Directed Greybox Fuzzing

스마트보안전공 202334843 이동하

Directed Greybox Fuzzing

2017 ACM SIGSAC Conference on Computer and Communications Security

https://mboehme.github.io/

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

https://strongcourage.github.io/

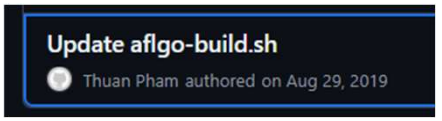
https://thuanpv.github.io/

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

https://abhikrc.com/



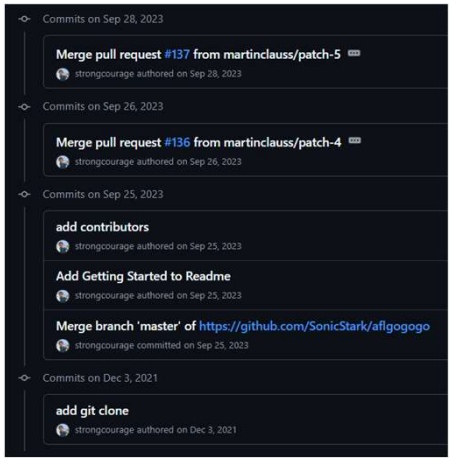


https://mboehme.github.io/

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

https://thuanpv.github.io/

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg



Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

https://strongcourage.github.io/

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

https://abhikrc.com/



Directed Greybox Fuzzing

2017 ACM SIGSAC Conference on Computer and Communications Security

https://mboehme.github.io/

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

https://thuanpv.github.io/

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg

Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

https://strongcourage.github.io/

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

https://abhikrc.com/

[CCS'17]

Directed Greybox Fuzzing

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury

(One-line Abstract) Outperforming directed symbolic execution using simulated annealing and a novel distance metric that is pre-computed.

24th ACM Conference on Computer and Communications Security (CCS) 2017, Accepted for publication.

Note: AFLGo which implements directed greybox fuzzing into AFL is available at https://github.com/aflgo/aflgo.

Update: Interactive tutorial explaining the algorithms behind AFL and AFLGo: https://www.fuzzingbook.org/html/GreyboxFuzzer.html!

Slides @Slideshare, **Presentation (ACM CCS'17)** @Youtube

Directed Greybox Fuzzing

2017 ACM SIGSAC Conference on Computer and Communications Security

https://mboehme.github.io/

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

https://strongcourage.github.io/

https://thuanpv.github.io/

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

https://abhikrc.com/

[CCS'16]

Coverage-based Greybox Fuzzing as Markov Chain

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury

(One-line Abstract) Effective path exploration without program analysis

23rd ACM Conference on Computer and Communications Security (CCS) 2016. pp. 1032-1043

Note: The journal extension has been accepted at the SE flagship journal IEEE TSE!

(April 2016), 345–360.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 1032–1043.

AFL++ code review

[1] 기존 Grey Box Fuzzer 의 장점

[2] Grey Box Fuzzer 의 한계 (not effectively directed)

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

[4] 기존 시도 분석&한계 (directed symbolic execution -> overhead)

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

[6] 구현
- 구현 1. Heartbleed 사례 연구
- 구현 2. distance measure, simulated annealing
- 구현 3. AFLGo 구현 세부 사항
- 구현 4. AFLGo patch test, katch와 비교
- 구현 5. OSS-FUZZ 통합
- 구현 6. crash reproduce, AFL/BugRedux와 비교

[7] 평가 지표 제시

[1] 기존 Grey Box Fuzzer 의 장점

BlackBox Fuzzing : no program analysis, no feedback

WhiteBox Fuzzing : mostly program analysis

GreyBox Fuzzing : no program analysis, coverage feedback

[1] 기존 Grey Box Fuzzer 의 장점

Faster than whitebox

BlackBox Fuzzing : no program analysis, no feedback

WhiteBox Fuzzing : mostly program analysis

GreyBox Fuzzing : no program analysis, coverage feedback

Faster than whitebox

[1] 기존 Grey Box Fuzzer 의 장점

BlackBox Fuzzing : no program analysis, no feedback

WhiteBox Fuzzing : mostly program analysis Testing specific paths, detailed bug finding

GreyBox Fuzzing : no program analysis, coverage feedback

[1] 기존 Grey Box Fuzzer 의 장점

BlackBox Fuzzing : no program analysis, no feedback

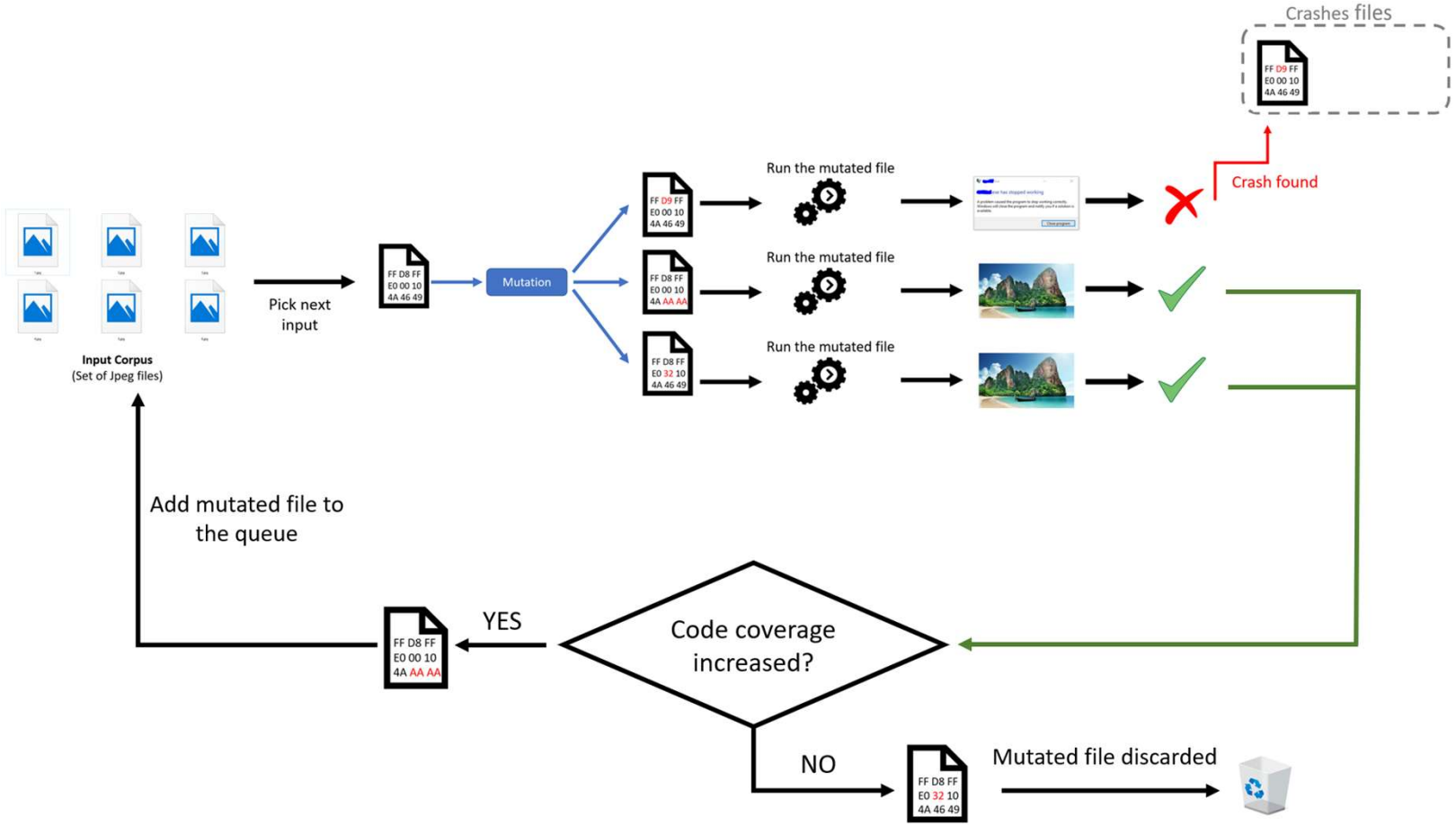
WhiteBox Fuzzing : mostly program analysis

GreyBox Fuzzing : no program analysis, coverage feedback



All program analysis at instrumentation time
-> No Runtime Overhead

[1] 기존 Grey Box Fuzzer 의 장점



[2] Grey Box Fuzzer 의 한계 (not effectively directed)

BlackBox Fuzzing : no program analysis, no feedback

WhiteBox Fuzzing : mostly program analysis

GreyBox Fuzzing : no program analysis, coverage feedback



can’t be directed

[2] Grey Box Fuzzer 의 한계 (not effectively directed)

What is Directed Fuzzing?

특정 경로를 집중적으로 탐색

목표 지향적 경로 탐색 = 특정 함수, 코드 패치 구간, 의심 위치

Ohters : 전체 경로를 탐색

[2] Grey Box Fuzzer 의 한계 (not effectively directed)

What is Directed Fuzzing?

특정 경로를 집중적으로 탐색

목표 지향적 경로 탐색 = 특정 함수, 코드 패치 구간, 의심 위치

Static Information

Ohters : 전체 경로를 탐색

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

특정 경로를 집중적으로 탐색
목표 지향적 경로 탐색 = 특정 함수, 코드 패치 구간, 의심 위치

patch testing : commit 등을 대상으로 함
crash reproduction : stack trace 등을 대상으로 함
SA Report Verification : “dangerous” location 을 대상으로 함
Information Flow Detection : private information이 흐르는 지점을 대상으로 함

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

program analysis : identify program path

symbolic execution : path condition check

constraint solve : find an input

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

symbolic execution

state{stmt, σ, π}

stmt : statement, 다음에 평가할 코드를 저장(분기, 점프 등)

σ : symbolic store, 프로그램의 변수를 심볼릭 값 α_β 과 고정값을 매핑하여 저장

매핑관계(association)를 ↦ 기호로 표현합니다.(ex. b↦ α_β)

π : path constraint로 경로 제약 조건이 저장된다. stmt가 실행되기 위해 설정한 α_1 등의 심볼릭 값이 어떤 값이 되어야 하는지 저장된다.
초기에는 ture가 저장되어 있다.

