Grey-box Concolic Testing on Binary Code

논문 리뷰

01/18 이동하

Grey-box Fuzzing

lagacy : Code Coverage 기반
Coverage가 늘어나는 방향으로 feed back을 주어 input을 생성

Grey-box Fuzzing

lagacy : Code Coverage 기반

Coverage가 늘어나는 방향으로 feed back을 주어 input을 생성

Directed하지 않음

Grey-box Fuzzing

legacy : Code Coverage 기반
Coverage가 늘어나는 방향으로 feed back을 주어 input을 생성
Directed하지 않음

State-of-the-art : generate specific input value
Ex) symbolic execution

Grey-box Fuzzing

legacy : Code Coverage 기반

Coverage가 늘어나는 방향으로 feed back을 주어 input을 생성

Directed하지 않음

State-of-the-art : generate specific input value

Ex) dynamic symbolic execution

Heavy cost!!

Path explosion issue 등으로 실용적이지 못함

Grey-box Fuzzing

novel test case generation technique (grey-box concolic testing) 을 제안함

branch condition 까지 고려하는데, 복잡하게 분석하지는 않는다.

real-world 어플리케이션 대상으로 합리적인 스케일로 condition을 분석할 것 이다.

grey-box concolic testing

White-box 와 달리 branch condition에 대한 input 조건 분석을 근사적인 수준으로 진행

approximated path constraint 는 SMT 의존적이지 않고, 연산량이 과하지 않음

당연히 정확도는 줄어들지만, 새로운 실행 path 탐색에는 충분하다.

(trade-off)

grey-box concolic testing

기존에 분석하던 제약 조건 : path condition

Solve를 위해 컴퓨팅 리소스가 많이 들어간다.

Path explosion 등의 문제로 real-world에서 현실적 사용이 어렵다.

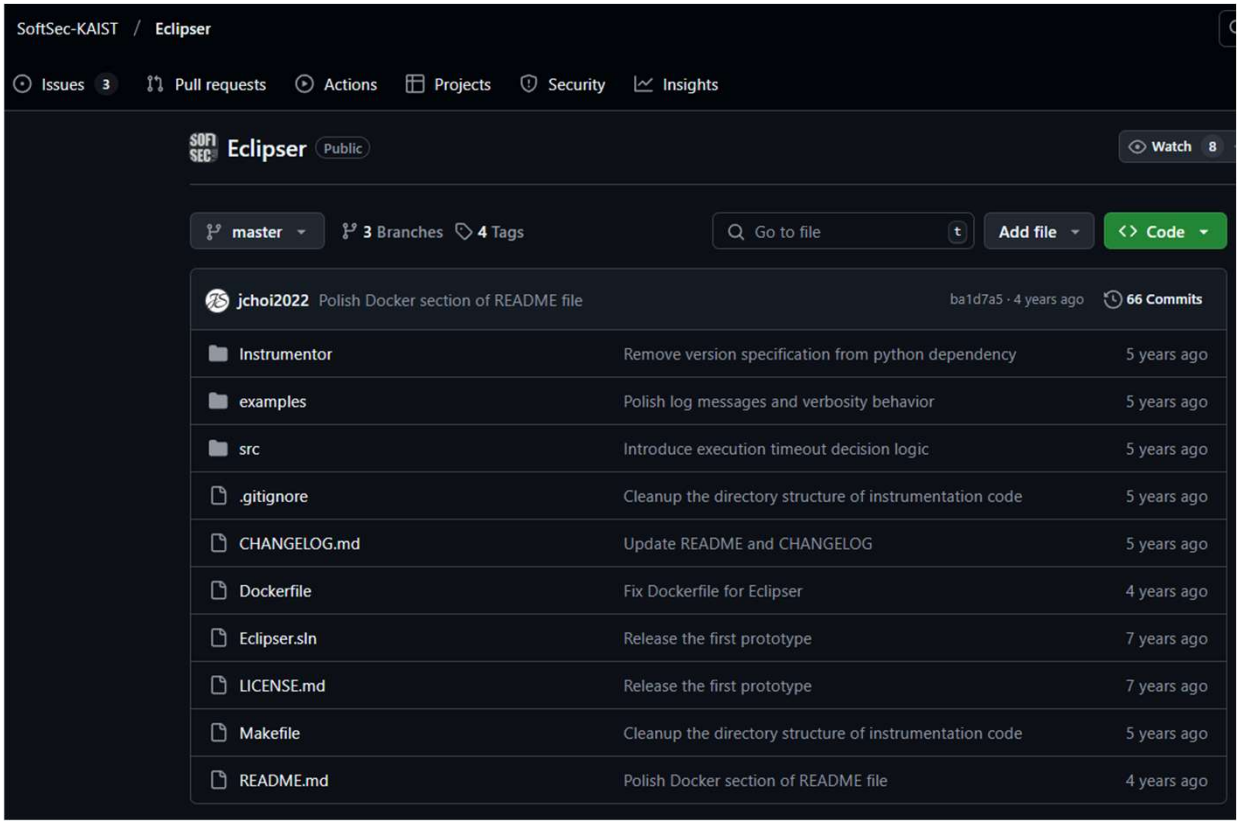
새롭게 제안하는 근사 제약 조건 : approximate path condition

막대한 자원이 요구되지 않는다.

