Grey-box Fuzzing**
  - no program analysis, but feedback

Why Grey-box Fuzzing ?

- Black-box Fuzzing

Defination: techniques that do not see the internals of the PUT,and can observe only the input/output behavior of the PUT, treating it as a black-box[1].

-No program analysis, no feedback

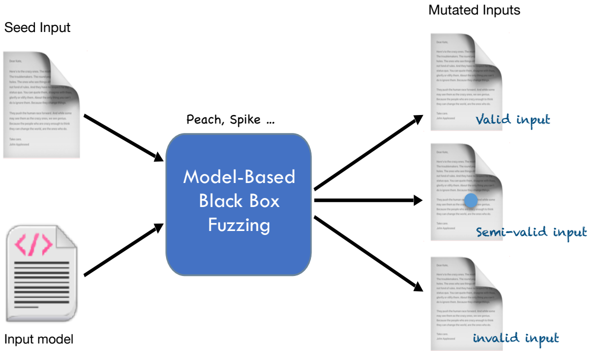


Why Grey-box Fuzzing ?

- Black-box Fuzzing

Defination: techniques that do not see the internals of the PUT,and can observe only the input/output behavior of the PUT, treating it as a black-box[1].

- No program analysis, no feedback



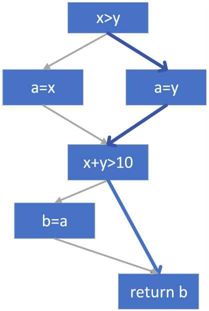
- You have no view of the PUT,but have some view of the input/output domain
- Fuzzing congfigurations are not changed according to some feedback
  - some fuzzer may add the testcases to seed pool
- Not effective

Why Grey-box Fuzzing ?

- White-box Fuzzing

  Defination: techniques that generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT[1].

  - Requires heavy-weight program analysis and constraint solving.



Cover more paths

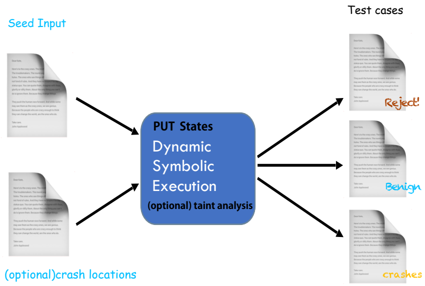
$x \le y \land x+y \le 10$

$x \le y \land \neg x+y \le 10$

$\neg x \le y$

Program Analysis

- **Symbolic Excution**
- **Constrains Satisfation**

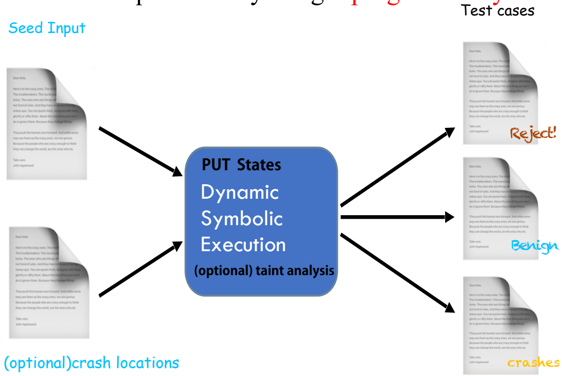


Why Grey-box Fuzzing ?

- White-box Fuzzing

Defination: techniques that generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT[1].

- Requires heavy-weight program analysis and constraint solving.



- You have the view of the PUT state(CFG,CG)
- Heavy-weight Program analysis (effective but not efficient!)

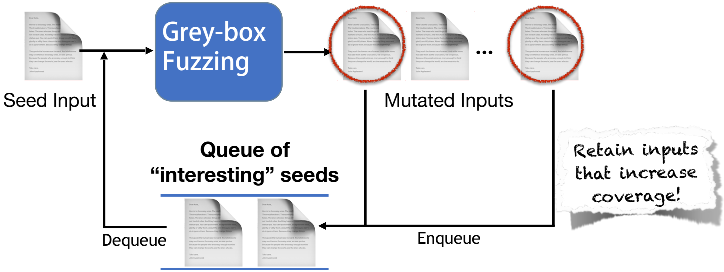
Why Grey-box Fuzzing ?

- Grey-box Fuzzing

  Defination: techniques that can obtain *some* information internal to the PUT and/or its executions to generates test cases[1].

  - Uses only lightweight instrumentation to glean some program structure

  - And coverage feedback



Why Grey-box Fuzzing ?