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state A

σ = a ↦ α_a, b ↦ α_b

π = true

stmt = `int x = 1, y = 0;`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state B

σ = x ↦ 1, y ↦ 0
σ = a ↦ α_a, b ↦ α_b
π = true
stmt = `if (a != 0)`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
            y = 3 + x;
            if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state C

σ = x ↦ 1, y ↦ 0
σ = a ↦ α_a, b ↦ α_b
π = $\alpha_a \neq 0$
stmt = `y = 3 + x;`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
            y = 3 + x;
            if (b == 0)
                        x = 2 * (a + b);
        }
        assert(x - y != 0);
}
```

state D

σ = x ↦ 1, y ↦ 0
σ = a ↦ α_a, b ↦ α_b
π = $\alpha_a = 0$
stmt = `assert(x - y != 0);`

state C / D 별개로 내려감

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state D

$\sigma = x \mapsto 1, y \mapsto 0$

$\sigma = a \mapsto \alpha_a, b \mapsto \alpha_b$

$\pi = \alpha_a = 0$

stmt = `assert(x - y != 0);`

1 0

ture

정상 종료

a=0 일 때 정상 종료라는 것을 확인

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state C

σ = x ↦ 1, y ↦ 0
σ = a ↦ α_a, b ↦ α_b
π = $\alpha_a \neq 0$
stmt = `y = 3 + x;`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state E

σ = x ↦ 1, y ↦ 4
σ = a ↦ α_a, b ↦ α_b
π = $\alpha_a \neq 0$
stmt = `if (b == 0)`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
       int x = 1, y = 0;
       if (a != 0) {
            y = 3 + x;
            if (b == 0)
                       x = 2 * (a + b);
        }
        assert(x - y != 0);
}
```

state F

$\sigma = x \mapsto 1, y \mapsto 4$

$\sigma = a \mapsto \alpha_a, b \mapsto \alpha_b$

$\pi = (\alpha_a \neq 0) \wedge (\alpha_b \neq 0)$

stmt = `assert(x - y != 0);`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state F

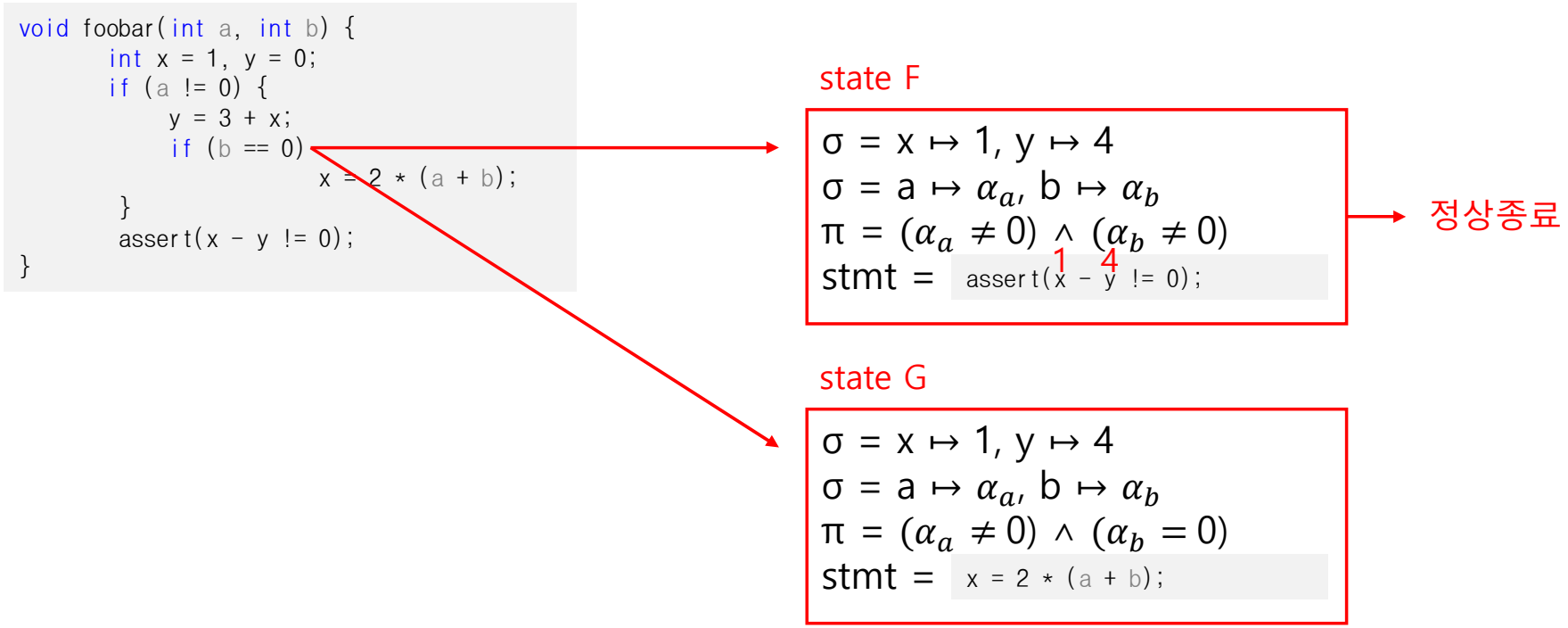
σ = x ↦ 1, y ↦ 4
σ = a ↦ α_a, b ↦ α_b
π = ($\alpha_a \neq 0$) ∧ ($\alpha_b \neq 0$)
stmt = `assert(x - y != 0);`

state G

σ = x ↦ 1, y ↦ 4
σ = a ↦ α_a, b ↦ α_b
π = ($\alpha_a \neq 0$) ∧ ($\alpha_b = 0$)
stmt = `x = 2 * (a + b);`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?



[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
       int x = 1, y = 0;
       if (a != 0) {
            y = 3 + x;
            if (b == 0)
                       x = 2 * (a + b);
        }
        assert(x - y != 0);
}
```

state G

σ = x ↦ 1, y ↦ 4
σ = a ↦ α_a, b ↦ α_b
π = ($\alpha_a \neq 0$) ∧ ($\alpha_b = 0$)
stmt = `x = 2 * (a + b);`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
       int x = 1, y = 0;
       if (a != 0) {
            y = 3 + x;
            if (b == 0)
                       x = 2 * (a + b);
        }
        assert(x - y != 0);
}
```

state H

σ = x ↦ 2*(α_a+α_b), y ↦ 4
σ = a ↦ α_a, b ↦ α_b
π = ($\alpha_a \neq 0$) ∧ ($\alpha_b = 0$)
stmt = `assert(x - y != 0);`

