

# 定向覆盖模糊测试工具的设计与实现

## 毕业设计中期检查

雷尚远

南京邮电大学计算机学院

2023 年 4 月 17 日



## ① Background

## ② Theory

## ③ Work

## ④ References

## ① Background

Pre-Knowledge

Motivation

Research Status

## ② Theory

## ③ Work

## ④ References

# 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:**  $\mathbb{C}, t_{\text{limit}}$

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

```

1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
6     //  $O_{\text{bug}}$  is embedded in a fuzzer
7      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
8      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
9      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

# Fuzzing Algorithm

```

1 Input:  $\mathbb{C}$ ,  $t_{\text{limit}}$ 
2 Output:  $\mathbb{B}$  // a finite set of bugs
3  $\mathbb{B} \leftarrow \emptyset$ 
4  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
5 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
6      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
7      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
8     //  $O_{\text{bug}}$  is embedded in a fuzzer
9      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
10     $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
11     $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
12 return  $\mathbb{B}$ 

```

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

# Fuzzing Algorithm

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

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

1.  $\mathbb{B} \leftarrow \emptyset$

$$2 \quad \mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$$

```
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
```

```

4    $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elansed}}, t_{\text{limit}})$ 

```

5	$\text{tcs} \leftarrow \text{InputGen}(\text{conf})$
---	--

```
// Obug is embedded in a fuzzer
```

6  $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, tcs, \mathcal{O}_{\text{bug}})$ 

```

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

```

$$8 \quad \mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$$

```

9 return B

```

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

- **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, kernel Fuzzing



# Fuzzing Algorithm

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
6     //  $O_{\text{bug}}$  is embedded in a fuzzer
7      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
8      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
9      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
10 return  $\mathbb{B}$ 

```

## Stop Condition

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

# Fuzzing Algorithm

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $tcs \leftarrow \text{InputGen}(\text{conf})$ 
6     //  $O_{\text{bug}}$  is embedded in a fuzzer
7      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, tcs, O_{\text{bug}})$ 
8      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
9      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

SCHEDULE ( $\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}}$ )  $\rightarrow$  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:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $tcs \leftarrow \text{InputGen}(\text{conf})$ 
        //  $O_{\text{bug}}$  is embedded in a fuzzer
6      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, tcs, O_{\text{bug}})$ 
7      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

$\text{INPUTGEN}(\text{conf}) \rightarrow tcs$

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

# Fuzzing Algorithm

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
6     //  $O_{\text{bug}}$  is embedded in a fuzzer
7      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
8      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
9      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

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

- **Fuzzing PUT**

- tcs
- $\mathbb{B}'$

- **Feedback Information**

- conf, tcs
- execinfos (tcs, crashes, stack  
backtrace hash, edge coverage, etc.)

# Fuzzing Algorithm

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
        //  $O_{\text{bug}}$  is embedded in a fuzzer
6      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
7      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 
    
```

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

# Fuzzing Algorithm

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
6     //  $O_{\text{bug}}$  is embedded in a fuzzer
7      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
8      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
9      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

## stop condition

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

# ① Background

Pre-Knowledge

**Motivation**

Research Status

## ② Theory

## ③ Work

## ④ References

# Classification

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

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
        //  $O_{\text{bug}}$  is embedded in a fuzzer
6      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
7      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

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



# Classification

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
        //  $O_{\text{bug}}$  is embedded in a fuzzer
6      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
7      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

## Program Instrumentation

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

# Classification