Grey-box Fuzzing is frequently used

- **State-of-the-art** in automated vulnerability detection
- **Extremely efficient** coverage-based input generation
  - All program analysis before/at instrumentation time.
  - Start with a seed corpus, choose a seed file, fuzz it.
  - Add to corpus only if new input increases coverage.

Why Directed Grey-box Fuzzing ?

Directed Fuzzing has many applications

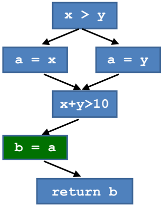
- **Patch Testing**: reach changed statements
- **Crash Reproduction**: exercise stack trace
- **SA Report Verification**: reach "dangerous" location
- **Information Flow Detection**: exercise source-sink pairs

Why Directed Grey-box Fuzzing ?

Directed Fuzzing

- **Goal:reach a specific target**
  - **Target Locations**: the line number in the source code or the virtual memory address at the binary level[2].
  - **Target Bugs**: use-after-free vulnerabilities, etc.

- **DSE:classical constraint satisfaction problem**
  - uses program analysis and constraint solving to generate inputs that systematically and effectively explore the state space of feasible paths[3].
  - **Program analysis** to identify **program paths** that reach given program locations.
  - **Symbolic Execution** to derive **path conditions** for any of the identified paths.
  - **Constraint Solving** to find an input

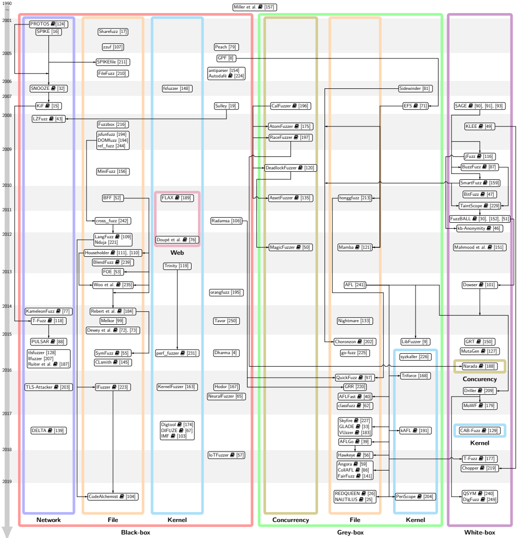


$$\varphi_1 = (x>y) \wedge (x+y>10)$$
$$\varphi_2 = \neg(x>y) \wedge (x+y>10)$$

Why Directed Grey-box Fuzzing ?

- **Effectiveness comes at the cost of efficiency**
- **Heavy-weight program analysis**

Genealogy tracing significant fuzzers' lineage[1]



[1]paper[1]-Figure1

Representative Work

- **AFLGo(2017)**[4]
- **Hawkeye(2018)**[5]

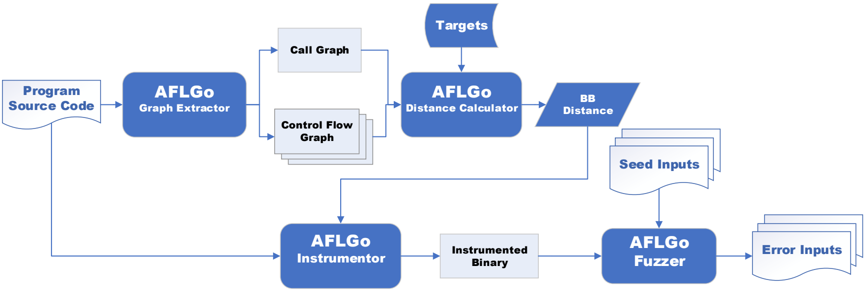
OverView

Directed Fuzzing as **optimisation problem!**

- **Instrumentation Time**
  1. Extract **call graph** (CG) and **control-flow graphs** (CFGs).
  2. For each **BB**, compute **distance** to target locations.
  3. Instrument program to **aggregate distance values**.
- **Runtime**
  1. collect coverage and distance **information**, and
  2. decide **how long to be fuzzed** based on distance.
     - If input is **closer** to the targets, it is fuzzed for **longer**.
     - If input is **further away** from the targets, it is fuzzed for **shorter**.