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
             y = 3 + x;
             if (b == 0)
                        x = 2 * (a + b);
         }
         assert(x - y != 0);
}
```

state H

σ = x ↦ 2*(α_a+α_b), y ↦ 4
σ = a ↦ α_a, b ↦ α_b
$\pi = (\alpha_a \neq 0) \wedge (\alpha_b = 0)$
stmt = `assert(x - y != 0);`

$\pi = (\alpha_a \neq 0) \wedge (\alpha_b = 0) \wedge \{\ 2(\alpha_a+\alpha_b) - 4 \neq 0\ \}$ ⟶ 정상종료 조건

[3] 그 한계를 해결하기 위한 방법 (directed fuzzing, CFG analyze)

What is Directed Fuzzing?

constraint solve : find an input

$$\pi = (\alpha_a \neq 0) \wedge (\alpha_b = 0) \wedge \{\ 2(\alpha_a + \alpha_b) - 4 \neq 0\ \}$$

이러한 조건 π 를 구하는 것을 constraint solve라고 한다.

SAT, SMT, etc.

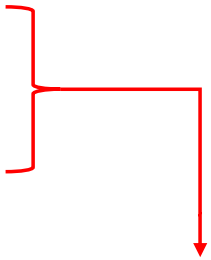
[4] 기존 시도 분석&한계 (directed symbolic execution -> overhead)

What is Directed Fuzzing?

program analysis : identify program path

symbolic execution : path condition check

constraint solve : find an input



memory 제약이 존재

computing power 제약 존재

실행환경과의 상호작용 배재(커널, 네트워크)

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

최적화 문제(Optimization problems)란 여러개의 선택가능한 후보 중에서 최적의 해(Optimal value) 또는 최적의 해에 근접한 값을 찾는 문제를 일컫는다.

일반적으로 기계학습 분야에서는 비용함수(Cost function)를 최소화 또는 최대화 시키는 모델의 파라미터(parameter)를 구하게 되는데, 이것은 최적화 문제로 정의될 수 있다.

Ref :
https://wikidocs.net/17203#:~:text=%EC%B5%9C%EC%A0%81%ED%99%94%20%EB%AC%B8%EC%A0%9C(Optimization%20problems)%EB%9E%80,%EC%9D%84%20%EC%B0%BE%EB%8A%94%20%EB%AC%B8%EC%A0%9C%EB%A5%BC%20%EC%9D%BC%EC%BB%AB%EB%8A%94%EB%8B%A4.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

1. Instrumentation Time

- Call Graph (CG)와 Control-Flow Graph (CFG) 추출

- 각 BB(Basic Block)에 대해 목표 지점까지의 거리 계산

- 프로그램 계측: 거리 값 집계를 위한 수정

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

2. RunTime, for each input

- collect coverage and distance information

- Distance 기반 퍼징 시간 결정

입력값이 목표 지점에 가까운 경우, 이를 더 오랫동안 퍼징합니다.

입력값이 목표 지점에서 멀리 떨어져 있는 경우, 이를 짧게 퍼징합니다.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

1. Instrumentation Time

- Call Graph (CG)와 Control-Flow Graph (CFG) 추출

- 각 BB(Basic Block)에 대해 목표 지점까지의 거리 계산

- 프로그램 계측: 거리 값 집계를 위한 수정

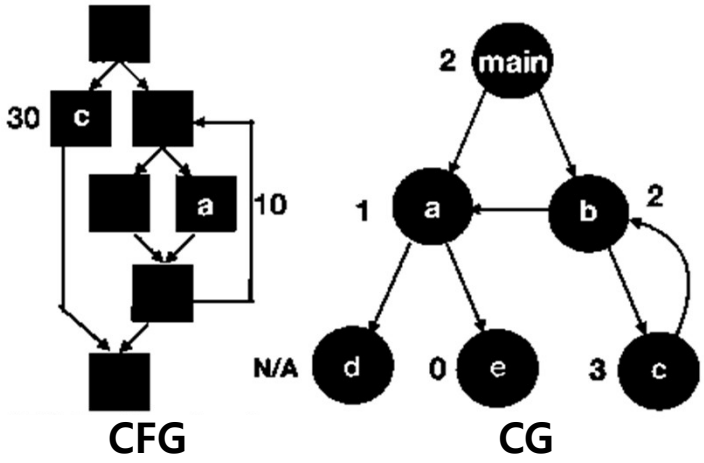
[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

1. Instrumentation Time

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

BB-level target distance : CFG(control flow graph)



[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

1. Instrumentation Time

기존 방식 (Coverage-Based Greybox Fuzzing)

기존의 AFL이나 LibFuzzer 같은 커버리지 기반 그레이박스 퍼저는 코드 커버리지 정보만을 활용하여 입력을 생성하고 유지.
목표 위치(target location)에 대한 정보는 고려되지 않으므로, 퍼징이 특정 취약점으로 집중되지 않고, 탐색이 분산될 수 있다.



거리 정보 활용:
프로그램 계측 시점(Instrumentation Time)에 목표 지점까지의 거리를 계산하고, 이 정보를 런타임에서 활용합니다.

퍼징을 최적화 문제로 재구성하여, 목표 지점에 더 가까운 입력값에 자원을 더 집중적으로 할당하는 방식으로 탐색 효율성을 높입니다.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

2. RunTime, for each input

- collect coverage and distance information

- Distance 기반 퍼징 시간 결정

입력값이 목표 지점에 가까운 경우, 이를 더 오랫동안 퍼징합니다.

입력값이 목표 지점에서 멀리 떨어져 있는 경우, 이를 짧게 퍼징합니다.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

2. RunTime, for each input

Seed Distance : seed input 과 basic block(BB) 거리 값의 평균
Instrumentation Time에 구한 값을 활용

BB-level Distance Values:실행된 기본 블록들에 대해 사전에 계산된 거리 값.
Executed BBs Count:실행된 기본 블록의 총 개수.

Seed Distance = (BB-level Distance Values)/(Executed BBs Count)

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

2. RunTime, for each input

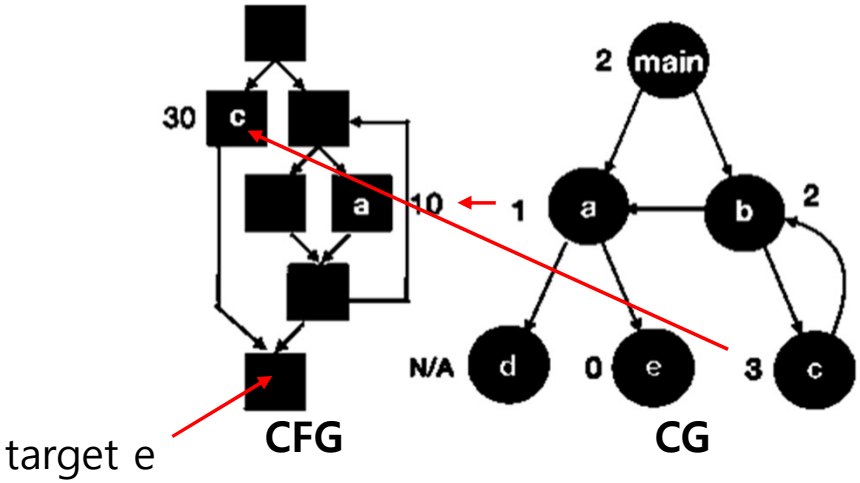
계산된 Seed Distance는 입력이 목표 위치에 얼마나 가까운지를 나타낸다.

Seed Distance가 낮을수록(목표와 가까움) 더 많은 자원을 할당.
Seed Distance가 높을수록(목표와 멀어짐) 자원 할당을 줄임.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

~~2. RunTime, for each input~~



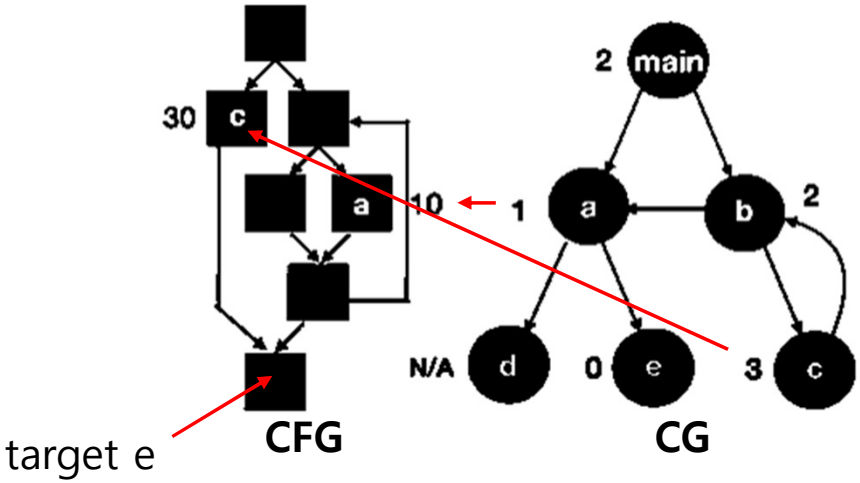
1. Target BB assign distance 0

2. Call(BB) function assign 10*FLTD

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

~~2. RunTime, for each input~~



1. Target BB assign distance 0

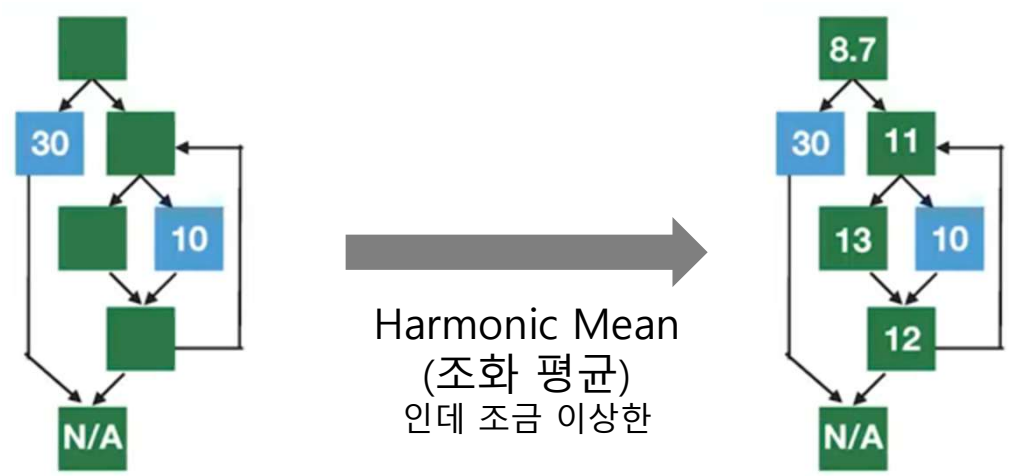
2. Call(BB) function assign 10*FLTD

Function-Level Target Distance

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

~~2. RunTime, for each input~~



[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

~~2. RunTime, for each input~~



$$\text{BB Distance} = \left(\sum_{i=1}^{n} \frac{1}{\text{길이}(\text{Path Length to Function Calling BB}) + 10 \times \text{FLTD}} \right)^{-1}$$

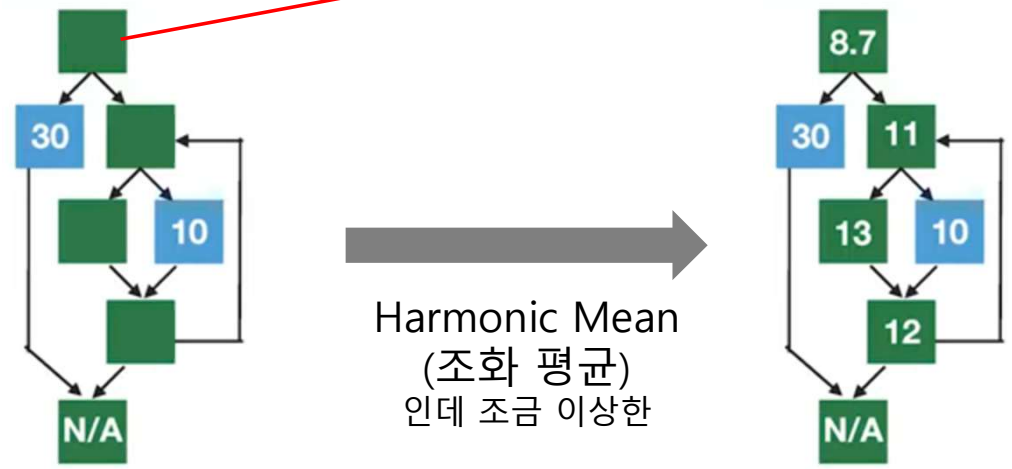
[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

~~2. RunTime, for each input~~