```

Input:  $\mathbb{C}, t_{\text{limit}}$ 
Output:  $\mathbb{B}$  // a finite set of bugs
1  $\mathbb{B} \leftarrow \emptyset$ 
2  $\mathbb{C} \leftarrow \text{Preprocess}(\mathbb{C})$ 
3 while  $t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathbb{C})$  do
4      $\text{conf} \leftarrow \text{Schedule}(\mathbb{C}, t_{\text{elapsed}}, t_{\text{limit}})$ 
5      $\text{tcs} \leftarrow \text{InputGen}(\text{conf})$ 
        //  $O_{\text{bug}}$  is embedded in a fuzzer
6      $\mathbb{B}', \text{execinfos} \leftarrow \text{InputEval}(\text{conf}, \text{tcs}, O_{\text{bug}})$ 
7      $\mathbb{C} \leftarrow \text{ConfUpdate}(\mathbb{C}, \text{conf}, \text{execinfos})$ 
8      $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$ 
9 return  $\mathbb{B}$ 

```

## Program Instrumentation

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

# Classification

## Classification of Fuzzing

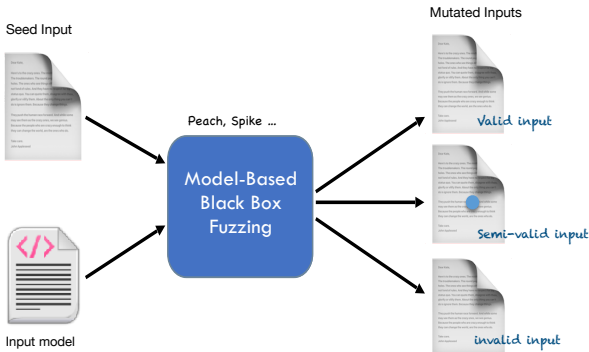
- **Black-box Fuzzing**
  - no program analysis, no feedback
- **White-box Fuzzing**
  - mostly program analysis
- **Grey-box Fuzzing**
  - no program analysis, but feedback

## Why Grey-box Fuzzing ?

- Black-box Fuzzing

**Definition:** 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

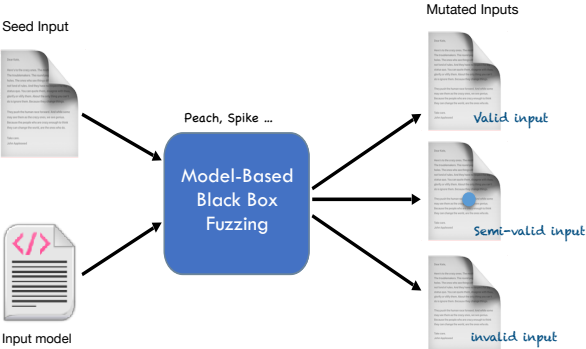


# Why Grey-box Fuzzing ?

- Black-box Fuzzing

**Definition:** 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**



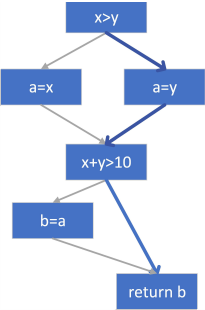
- You have no view of the PUT, but have some view of the input/output domain
- Fuzzing configurations are not changed according to some feedback - some fuzzer may add the testcases to seed pool
- Not effective**

# Why Grey-box Fuzzing ?

- White-box Fuzzing

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

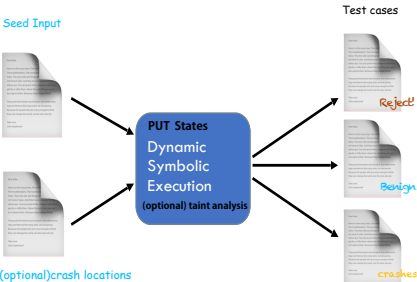
- Requires heavy-weight **program analysis** and constraint solving.