OverView

AFLGo Architecture



Algorithm

Directed Grey-box Fuzzing

```
Input: 𝕊 // a finite set of seeds
Input: 𝕋 // a finite set of targer sits
Output: 𝕊′ // a finite set of buggy seeds
𝕊′ ← ∅
SeedQueue ← 𝕊
Graphs ← GraphExt(Code)
BBdistance ← DisCalcu(𝕋, Graphs)
while !siganl ∧ t_elapsed < t_limit do
    s ← Dequeue(SeedQueue)
    trace ← Execution(s)
    distance ← SeedDis(trace, BBdistance)
    e ← AssinEnergy(s, t_elapsed, distance)
    for i ← 1 to e do
        s′ ← Mutation(s)
        if s′ crashes then 𝕊′ ← 𝕊′ ∪ s′
        if IsIntersting(s′) then Enqueue(s′, SeedQueue)
return 𝕊′
```

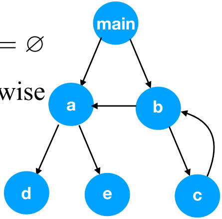
Instrumentation

```
Input: 𝕊 // a finite set of seeds
Input: 𝕋 // a finite set of targer sits
Output: 𝕊′ // a finite set of buggy seeds
𝕊′ ← ∅
SeedQueue ← 𝕊
Graphs ← GraphExt(Code)
BBdistance ← DisCalcu(𝕋, Graphs)
while !siganl ∧ t_elapsed < t_limit do
    s ← Dequeue(SeedQueue)
    trace ← Execution(s)
    distance ← SeedDis(trace, BBdistance)
    e ← AssinEnergy(s, t_elapsed, distance)
    for i ← 1 to e do
        s′ ← Mutation(s)
        if s′ crashes then 𝕊′ ← 𝕊′ ∪ s′
        if IsIntersting(s′) then Enqueue(s′, SeedQueue)
return 𝕊′
```

Instrumentation

- **Function-level target distance**[2]:using call graph (CG)

$$d_f(n, T_f) = \begin{cases} \text{undefined}, & \text{if } R(n, T_f) = \varnothing \\ [\sum_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}]^{-1}, & \text{otherwise} \end{cases}$$

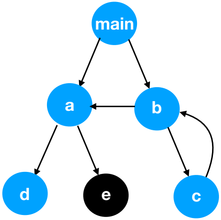


[2] R(n,Tf) is the set of all target functions that are reachable from n in CG

Instrumentation

- **Function-level target distance**:using call graph (CG)

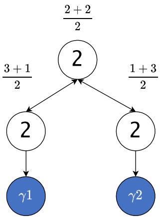
1. Identify **target functions** in CG



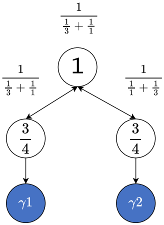
Instrumentation

- **Function-level target distance**:using call graph (CG)

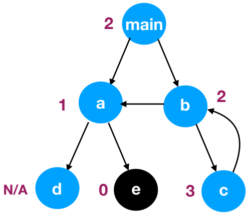
1. Identify **target functions** in CG
2. For each **function**, compute the **harmonic mean** of the **length of the shortest path** to targets



(a)arithmetic mean



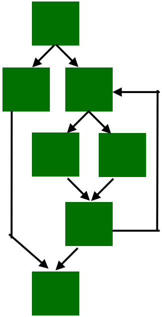
(b)harmonic mean



Instrumentation

- **Function-level target distance**:using call graph (CG)
- **BB-level target distance** [2]:using control-flow graphs (CFG)

$$d_b(m, T_b) = \begin{cases} 0, & \text{if } m \in T_b \\ c \cdot \min\limits_{n \in N(m)} (d_f(n, T_f)), & \text{if } m \in T \\ \left[\sum\limits_{t \in T} (d_b(m,t) + d_b(t, T_b))^{-1}\right]^{-1}, & \text{otherwise} \end{cases}$$



CFG for function b

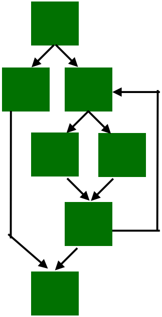
2

- N (m) is the set of functions called by basic block m
- T is the set of basic blocks in control-flow graph

Instrumentation

- **Function-level target distance**:using call graph (CG)
- **BB-level target distance** :using control-flow graphs (CFG)