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

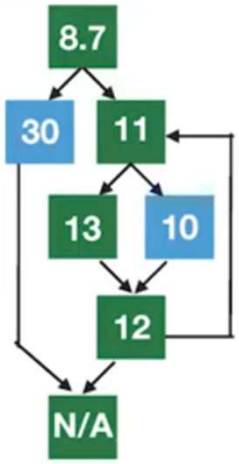
(((1 + 30)^(-1)) + ((2 + 10)^(-1)))^(-1) =
8.6511627907

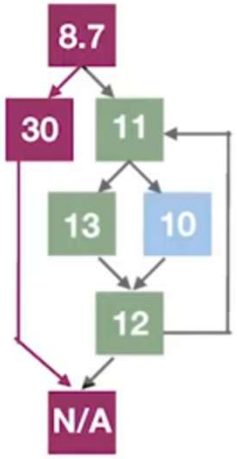


Harmonic Mean
(조화 평균)
인데 조금 이상한

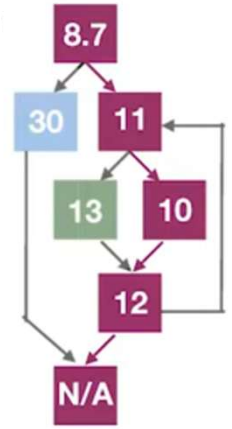
[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem





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



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

Seed Distance = sum(BB distance)/(Number of executed BB)

= Avg(BB distance) ☺

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

2. RunTime, for each input

계산된 Seed Distance는 입력이 목표 위치에 얼마나 가까운지를 나타낸다.

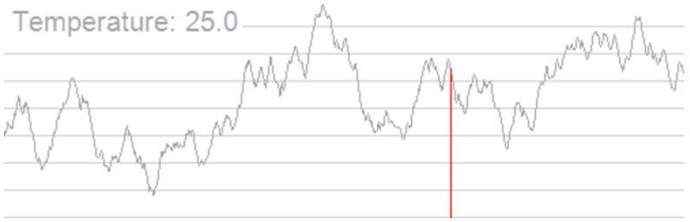
Seed Distance가 낮을수록(목표와 가까움) 더 많은 자원을 할당.
Seed Distance가 높을수록(목표와 멀어짐) 자원 할당을 줄임.

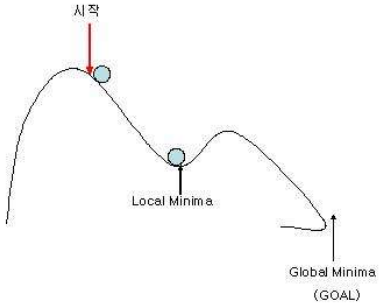
[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

1. (금속이나 유리를 불에 달궜다가 천천히 식혀) 강화시키다, 담금질하다.
2. (정신 등을) 단련하다, 강화하다.





Simulated Annealing은 아주 어려운 문제를 풀기 위해 점진적으로 그 해(solution)에 가까운 방향으로 이동하되 적은 확률로 예상되는 해에 아주 먼 방향으로 이동하는 것 (AI에서 본 듯 한)

논문에서 목적 : 목표 프로그램 위치(target locations)에 최대한 빠르게 도달하기 위함

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

초기에는 랜덤하게 탐색하며 explore을 강조하지만, 점진적으로 검색을 최적화해 exploit 단계로 전환.

에너지 할당: SA는 seed(입력 데이터)에 할당된 에너지를 제어하여, 타겟에 가까운 seed에 더 많은 에너지를 할당하고, 멀리 있는 seed는 에너지를 줄임.

초기에는 "온도(temperature)" 값이 높아 잘못된 해도 허용하며 탐험을 유도.
시간이 지나면서 온도가 낮아지고 더 나은 해결책만 선택하며 수렴.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

explore(초기)

초기 단계에서는 프로그램의 다양한 경로를 탐색.
random mutation을 통해 새로운 입력을 생성하며 새로운 경로를 개척.
seed distance가 멀더라도 균일한 에너지를 할당하여 다양성을 유지.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

exploit(후기)

특정 입력이 타겟에 가까운 경우, 해당 입력에 높은 에너지를 할당하여 집중적으로 탐구.
Annealing Schedule이 온도를 낮추면서, 탐색 공간을 줄이고 최적의 솔루션에 점점 더 집중.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

annealing-based power schedule -> AFLGo

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

이해가 안가는 포인트 :

조화 평균은 타겟과의 거리에서 근접성을 더 명확히 반영한다고 함
그러나 근거를 이해할 수 없었음.
[추후에 기회가 된다면 다시 살펴 볼 지점]

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

기본 블록 m에서 타겟 블록 집합 T_b까지의 거리 $d_b(m, T_b)$

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

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

기본 블록 m에서 타겟 블록 집합 T_b까지의 거리 $d_b(m, T_b)$

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

→ m이 target block에 속할 경우

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

기본 블록 m에서 타겟 블록 집합 T_b까지의 거리 $d_b(m, T_b)$

최소 타겟 거리 * c

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

m이 target 함수를 call 할 수 있을 때,

c=10
CG->CFG 거리 확대 상수(논문에서 상수 10으로 정의함, 이유는 모르겠음)

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

기본 블록 m에서 타겟 블록 집합 T_b까지의 거리 $d_b(m, T_b)$

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

→ 다른 블럭 거쳐서 target에 도달하는 경로 평균거리

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

exercised basic blocks. We define the seed distance $d(s, T_b)$ as

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

seed(s)

실행경로 $\xi(s)$

크시..?

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

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

정규화

$$\tilde{d}(s, T_b) = \frac{d(s, T_b) - \text{minD}}{\text{maxD} - \text{minD}}$$

= 0~1 구간 출력

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Simulated Annealing

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

정규화

$$\text{minD} = \min_{s' \in S} [d(s', T_b)]$$

$$\text{maxD} = \max_{s' \in S} [d(s', T_b)]$$

$$\tilde{d}(s, T_b) = \frac{d(s, T_b) - \text{minD}}{\text{maxD} - \text{minD}}$$

= 0~1 구간 출력

0 : 가까운 경우
1 : 먼 경우

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem



instruction time
(BB distance processing & more)

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

only seed distance processing

run time

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Markov Chain을 사용하여 프로그램 경로(path)의 탐색 과정을 모델링

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Markov Chain in AI

상태(State): 시스템이 가질 수 있는 가능한 상태들의 집합.
(예: 주식 시장에서의 "상승", "하락", "유지" 상태)

전이 확률(Transition Probability): 한 상태에서 다른 상태로 이동할 확률

상태 전이 행렬(Transition Matrix): 모든 상태 간의 전이 확률을 행렬로 표현한 것.

[5] 새로운 해결 방법 제시 (directed greybox fuzzing, DGF+DSE)

Directed Fuzzing as optimization problem

Markov Chain in greybox fuzzing

상태(State): 프로그램 실행 경로, 실행된 블럭

전이 확률(Transition Probability): 한 경로에서 다른 경로로 이동할 확률

상태 전이 행렬(Transition Matrix): 프로그램의 실행 흐름과 새로운 경로로 이동 할 가능성

[6] 구현

Simulated Annealing

온도 조정 : $T_{\exp}$

1미만의 값, 일반적으로 0.8 이상 0.99이하의 값을 쓴다고 한다