**Cover more paths**

$x \leq y \wedge x + y \leq 10$   
 $x \leq y \wedge \neg x + y \leq 10$   
 $\neg x \leq y$

Program Analysis  
- Symbolic Execution  
- Constrains Satisfaction



# Why Grey-box Fuzzing ?

## • White-box Fuzzing

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

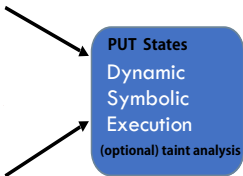
- Requires heavy-weight **program analysis** and constraint solving.

Test cases

Seed Input



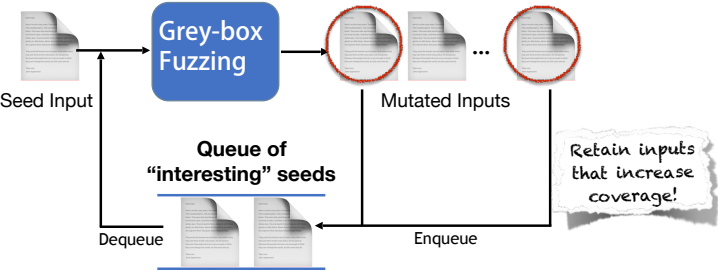
(optional) crash locations



- 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
  - Definition:** 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 ?

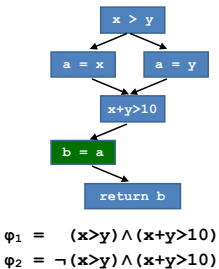
## Directed Fuzzing has many applications

- **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



# Why Directed Grey-box Fuzzing ?

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

## ① Background

Pre-Knowledge

Motivation

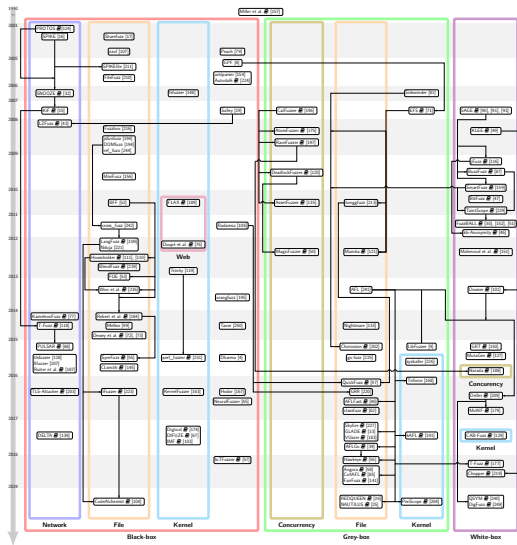
Research Status

## ② Theory

## ③ Work

## ④ References

## Genealogy tracing significant fuzzers' lineage<sup>1</sup>

<sup>1</sup>paper[1]-Figure1

## Representative Work

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

## ① Background

## ② Theory

### AFLGo

## ③ Work

## ④ References



## ① Background

## ② Theory

### AFLGo

## ③ Work

## ④ References

# Overview

## Directed Fuzzing as **optimisation problem!**

- **Instrumentation Time**

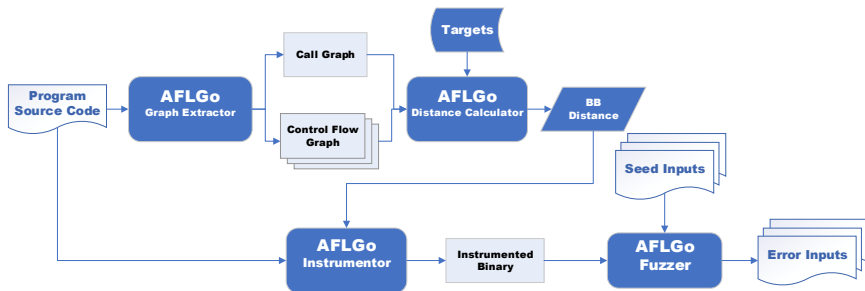
- ① Extract **call graph** (CG) and **control-flow graphs** (CFGs).
- ② For each **BB**, compute **distance** to target locations.
- ③ Instrument program to **aggregate distance values**.

- **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:**  $\mathbb{S}$  // a finite set of seeds

**Input:**  $\mathbb{T}$  // a finite set of target sits

**Output:**  $\mathbb{S}'$  // a finite set of buggy seeds