1. Identify **target BBs** and assign distance 0 (none in function b)

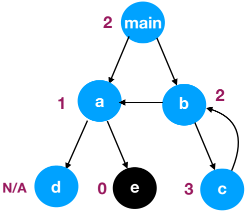


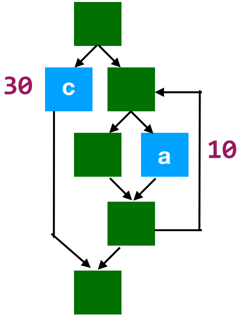
CFG for function b

Instrumentation

- **Function-level target distance**:using call graph (CG)
- **BB-level target distance** :using control-flow graphs (CFG)

1. Identify **target BBs** and assign distance 0
2. Identify BBs that **call function** and assign **10*FLTD**



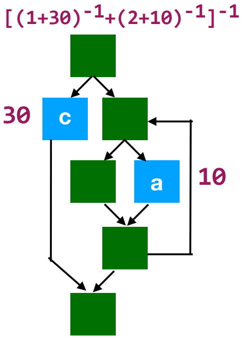


CFG for function b

Instrumentation

- **Function-level target distance**:using call graph (CG)
- **BB-level target distance** :using control-flow graphs (CFG)

1. Identify **target BBs** and assign distance 0
2. Identify BBs that **call function** and assign **10*FLTD**
3. For each BB, compute harmonic mean of (length of shortest path to any function-calling BB + 10*FLTD).

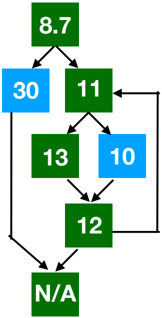


CFG for function b

Instrumentation

- **Function-level target distance**:using call graph (CG)
- **BB-level target distance** :using control-flow graphs (CFG)

1. Identify **target BBs** and assign distance 0
2. Identify BBs that **call function** and assign **10*FLTD**
3. For each BB, compute harmonic mean of (length of shortest path to any function-calling BB + 10*FLTD).



CFG for function b

Runtime

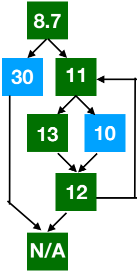
```
Input: 𝕊 // a finite set of seeds
Input: 𝕋 // a finite set of targer sits
Output: 𝕊′ // a finite set of buggy seeds
𝕊′ ← ∅
SeedQueue ← 𝕊
Graphs ← GraphExt(Code)
BBdistance ← DisCalcu(𝕋, Graphs)
while !siganl ∧ t_elapsed < t_limit do
    s ← Dequeue(SeedQueue)
    trace ← Execution(s)
    distance ← SeedDis(trace, BBdistance)
    e ← AssinEnergy(s, t_elapsed, distance)
    for i ← 1 to e do
        s′ ← Mutation(s)
        if s′ crashes then 𝕊′ ← 𝕊′ ∪ s′
        if IsIntersting(s′) then Enqueue(s′, SeedQueue)
return 𝕊′
```

Runtime

Seed distance[a] from instrumented binary

$$d(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|}$$

[a] $\xi(s)$ is the execution trace of a seed s

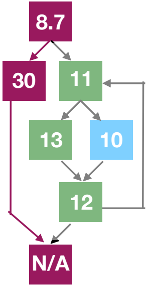


CFG for function b

Runtime

Seed distance from instrumented binary

- Two 64-bit shared memory entries
  - Aggregated BB-level distance values
  - Number of executed BBs

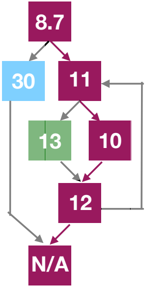


Seed Distance: 19.4
= (8.7+30)/2

Runtime

Seed distance from instrumented binary

- Two 64-bit shared memory entries
  - Aggregated BB-level distance values
  - Number of executed BBs



Seed Distance: 10.4
= (8.7+11+10+12)/4

Runtime

Directed Fuzzingas Optimisation Problem

- **Directed** Greybox Fuzzing
  - Assign **more energy** to seeds that are **closer** to the given targets
  - energy:The number of fuzz generated for a seed s is also called the energy of s.
- **Simulated Annealing**
  - To avoid **local minimum** rather than **global minimum** distance
  - **Sometimes** assign **more energy** to **further-away** seeds
- ***Exploration* vs *Exploitation***
  - **Exploration** phase:
    Energy of **closer** seeds similar to energy of **further-away** seeds
  - **Exploitation** phase:
    - Energy of **closer** seeds is assigned to be **higher** and higher
    - Energy of **further-away** seeds is assigned to be **lower** and lower