냉각 스케줄 : $T_{\exp} = T_0 \cdot \alpha^k$ → 현재 냉각 단계(시간 t와 비례)

초기 값 1

[6] 구현

Simulated Annealing

임계온도 : 0.05 일 경우,

$T_k \leq 0.05$ 에서 explore -> exploit 으로 전환

k_x exploit으로 넘어가는 냉각 단계

$k_x = \log(0.05) / \log(\alpha)$

$\cdot \alpha^k$ → 현재 냉각 단계(시간 t와 비례)

시간에 따른 온도로 방정식 정리 : $T_{\exp} = \alpha^{\frac{t}{t_x}\frac{\log(0.05)}{\log(\alpha)}}$

$= 20^{-\frac{t}{t_x}}$

[6] 구현

Simulated Annealing

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$

seed(s)의 energy

[6] 구현

Simulated Annealing

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$

seed(s)의 정규화된 distance

[6] 구현

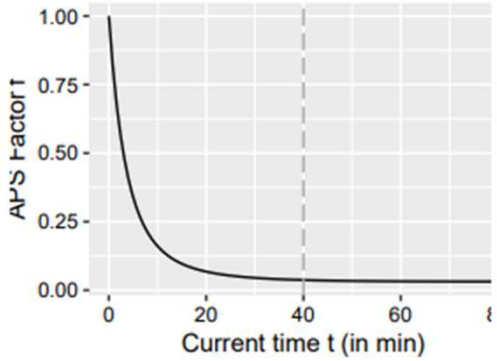
Simulated Annealing

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$

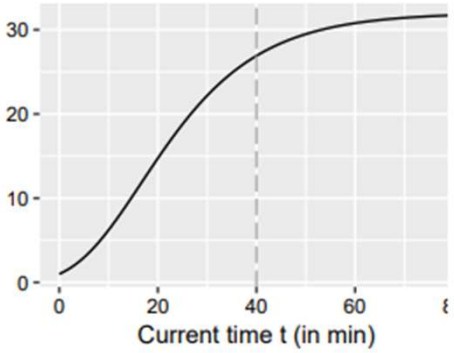
현재 온도 현재 온도

[6] 구현

Simulated Annealing



(a) Distance $\tilde{d}(s, T_b) = 1$



(b) Distance $\tilde{d}(s, T_b) = 0$

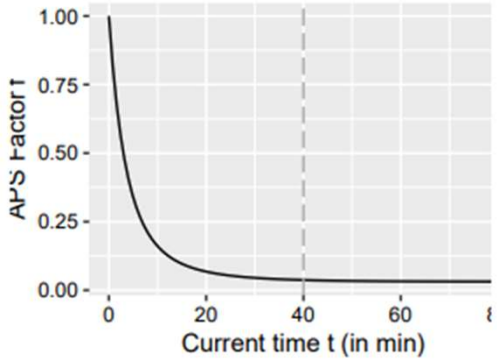
가까움

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$

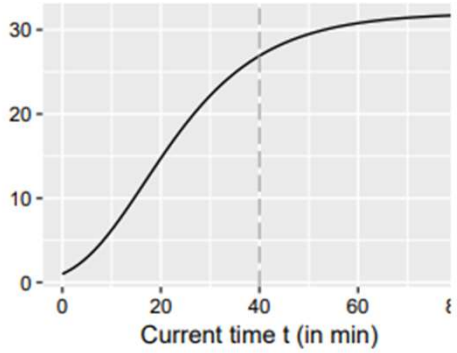
t에 따라 식어감(초기 : 1)

[6] 구현

Simulated Annealing



(a) Distance $\tilde{d}(s, T_b) = 1$



(b) Distance $\tilde{d}(s, T_b) = 0$

가까움

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5T_{\text{exp}}$$

t에 따라 식어감(초기 : 1)

t에 따라 식어감(후기 : 0) -> 가까운 시드에 높은 에너지

[6] 구현

Simulated Annealing

$$\hat{p}(s, T_b) = p_{\text{afl}}(s) \cdot 2^{10 \cdot p(s, T_b) - 5}$$

afl power schedule(APS)
기본 AFL 값

$\hat{p}(s, T_b)$

APS와 통합된 최종 에너지

$$f = 2^{10(p(s, T_b) - 0.5)}$$

AFL의 기존 에너지에 대한 배수 조정 값(scale)

[6] 구현

Simulated Annealing

AFLGo 구성 요소

Graph Extractor (그래프 추출기)
Distance Calculator (거리 계산기)
Instrumentor (계측 도구)
Fuzzer (퍼저)

[6] 구현

Simulated Annealing

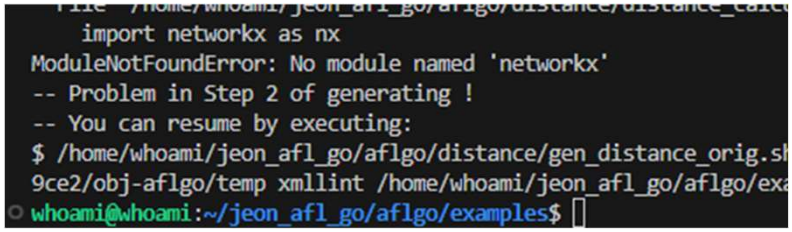
AFLGo 구성 요소

Graph Extractor (그래프 추출기) Call Graph (CG)와 Control-Flow Graphs (CFGs)를 생성

Distance Calculator (거리 계산기) Python 스크립트로 구현되며, networkx 패키지를 사용

Instrumentor (계측 도구)

Fuzzer (퍼저)



[7] 평가 지표 제시

Katch vs AFLGo:
•**커버리지 비교**: AFLGo는 Katch보다 13% 더 많은 패치 변경된 기본 블록(BBs)을 커버.
•**세부 결과**:
- Binutils: Katch는 198, AFLGo는 223개의 변경된 BBs 커버.
- Diffutils: Katch는 63, AFLGo는 64개의 변경된 BBs 커버.

- 두 기술을 결합하면 각각보다 42%, 26% 더 많은 목표를 커버.

AFLGo의 성과:
•13개의 신규 버그 발견(7개 CVE 포함).
•기존 Katch가 발견한 7개의 버그 중 4개도 탐지.
•주요 발견 유형: 버퍼 오버플로우, 널 포인터 참조 등.

AFLGo를 Google의 **OSS-Fuzz**에 통합하여 실용성 확인

[7] 평가 지표 제시

	CVE-ID	Tool	Runs	μTTE	Factor	$\hat{A}_{12}$
LibPNG [54]	2011-2501	AFLGo	20	0h06m	2.81	**0.79**
		AFL	20	0h18m	–	–
	2011-3328	AFLGo	20	0h40m	4.48	**0.94**
		AFL	18	3h00m	–	–
	2015-8540	AFLGo	20	0m26s	10.66	**0.87**
		AFL	20	4m34s	–	–
Binutils [6]	2016-4487	AFLGo	20	0h02m	1.64	0.59
		AFL	20	0h04m	–	–
	2016-4488	AFLGo	20	0h11m	1.53	**0.72**
		AFL	20	0h17m	–	–
	2016-4489	AFLGo	20	0h03m	2.25	**0.68**
		AFL	20	0h07m	–	–
	2016-4490	AFLGo	20	1m33s	0.64	**0.31**
		AFL	20	0m59s	–	–
	2016-4491	AFLGo	5	6h38m	0.85	0.44
		AFL	7	5h46m	–	–
	2016-4492	AFLGo	20	0h09m	1.92	**0.81**
		AFL	20	0h16m	–	–
	2016-6131	AFLGo	6	5h53m	1.24	0.61
		AFL	2	7h19m	–	–

	Mean $\hat{A}_{12}$	*Median* $\hat{A}_{12}$
AFLGo	0.66	**0.68**

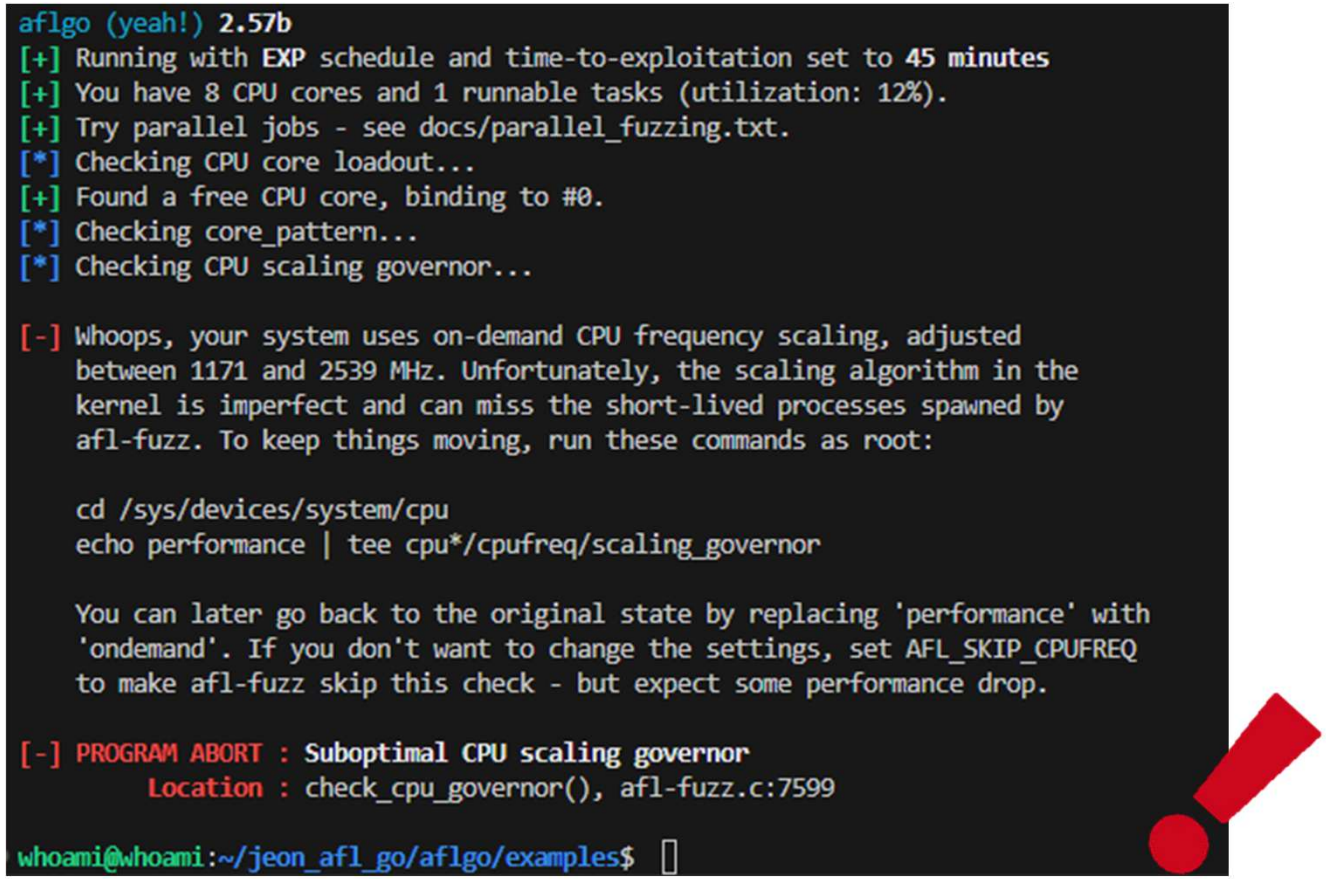
[7] 평가 지표 제시

일부 기본 블록(BBs)은 특정 운영 체제나 아키텍처에서만 도달 가능.
특정 블록은 환경에 따라 실행되지 않아 탐지되지 않음.

함수 포인터 사용: 많은 기본 블록이 함수 포인터로 간접 호출되거나 점프하여 접근 가능.
이런 경우, 호출 그래프(Call Graph)나 제어 흐름 그래프(CFG)에서 탐지되지 않음.
AFLGo와 Katch 모두 이런 유형의 블록 탐지에 한계가 있음.

프로그램이 복잡한 입력 구조를 요구할 때, 탐색 공간이 너무 커져 탐지가 어려움.
특히 Binutils의 경우, 특정 구조를 가진 입력 파일만이 일부 목표를 실행 가능.
예: 특정 섹션이 포함된 파일만 허용.

regression test suite(기존 것)자체의 한계가 있다는데, 이해 못함


aflgo (yeah!) 2.57b
[+] Running with EXP schedule and time-to-exploitation set to 45 minutes
[+] You have 8 CPU cores and 1 runnable tasks (utilization: 12%).
[+] Try parallel jobs - see docs/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Checking CPU scaling governor...
[-] Whoops, your system uses on-demand CPU frequency scaling, adjusted
between 1171 and 2539 MHz. Unfortunately, the scaling algorithm in the
kernel is imperfect and can miss the short-lived processes spawned by
afl-fuzz. To keep things moving, run these commands as root:
cd /sys/devices/system/cpu
echo performance | tee cpu*/cpufreq/scaling_governor
You can later go back to the original state by replacing 'performance' with
'ondemand'. If you don't want to change the settings, set AFL_SKIP_CPUFREQ
to make afl-fuzz skip this check - but expect some performance drop.
[-] PROGRAM ABORT : Suboptimal CPU scaling governor
Location : check_cpu_governor(), afl-fuzz.c:7599
whoami@whoami:~/jeon_afl_go/aflgo/examples$