```

1  $\mathbb{S}' \leftarrow \emptyset$ 
2  $SeedQueue \leftarrow \mathbb{S}$ 
3  $Graphs \leftarrow GraphExt(Code)$ 
4  $BBdistance \leftarrow DisCalcu(\mathbb{T}, Graphs)$ 
5 while !signa1 do
6      $s \leftarrow Dequeue(SeedQueue)$ 
7      $e \leftarrow AssinEnergy(s, t_{elapsed}, BBdistance)$ 
8     for  $i \leftarrow 1$  to  $e$  do
9          $s' \leftarrow Mutation(s)$ 
10         $trace \leftarrow Execution(s')$ 
11        if  $s'$  crashes then  $\mathbb{S}' \leftarrow \mathbb{S}' \cup s'$ 
12        if IsIntersting( $s'$ ) then
13             $distance \leftarrow SeedEva(trace, BBdistance)$ 
14             $SeedQueue \leftarrow SortInser(distance, s', SeedQueue)$ 
15 return  $\mathbb{S}'$ 
  
```

# Instrumentation

```

Input:  $S$  // a finite set of seeds
Input:  $T$  // a finite set of target sites
Output:  $S'$  // a finite set of buggy seeds
1  $S' \leftarrow \emptyset$ 
2  $SeedQueue \leftarrow S$ 
3  $Graphs \leftarrow GraphExt(Code)$ 
4  $BBdistance \leftarrow DisCalcu(T, Graphs)$ 
5 while  $!signa$  do
6    $s \leftarrow Dequeue(SeedQueue)$ 
7    $e \leftarrow AssinEnergy(s, t_{elapsed}, BBdistance)$ 
8   for  $i \leftarrow 1$  to  $e$  do
9      $s' \leftarrow Mutation(s)$ 
10     $trace \leftarrow Execution(s')$ 
11    if  $s'$  crashes then  $S' \leftarrow S' \cup s'$ 
12    if  $IsInteresting(s')$  then
13       $distance \leftarrow SeedEva(trace, BBdistance)$ 
14       $SeedQueue \leftarrow SortInser(distance, s', SeedQueue)$ 
15 return  $S'$ 

```

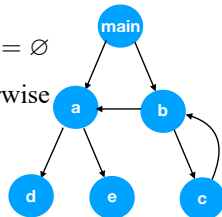
# Instrumentation

- **Function-level target distance<sup>2</sup>**: using call graph (CG)

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

if  $R(n, T_f) = \emptyset$

otherwise

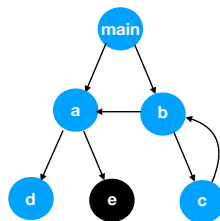


<sup>2</sup> $R(n, T_f)$  is the set of all target functions that are reachable from  $n$  in CG

# Instrumentation

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

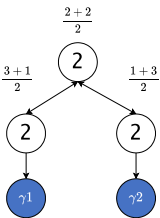
- ① Identify **target functions** in CG



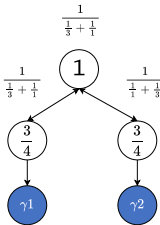
# Instrumentation

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

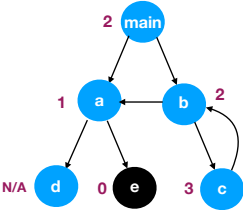
- 1 Identify target functions in CG
- 2 For each **function**, compute the **harmonic mean** of the **length of the shortest path** to targets



(a)arithmetic mean



(b)harmonic mean

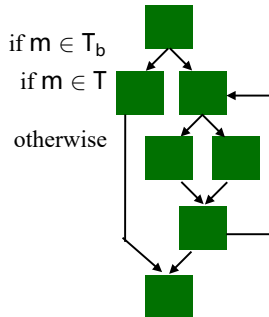




# Instrumentation

- **Function-level target distance**: using call graph (CG)
- **BB-level target distance**<sup>2</sup>: using control-flow graphs (CFG)

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



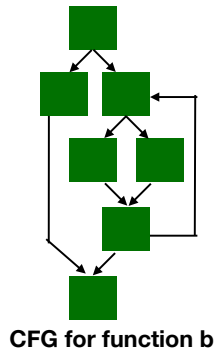
CFG for function b

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

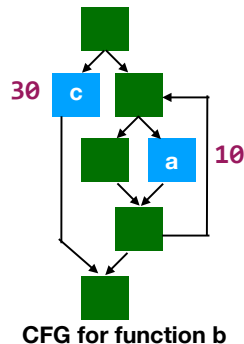
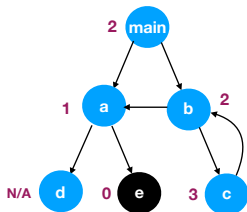
- **Function-level target distance**: using call graph (CG)
  - **BB-level target distance**: using control-flow graphs (CFG)
- ① Identify **target BBs** and assign distance 0  
(none in function b)



# Instrumentation

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

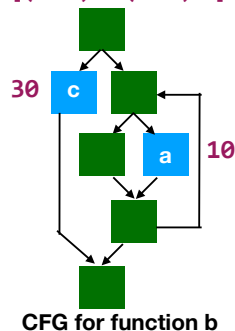
- 1 Identify **target BBs** and assign distance 0
- 2 Identify BBs that **call function** and assign **10\*FLTD**



# 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).

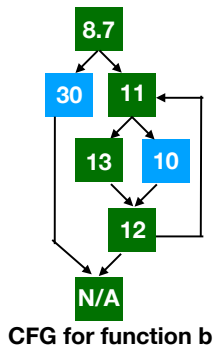
$$[(1+30)^{-1} + (2+10)^{-1}]^{-1}$$



# 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  **$10 * \text{FLTD}$**
- 3 For each BB, compute harmonic mean of (length of shortest path to any function-calling BB +  $10 * \text{FLTD}$ ).



# Runtime

```
Input:  $\mathbb{S}$  // a finite set of seeds
Input:  $\mathbb{T}$  // a finite set of target sites
Output:  $\mathbb{S}'$  // a finite set of buggy seeds

1  $\mathbb{S}' \leftarrow \emptyset$ 
2  $SeedQueue \leftarrow \mathbb{S}$ 
3  $Graphs \leftarrow GraphExt(Code)$ 
4  $BBdistance \leftarrow DisCalcu(\mathbb{T}, Graphs)$ 
5 while  $!signa1$  do
6    $s \leftarrow Dequeue(SeedQueue)$ 
7    $e \leftarrow AssinEnergy(s, t_{elapsed}, BBdistance)$ 
8   for  $i \leftarrow 1$  to  $e$  do
9      $s' \leftarrow Mutation(s)$ 
10     $trace \leftarrow Execution(s')$ 
11    if  $s'$  crashes then  $\mathbb{S}' \leftarrow \mathbb{S}' \cup s'$ 
12    if  $IsInteresting(s')$  then
13       $distance \leftarrow SeedEva(trace, BBdistance)$ 
14       $SeedQueue \leftarrow SortInser(distance, s', SeedQueue)$ 
15 return  $\mathbb{S}'$ 
```

## ① Background

## ② Theory

## ③ Work

## ④ References

## ① Background

## ② Theory

## ③ Work

## ④ References



- [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!