Runtime

Directed Fuzzingas Optimisation Problem

- **Temperature**
  $\mathsf{T} \in [0, 1]$ specifies "importance" of distance.
  - normalized seed distance

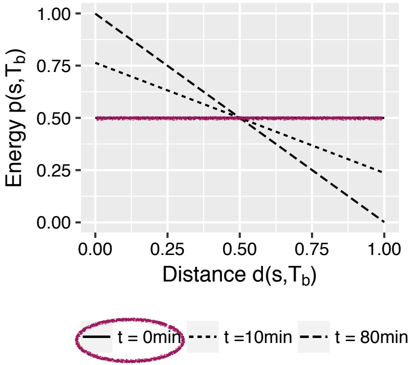
$$\tilde{\mathsf{d}}(\mathsf{s}, \mathsf{T_b}) = \frac{\mathsf{d}(\mathsf{s}, \mathsf{T_b}) - \mathrm{minD}}{\mathrm{maxD} - \mathrm{minD}} \in [0, 1]$$

  - At T=1, **exploration** (normal AFL)
  - At T=0, **exploitation** (gradient descent)
- **Cooling schedule** :controls (global) temperature
  - Classically, exponential cooling.

Runtime

Integrating Simulated Annealing as power schedule

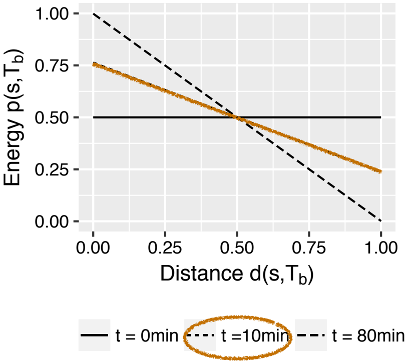
- In the beginning (t = 0min), assign the **same energy** to **all seeds**



Runtime

Integrating Simulated Annealing as power schedule

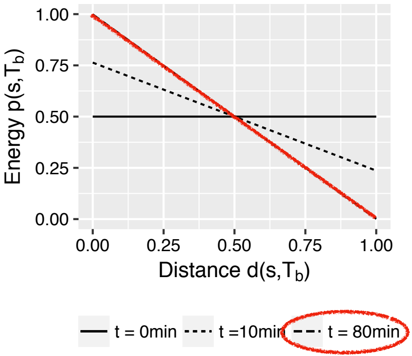
- In the beginning (t = 0min), assign the **same energy** to **all seeds**
- Later (t=10min), assign**a bit more energy** to seeds that are **closer**



Runtime

Integrating Simulated Annealing as power schedule

- In the beginning (t = 0min), assign the **same energy** to **all seeds**
- Later (t=10min), assign**a bit more energy** to seeds that are **closer**
- At exploitation (t=80min), assign **maximal energy** to seeds that are **closest**



Runtime

```
Input: 𝕊 // a finite set of seeds
Input: 𝕋 // a finite set of targer sits
Output: 𝕊′ // a finite set of buggy seeds
𝕊′ ← ∅
SeedQueue ← 𝕊
Graphs ← GraphExt(Code)
BBdistance ← DisCalcu(𝕋, Graphs)
while !siganl ∧ t_elapsed < t_limit do
    s ← Dequeue(SeedQueue)
    trace ← Execution(s)
    distance ← SeedDis(trace, BBdistance)
    e ← AssinEnergy(s, t_elapsed, distance)
    for i ← 1 to e do
        s′ ← Mutation(s)
        if s′ crashes then 𝕊′ ← 𝕊′ ∪ s′
        if IsIntersting(s′) then Enqueue(s′, SeedQueue)
return 𝕊′
```

[1] MANÈS V J, HAN H, HAN C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11) : 2312 – 2331.

[2] WANG P, ZHOU X, LU K, et al. The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing[EB]. arXiv, 2022.

[3] MA K-K, YIT PHANG K, FOSTER J S, et al. Directed symbolic execution[C] // Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18. 2011 : 95 – 111.

[4] BÖHME M, PHAM V-T, NGUYEN M-D, et al. Directed Greybox Fuzzing[C/OL] // Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA : ACM, 2017 : 2329 – 2344. http://dx.doi.org/10.1145/3133956.3134020.

[5] CHEN H, XUE Y, LI Y, et al. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer[C/OL] // Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada : ACM, 2018 : 2095 – 2108. http://dx.doi.org/10.1145/3243734.3243849.

Thanks!