```
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCombiningHalfMarks.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCombiningMarksforSymbols.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsControlPictures.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCurrencySymbols.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCypriotSyllabary.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCyrillic.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsCyrillicSupplement.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsDeseret.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsDevanagari.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsDingbats.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsEnclosedAlphanumerics.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsEnclosedCJKLettersandMonths.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsEthiopic.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsGeneralPunctuation.dot..
Computing distance for /home/whoami/jeon_afl_go/aflgo/examples/libxml2_ef709ce2/obj-aflgo/temp/dot-files/cfg.xmlUCSIsGeometricShapes.dot..
```

```
aflgo (yeah!) 2.57b
[+] Running with EXP schedule and time-to-exploitation set to 45 minutes
[+] You have 8 CPU cores and 2 runnable tasks (utilization: 25%).
[+] Try parallel jobs - see docs/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:dtd1'...
[*] Spinning up the fork server...
[+] All right - fork server is up.

[-] The program took more than 1000 ms to process one of the initial test cases.
    This is bad news; raising the limit with the -t option is possible, but
    will probably make the fuzzing process extremely slow.

    If this test case is just a fluke, the other option is to just avoid it
    altogether, and find one that is less of a CPU hog.

[-] PROGRAM ABORT : Test case 'id:000000,orig:dtd1' results in a timeout
        Location : perform_dry_run(), afl-fuzz.c:2915
```

(State of) The Art of War: Offensive Techniques in Binary Analysis

스마트보안전공 202334843 이동하

바이너리 분석 : 컴파일 된 바이너리 파일을 분석

offensive analysis 관점에서 바이너리 분석하는 오픈소스 분석 프레임워크(angr)

바이너리 분석 : 컴파일 된 바이너리 파일을 분석

offensive analysis 관점에서 바이너리 분석하는 오픈소스 분석 프레임워크

바이너리 분석의 필요성 :

Interpreted 언어도 최적화를 위해 바이너리화 되거나,
JIT(Just-In-Time) 컴파일러를 통해 바이너리로 추출되는 경우가 있다.

성능이 중요한 OS level, IoT 등의 어플리케이션은 compiler-based languages 로 작성된다.

문제점:

컴파일러도 완벽하지 않다.

소스코드 분석에서 발견되지 않은 결함도 컴파일 타임이 지나면서 생겨날 수 있다.
실 사례: Xcode Ghost 버그

문제점:

컴파일러도 완벽하지 않다.

소스코드 분석에서 발견되지 않은 결함도 컴파일 타임이 지나면서 생겨날 수 있다.
실 사례: Xcode Ghost 버그

기존 연구가 프로토타입으로 끝났다.
후속 연구가 이어지기 어려웠고, 이로 인해 발전이 어려웠다. 기법의 비교 또한 불명확하다.
일관성있는 모델이 없었다.

솔루션(논문 주제):

angr 프레임워크를 개발
바이너리 분석 기법을 쉽게 비교, 개발 할 수 있는 오픈소스 프레임워크

동적/정적을 모두 사용하여 기존 도구들을 기본 제공한다.

DARPA Cyber Grand Challenge(CGC) 데이터셋을 통하여 상대적 효과를 평가할 수 있다.

구현의 차이를 최소화하여, 이론적 차이에 기반한 비교가 가능하다.
논문에서는 이를 공격적 분석 기법을 재현하면서 입증하였다.

[1] 자동화된 바이너리 분석

다양한 기법마다 해소해야 할 제약이 존재한다. 어떤 어려움이 있는지를 아래와 같이 정리한다.

제약 사항은 Trade-Off 관계로 생각할 수 있다.

*Trade-Off : 완전 고용과 물가 안정의 관계.
실업률을 줄이면 물가가 상승하고, 물가를 안정시키면 실업률이 높아진다는 모순적 관계를 이르는 말

상충관계. 하나를 얻으면 다른 하나를 잃을 수 있는 관계

[1] 자동화된 바이너리 분석

Coverage와 Replayability

재현 가능성이 높다 -> 모든 실행 흐름을 추적해야 한다 -> 분석할 수 있는 코드 범위가 제약된다

재현 가능성이 낮다 -> 많은 코드를 분석할 수 있다 -> 발견된 취약점이 재현되지 않을 가능성이 높다
오탐(False Positive)의 가능성이 있다

*다른 관점

오탐 vs. 누락 :

재현 가능성이 낮은 분석은 오탐을 줄이는 대신 누락(False Negative)이 늘어날 수 있다.

[1] 자동화된 바이너리 분석

Speed(Resource)와 Insight

깊은 insight에는 높은 resource가 요구된다.

- Dynamic Analysis는 빠르게 코드를 추적, 그러나 발견한 동작을 이해하지 못함
- Symbolic Analysis는 입력 프로그램과 동작간의 관계까지 이해한다. 그러나 자원이 많이 필요하며, 실행 시간이 길어진다.

[1] 자동화된 바이너리 분석

Replayability 기술적 설명

Scalability 문제가 있다. 모든 동작을 추적하려만 많은 리소스가 필요하다.
따라서 일부 정보를 삭제하거나, 완전성을 희생하여야 한다.

완전성(soundness)==모든 취약점을 찾을 수 있다는 보장

[1] 자동화된 바이너리 분석

Insight 기술적 설명

특정 분기가 실행되기 위한 조건을 분석, 저장

정적분석에서 도메인 데이터가 부족 할 경우, 낮은 insight를 가진다.

*앞에서 설명한 논문과 겹치는 부분은 생략

[1] 자동화된 바이너리 분석

Environment Modeling 기술적 설명

프로그램과 실행 환경 사이의 상호작용을 모델링 해야한다.

Linux를 예시로, 300개 이상의 syscall이 존재한다. 이를 모델링하는 것은 매우 복잡하다.

[1] 자동화된 바이너리 분석

Dynamic Symbolic Execution (DSE) 기술적 설명

다수의 입력 조건을 한번에 분석한다.
그러나 path explosion이라는 문제점이 존재한다. 제약조건을 기반으로 입력값을 생성하는데,
Branch가 많아질 수록 분석 경로가 exponential하게 늘어난다.

[1] 자동화된 바이너리 분석

Trade-Off 정리

정적 분석은 취약점의 위치를 확인하지만, 재현 가능한 입력값 제공에 한계가 있음.

동적 분석은 취약점 재현 입력값 생성이 가능하지만, 단순 퍼징으로는 깊은 취약점을 찾기 어려움.

DSE는 두 접근법의 장점을 결합하지만, 경로 폭발 문제가 주요 한계로 작용.

분석 기법 설계는 **확장성, 정확성, 의미적 통찰력** 간의 균형을 고려해야 함.

[1] 자동화된 바이너리 분석

```
// difficult check for fuzzing
if (magic == 0x31337987) {
  // buffer overflow
  memcpy(buf, data, 100);
}
```

fuzzing에서는 트리거하기 어렵지면 정적으로는 쉬운 케이스

```
if (magic < 100 && magic % 15 == 2 &&
    magic % 11 == 6) {
  // Only solution is 17; safe
  memcpy(buf, data, magic);
}
```

Symbolic Execution으로 테스트 가능한 케이스

```
// Symbolic execution will suffer from
// path explosion
int count = 0;
for (int i = 0; i < 100; i++) {
  if (data[i] == 'Z') {
    count++;
  }
}

if (count >= 8 && count <= 16) {
  // buffer overflow
  memcpy(buf, data, count*20);
}
```

Symbolic Execution으로 테스트하면 path explosion 발생

[1] 자동화된 바이너리 분석

DARPA CGC

자동화된 바이너리 분석 시스템으로 경연

1. Vulnerability Identify
2. Exploit
3. Patch

운영 체제 단순화:

•분석 환경의 복잡성을 줄이기 위해 DECREE OS라는 단순한 운영 체제를 설계.

•DECREE OS는 단 7개의 시스템 호출만 지원(예: 데이터 송수신, 난수 생성 등).

https://www.ll.mit.edu/r-d/datasets/cyber-grand-challenge-datasets

https://github.com/lungetech/cgc-challenge-corpus/blob/master/CADET_00001/src/service.c

Vulnerability

CWE-121: Stack-based Buffer Overflow

The palindrome vulnerability is a vanilla buffer overflow on the stack. The program allocates 64 bytes of stack space for network input and reads up to 128 bytes at a time.