정확도가 떨어지지만, coverage 증대에는 문제가 없다

grey-box concolic testing

Eclipser



grey-box concolic testing

전제 : finite sequence of instructions
반복문은 고려 X
어차피 퍼징하면 input 넣고, 일정 시간 후 중단

p:program
i:input
$\sigma_p(i)$:execution
i[n]:$n^{th}\ byte\ of\ input$
i[n ← v]:i[n]을 v로 update

grey-box concolic testing

p:program

s_p:seed

$\sigma_p(s_p)$:seed execution

s_p[n]:seed의 n번째 바이트

s_p[n].constr: $\sigma_p(s_p)$와 동일한 실행경로가 실행되기 위한 $s_p[n]$의 근사 제약 조건

grey-box concolic testing

seed의 byte단위로 분기 조건을 solve한다

SMT solver 사용하지 않는다

execution behavior만으로 특정 분기를 통과하는 input을 조사한다
(같은 분기를 통과하는 여러 input case 사용)

grey-box concolic testing

SPAWN$(p, s_p, k) \rightarrow execs$

IDENTIFY$(p, execs) \rightarrow conds$

SELECT$(conds) \rightarrow conds'$

SEARCH$(p, k, pc, execs, cond) \rightarrow s'_p, c$

grey-box concolic testing

SPAWN(p, s_p, k) $\rightarrow$ $execs$

s_p[k] 를 변조하여 N_{SPAWN}개의 input set을 생성

grey-box concolic testing

SPAWN($\boldsymbol{p}, \boldsymbol{s_p}, \boldsymbol{k}$) $\rightarrow \boldsymbol{execs}$

s_p[k] 를 변조하여 N_{SPAWN}개의 input set을 생성

IDENTIFY($\boldsymbol{p}, \boldsymbol{execs}$) $\rightarrow \boldsymbol{conds}$

$execs$를 입력으로 한다. k번째 바이트의 영향을 받는 sequence of conditional statements(conds)를 식별

grey-box concolic testing

SPAWN$(\boldsymbol{p}, \boldsymbol{s_p}, \boldsymbol{k}) \rightarrow \boldsymbol{execs}$

s_p[k] 를 변조하여 N_{SPAWN}개의 input set을 생성

IDENTIFY$(\boldsymbol{p}, \boldsymbol{execs}) \rightarrow \boldsymbol{conds}$

$execs$를 입력으로 한다. k번째 바이트의 영향을 받는 sequence of conditional statements(conds)를 식별

SELECT$(\boldsymbol{conds}) \rightarrow \boldsymbol{conds'}$

IDENTIFY에서 찾은 조건문 중 N_{solve}개를 랜덤하게 선택, N_{solve}는 매개변수로 받는다(휴리스틱하게 결정)

grey-box concolic testing

SPAWN$(\boldsymbol{p}, \boldsymbol{s_p}, \boldsymbol{k}) \rightarrow \boldsymbol{execs}$

s_p[k] 를 변조하여 N_{SPAWN}개의 input set을 생성

IDENTIFY$(\boldsymbol{p}, \boldsymbol{execs}) \rightarrow \boldsymbol{conds}$

$execs$를 입력으로 한다. k번째 바이트의 영향을 받는 sequence of conditional statements(conds)를 식별

SELECT$(\boldsymbol{conds}) \rightarrow \boldsymbol{conds'}$

IDENTIFY에서 찾은 조건문 중 N_{solve}개를 랜덤하게 선택, N_{solve}는 매개변수로 받는다(휴리스틱하게 결정)

SEARCH$(\boldsymbol{p}, \boldsymbol{k}, \boldsymbol{pc}, \boldsymbol{execs}, \boldsymbol{cond}) \rightarrow \boldsymbol{s'_p}, \boldsymbol{c}$

선택된 조건분인 cond를 통과하는 seed를 생성

s_p가 cond에서 어떤 분기를 타는지 조사(eax trace), 그와 다른 분기를 타는 s'_p를 찾아 $\sigma_p(s'_p)$시도

s'_p를 찾고, cond에서 $\sigma_p(s_p)$과 같은 분기를 타는 제약조건인 c를 찾아야 종료

pc는 뒤에서 다룸

grey-box concolic testing

Algorithm 1: Grey-box Concolic Testing.

```
1  function GreyConc(p, s_p, k)
2      pc ← {} // Approximate path constraint
3      seeds ← ∅
4      execs ← SPAWN(p, s_p, k)
5      conds ← IDENTIFY(p, execs)
6      for cond in SELECT(conds) do
7          s'_p, c ← SEARCH(p, k, pc, execs, cond)
8          seeds ← seeds + s'_p
9          pc ← pc ∧ c // Merge two constraints
10     return seeds
```