```c
#include <libcgc.h>
#include "libc.h"

#define HI "\nWelcome to Palindrome Finder\n\n"
#define ASK "\tPlease enter a possible palindrome: "
#define YES "\t\tYes, that's a palindrome!\n\n"
#define NO "\t\tNope, that's not a palindrome\n\n"
#define EASTEREGG "\n\nEASTER EGG!\n\n"

int check();

int main(void) {
    int r;

    if (transmit_all(1, HI, sizeof(HI)-1) != 0) {
        _terminate(0);
    }

    while(1){
        if (transmit_all(1, ASK, sizeof(ASK)-1) != 0) {
            _terminate(0);
        }
        r = check();
        if (r == -1){
            break;
        }
        else if (r == 0){
            if (transmit_all(1, NO, sizeof(NO)-1) != 0) {
                _terminate(0);
            }
        }
```

Real world concept CTF 같았다

[1] 자동화된 바이너리 분석

기존 정적분석 정리

프로그램을 실행하지 않고 추론하는 기법
메모리 레이아웃, 실행 경로 등을 추상화하여 분석

그래프 기반/데이터 기반으로 나뉜다.
CFG 등의 graph 기반, 프로그램 내 데이터를 직접 모델링하는 data 기반

한계:
취약점을 찾아도 트리거 여부를 확인하기 어렵다.
데이터 도메인의 부족으로 insigh부족 문제가 있다.
높은 false positive 비율을 가진다.

[1] 자동화된 바이너리 분석

기존 정적분석 정리

CFG 설명은 생략

간접 점프가 CFG의 한계로 남아있다.
대상이 코드내에 명시되지 않은 경우, 런타임에 결정되는 경우 정적 분석이 어렵다
메모리에서 대상 코드를 읽어서 점프하는 경우(computed jump)

호출 대상이 지정한 콜백함수에 따라 점프 타겟이 달라지는 경우(context-sensitive)

OOP에서 vritual table써서 점프하는 경우(object-sensitive)
(런타임에서 결정되기 때문)

[1] 자동화된 바이너리 분석

기존 정적분석 정리

데이터 모델링 활용

VSA(value-set analysis)
특정 시점의 메모리와 레지스터의 상태를 추정하는 정적분석 기법이다
읽고 쓰는 메모리 접근 패턴을 분석하여 overlapping buffer를 찾는 등의 행위를 한다
(overlapping buffer는 잠재적으로 bof가 트리거 될 수 있는 벡터라고 한다)

높은 오탐 문제가 단점이라고 한다

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Concrete Execution: 실제 입력 데이터를 사용하여 경로를 분석.
Symbolic Execution: 입력 데이터를 추상화하여 가능한 모든 경로를 탐구.

입력값 재현 가능성(Replayability)이 높다

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Concrete Execution
프로그램이 특정 입력을 받을 때 어떤 경로를 탐색했는지 분석

테스트 케이스가 필요하다
데이터셋이 크거나 알려지지 않은 경우 어렵거나 불가능.

fuzzing이 대표적인 concrete execution

- 하드코딩된 규칙에 따라 생성된 입력값을 제공.
- 애플리케이션이 충돌하면 해당 입력값이 버그를 유발한 것으로 간주.
- 충돌이 없으면 입력값을 무작위로 변형.

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Concrete Execution
프로그램이 특정 입력을 받을 때 어떤 경로를 탐색했는지 분석

coverage-based fuzzing

- 입력값을 통해 실행된 코드의 범위를 극대화하려는 Fuzzing 기법.
- 더 많은 코드가 실행될수록 취약점 코드가 실행될 가능성이 높다는 원리.

어떤 입력이 해당 코드를 실행했는지에 대한 이해는 없음
여전히 insight는 부족한 방법이라고 한다

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Concrete Execution

프로그램이 특정 입력을 받을 때 어떤 경로를 탐색했는지 분석

Taint-based fuzzing

- 어떤 부분을 변형해야 하는지 이해하기 위해 애플리케이션이 어떻게 입력을 처리하는지 분석.

입력의 **어떤 부분**이 변형되어야 하는지에 대한 이해는 있지만

입력을 **어떻게** 변형해야 하는지는 알지 못한다

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Dynamic Symbolic Execution, DSE

동적 기법으로 프로그램을 에뮬레이션 환경에서 실행하되, 심볼릭 변수를 사용해 입력값을 추상화.
branch을 만나면 실행 경로를 fork하여 각 경로의 조건을 constraint(제약조건)으로 기록.

높은 insight가 있다고 판단.
매우 활발한 연구 분야라고 한다

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Symbolic-assisted Fuzzing

Fuzzing의 속도와 심볼릭 실행의 높은 의미적 통찰력을 결합한 접근법.
Fuzzing이 탐지한 입력값을 심볼릭 실행 엔진에서 분석 및 수정하여 더 깊은 코드를 탐색.

Fuzzing이 실행 가능한 코드를 탐지하고 입력값을 생성.
심볼릭 실행 엔진이 입력값을 분석하여, 이전에 탐지되지 않은 코드 경로를 탐색하도록 입력값 수정.
새로운 테스트 케이스를 생성하여 코드 커버리지를 확대.
(taint analysis 대신 symbolic execution engine이 들어간 듯)

[1] 자동화된 바이너리 분석

기존 동적분석 정리

Under-constrained Symbolic Execution
애플리케이션의 일부분만 실행하여 심볼릭 실행의 확장성을 높이는 기법.
전체 프로그램이 아닌 **부분적인 코드**에 대해 실행.

장점 :
- 행 경로 수를 줄여 심볼릭 실행의 효율성을 향상.
- 특정 코드 영역에서의 잠재적 취약점을 빠르게 식별.

단점 :
- 애플리케이션의 전체 문맥을 제공하지 않으므로, 오탐 가능성이 높다
- 재현 가능성이 낮다(**부분적 코드 실행**으로 인한 trade-off)

[2] 취약점 활용 배경

crash replay -> exploit generate

1. Crash Reproduction
2. Exploit Generation
3. Exploit Hardening

[2] 취약점 활용 배경

crash replay -> exploit generate

1. Crash Reproduction

de-randomization

취약점 탐지 기법은 현실적인 실행 환경이 아닌 단순화된 환경에서 애플리케이션을 실행

난수(Randomness)와 같은 동작을 하드코딩.
예: 실행 파일의 PID, 현재 시간 등을 고정값으로 설정.

실행 시점마다 로직이 달라지는 것을 방지, 입력 값에만 의존하도록
당연히 실제 환경에서 재현되지 않는 문제가 있다.

[2] 취약점 활용 배경

crash replay -> exploit generate

1. Crash Reproduction

Missing Data

de-randomize 한 값으로 crash를 발생(예측 한 값으로 재현 가능)

그러나 real world에서는 여전히 random한 값이기에 추측 불능한 경우

Data Leak Analysis으로 해결 가능하지만 angr는 구현하지 않음

[2] 취약점 활용 배경

crash replay -> exploit generate

1. Crash Reproduction

Missing Relationships

프로그램의 입력-출력 관계를 식별하지 못함으로 인해 발생하는 문제

프로그램이 제공하는 출력값과 요구하는 입력값 간의 **관계**를 고려하지 않아 crash가 재현되지 않음

input specification 을 생성하여 추론가능

Replayer를 사용하여 관계 자동 추론 가능

[2] 취약점 활용 배경

crash replay -> exploit generate

2. Exploit Generation

취약점 발견 기법으로 생성된 충돌 입력 중 일부만 실제로 익스플로잇 가능.
예: 널디레는 현대 운영 체제에서 주소 0번지를 매핑하지 않으므로,
크래시는 발생하지만 익스플로잇 불가능.

익스플로잇 가능한지 확인하여 우선순위 레이팅
Crash to Exploit 기존 연구 존재

[2] 취약점 활용 배경

crash replay -> exploit generate

3. Exploit Hardening

Mitigation bypass -> NX, ASLR 등을 우회해야 함(대부분 새로운 방어기법이 나와서 대응)
현 시점의 mitigation bypass로는 ROP가 있다.

Shellcode injection이 NX로 어려워지고, ASLR로 가젯을 넣을 방법이 어려워지니, 기존에 쓰인 코드조각(=gadget)을 재배치하여 ACE

[3] Analysis Engine 설계

Cross Architecture 지원

현대화 되면서 아키텍처의 수가 늘어나며, 다양한 하드웨어의 호환성이 필요해졌다.
따라서 angr 엔진이 지원해야 하는 다양한 아키텍처가 고려되었다.

[3] Analysis Engine 설계

Cross Platform 지원

운영체제 또한 다양해졌기에, 시스템 콜과 같은 개념을 추상화해야하고, PE/ELF 등 실행파일 형식을 폭 넓게 지원해야 한다.

[3] Analysis Engine 설계

다양한 분석 패러다임 지원

정적, 동적, 심볼릭 익스큐션 등 다양한 기존 기법을 지원해야 한다.

[3] Analysis Engine 설계

사용성

Python기반으로 구현하여 직관적인 api제공

Ipython shell에서 쉽게 사용가능

PyPy를 통해 python JIT 성능 개선 가능

[3] Analysis Engine 설계

설계 철학 - 모듈화

재사용성을 챙기며, 낮은 의존성을 유지하였다.
확장 가능성이 높아졌고, 혼합 사용이 가능해졌다.

효율적 기술 구현 : Veritesting, AEG 등 최신 기술을 짧은 시간 안에 재현 가능.
모듈화 설계의 장점 : 다른 연구자들이 angr의 개별 모듈을 재사용하여 새로운 분석 기법을 개발 가능.
사용자 친화적 인터페이스 : Python API와 IPython 환경을 통해 빠른 프로토타이핑 가능.

[3] Analysis Engine 설계

구현 – 모듈 IR

바이너리를 IR코드로 변환한다. libVEX를 사용하여 VEX라는 IR을 생성하는데,
Firmalice(펌웨어 분석 프레임워크)에 사용된 PyVEX를 사용했다.
이를 사용하면 32/64bit arch의 arm,mips,ppc,x86을 지원한다고 한다.

[3] Analysis Engine 설계

구현 – 모듈 SimuVEX

프로그램 상태를 표현한다(레지스터, 메모리, fd 등)

VEX IR의 symbolic execution engine이다

외에도 다양한 모듈들이 있다.(핵심적인 기술만 정리)

[3] Analysis Engine 설계

구현 – 모듈

기능이 너무 많아 직접 사용해보았다.

```
whoami@whoami:~/jeon_angr$ pip install angr
Collecting angr
  Downloading angr-9.2.135-py3-none-manylinux2014_x86_64.whl.metadata (4.8 kB)
Collecting CppHeaderParser (from angr)
  Downloading CppHeaderParser-2.7.4.tar.gz (54 kB)
```

```
Installing collected packages: z3-solver, unicorn, sortedcontainers, pyelftools, ply, mulpyplexer, mpmath, itanium-demangler, bitarray, unique-log-filter, sympy, pycryptodome, pycparser, psutil, pefile, future, CppHeaderParser, claripy, capstone, bitstring, archinfo, ailment, pyformlang, nampa, cffi, cart, pyvex, cle, angr
```

CTF나 취분에 쓰이는 다양한 패키지들의 디펜던시가 있었다.

Z3 가 claripy(추상화해서 simstate를 보여주는 모듈) frontend 의 symbolic domain으로 사용된다고 한다

[3] Analysis Engine 설계

```
import angr
```
[3] ✓ 2.1s

```
!pip install angr
```
[2] ✓ 29.4s

```
proj = angr.Project('rpc')
```
[5] ✓ 3.5s

가위바위보 c코드 적어줘

아래는 가위바위보 게임을 구현한 C 코드입니다. 사용자와 컴퓨터가 가위바위보를 플레이하며, 승자를 판별하는 간단한 프로그램입니다.

c Copy code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void print_menu() {
    printf("가위바위보 게임\n");
    printf("1. 가위\n");
```

```
whoami@whoami:~/jeon_angr$ nano rpc.c
whoami@whoami:~/jeon_angr$ gcc rpc.c -o rpc
```

```
[8]  proj.arch
     ✓ 0.0s
...  <Arch AMD64 (LE)>

[9]  proj.entry
     ✓ 0.0s
...  4198720

[10] hex(proj.entry)
     ✓ 0.0s
...  '0x401140'

[13] import monkeyhex
     proj.entry
     ✓ 0.0s
...  0x401140
```


```
{'filename': 'rpc',
 'loader': <Loaded rpc, maps [0x400000:0xb07fff]>,
 'arch': <Arch AMD64 (LE)>,
 '_sim_procedures': {0x529dc0: <SimProcedure __libc_start_main>,
  0x582980: <SimProcedure putchar>,
  0x580e50: <SimProcedure puts>,
  0x636550: <SimProcedure __stack_chk_fail>,
  0x5606f0: <SimProcedure printf>,
  0x5460a0: <SimProcedure srand>,
  0x5d99a0: <SimProcedure time>,
  0x562090: <SimProcedure __isoc99_scanf>,
  0x546760: <SimProcedure rand>,
  0x8181d0: <SimProcedure __tls_get_addr>,
  0x900000: <SimProcedure LinuxLoader>,
  0x900008: <SimProcedure _dl_rtld_lock_recursive>,
  0x900010: <SimProcedure _dl_rtld_unlock_recursive>,
  0x900018: <SimProcedure ReturnUnconstrained>,
  0x900020: <SimProcedure _dl_initial_error_catch_tsd>,
  0x900028: <SimProcedure _vsyscall>,
  0x8143f0: <SimProcedure _dl_get_tls_static_info>,
  0x900038: <SimProcedure CallReturn>,
  0x900040: <SimProcedure UnresolvableJumpTarget>,
  0x900048: <SimProcedure UnresolvableCallTarget>},
 'concrete_target': None,
 '_default_analysis_mode': 'symbolic',
 '_exclude_sim_procedures_func': None,
 '_exclude_sim_procedures_list': (),
 'use_sim_procedures': True,
 '_ignore_functions': [],
 'selfmodifying_code': False,
 '_translation_cache': True,
 '_eager_ifunc_resolution': None,
 '_executing': False,
 'entry': 0x401140,
 'storage': defaultdict(list, {}),
```

```
[15]  proj.loader
      ✓ 0.0s
...   <Loaded rpc, maps [0x400000:0xb07fff]>

[16]  proj.loader.shared_objects
      ✓ 0.0s
...   OrderedDict([('rpc', <ELF Object rpc, maps [0x400000:0x404017]>),
                   ('libc.so.6', <ELF Object libc.so.6, maps [0x500000:0x728e4f]>),
                   ('ld-linux-x86-64.so.2',
                    <ELF Object ld-linux-x86-64.so.2, maps [0x800000:0x83b2d7]>),
                   ('extern-address space',
                    <ExternObject Object cle##externs, maps [0x900000:0x97ffff]>),
                   ('cle##tls',
                    <ELFTLSObjectV2 Object cle##tls, maps [0xa00000:0xa1500f]>)])
```

```
[18]  block = proj.factory.block(proj.entry)
      block.pp()
      ✓ 0.0s
...         _start:
    401140  endbr64
    401144  xor     ebp, ebp
    401146  mov     r9, rdx
    401149  pop     rsi
    40114a  mov     rdx, rsp
    40114d  and     rsp, 0xfffffffffffffff0
    401151  push    rax
    401152  push    rsp
    401153  xor     r8d, r8d
    401156  xor     ecx, ecx
    401158  lea     rdi, [main]
    40115f  call    qword ptr [0x403fd8]
```

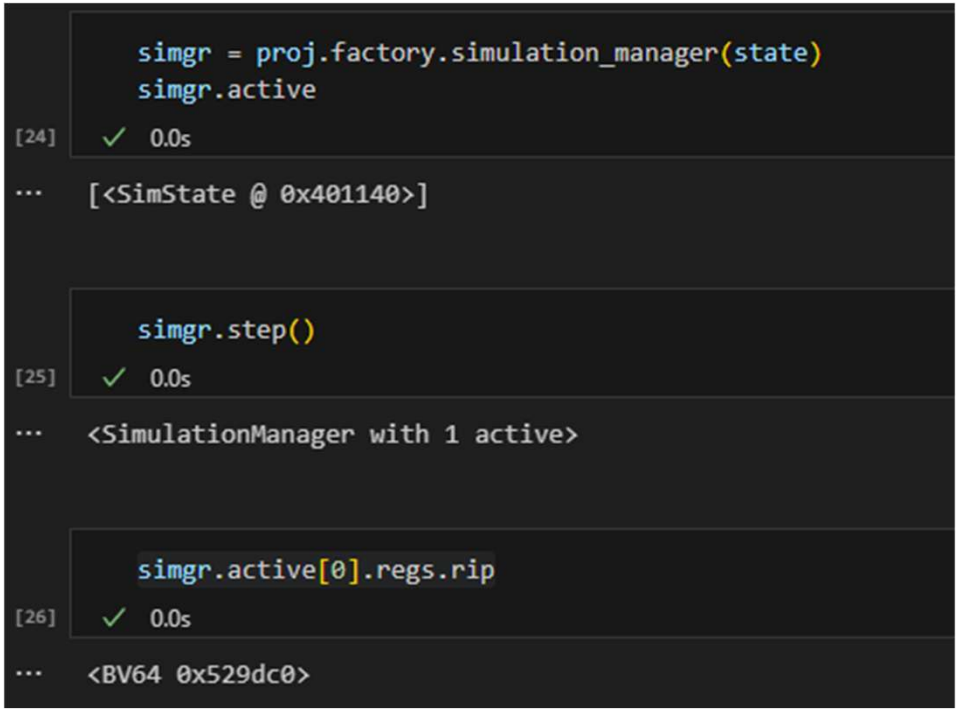
CLE 테스트

```
[19]  state = proj.factory.entry_state()
      ✓ 0.0s

[21]  state.regs.rip
      ✓ 0.0s
...   <BV64 0x401140>

[22]  state.regs.rax
      ✓ 0.0s
...   <BV64 0x1c>
```

```
[23]  state.mem[proj.entry].int.resolved
      ✓ 0.0s
...   <BV32 0xfa1e0ff3>
```


simgr = proj.factory.simulation_manager(state)
simgr.active
[24]
0.0s
[<SimState @ 0x401140>]
simgr.step()
[25]
0.0s
<SimulationManager with 1 active>
simgr.active[0].regs.rip
[26]
0.0s
<BV64 0x529dc0>


simgr.step()
[27]
0.1s
<SimulationManager with 1 active>
simgr.active[0].regs.rip
[28]
0.0s
<BV64 0x40132b>
#WOWOWOWOW
[29]
0.0s

https://github.com/jakespringer/angr_ctf

angr로 ctf문제 푸는 101

*논문에 제시된 코드가 한번에 쉽게 돌아간다는 점이 인상깊다

[장표]
시간남으면 앵거 씨탭 풀이한거 피피티 넣기

[3] Analysis Engine 설계

CFG recovery

~~What is CFG?~~

angr는 CFGFast / CFGAccurate 2가지 CFG 알고리즘을 제공한다(논문기준)

Ref : https://docs.angr.io/en/latest/analyses/cfg.html

```
[30]  cfg = proj.analyses.CFGFast()
      ✓ 7m 5.1s
```

```
cfg = proj.analyses.CFGEmulated(keep_state=True)
✓ 2.1s
```

```
# Generate a static CFG
>>> cfg = p.analyses.CFGFast()

# generate a dynamic CFG
>>> cfg = p.analyses.CFGEmulated(keep_state=True)
```

- CFGFast is much, much faster now. CFGAccurate has been renamed to CFGEmulated.

Ref : https://docs.angr.io/en/stable/appendix/changelog.html
angr 8.18.10.1

CFGFast / CFGAccurate(=CFGEmulated)

[3] Analysis Engine 설계

CFG recovery

forced execution

backward slicing

symbolic execution