grey-box concolic testing

N_{SPAWN} : 유저 매개변수

휴리스틱

N_{solve}

Algorithm 1: Grey-box Concolic Testing.

```
1  function GreyConc(p, s_p, k)
2      pc ← {} // Approximate path constraint
3      seeds ← ∅
4      execs ← SPAWN(p, s_p, k)
5      conds ← IDENTIFY(p, execs)
6      for cond in SELECT(conds) do
7          s'_p, c ← SEARCH(p, k, pc, execs, cond)
8          seeds ← seeds + s'_p
9          pc ← pc ∧ c // Merge two constraints
10     return seeds
```

grey-box concolic testing

Algorithm 1: Grey-box Concolic Testing.

```
1  function GreyConc(p, s_p, k)
2      pc ← {} // Approximate path constraint
3      seeds ← ∅
4      execs ← SPAWN(p, s_p, k)
5      conds ← IDENTIFY(p, execs)
6      for cond in SELECT(conds) do
7          s'_p, c ← SEARCH(p, k, pc, execs, cond)
8          seeds ← seeds + s'_p
9          pc ← pc ∧ c // Merge two constraints
10     return seeds
```

grey-box concolic testing

Algorithm 1: Grey-box Concolic Testing.

```
1  function GreyConc(p, s_p, k)
2      pc ← {} // Approximate path constraint
3      seeds ← ∅
4      execs ← SPAWN(p, s_p, k)
5      conds ← IDENTIFY(p, execs)
6      for cond in SELECT(conds) do
7          s'_p, c ← SEARCH(p, k, pc, execs, cond)
8          seeds ← seeds + s'_p
9          pc ← pc ∧ c // Merge two constraints
10     return seeds
```

grey-box concolic testing

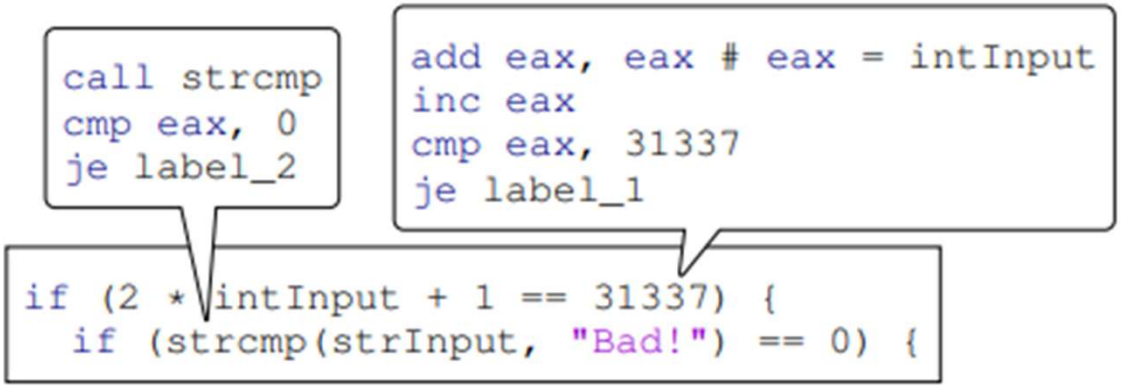
```
call strcmp
cmp eax, 0
je label_2
```

```
add eax, eax # eax = intInput
inc eax
cmp eax, 31337
je label_1
```

```
if (2 * intInput + 1 == 31337) {
  if (strcmp(strInput, "Bad!") == 0) {
```



grey-box concolic testing

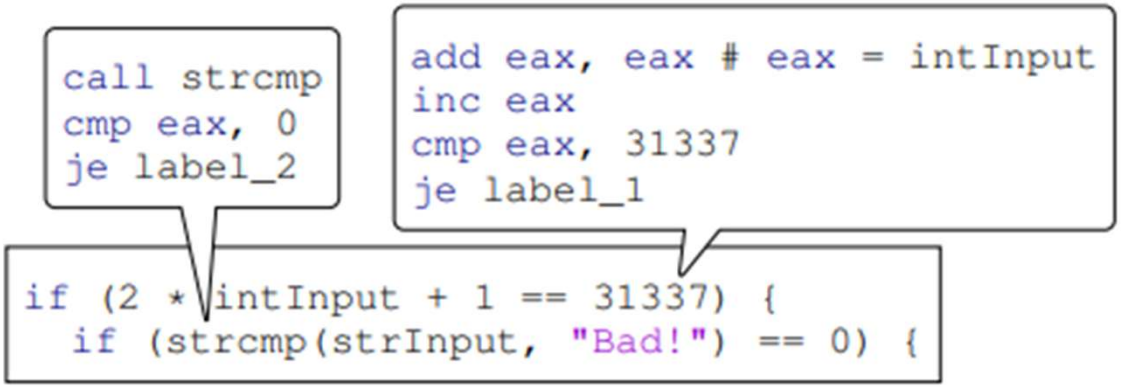


$S_p = \{0,0,0,0,0,0,0,0\}$

$N_{SPAWN} = 3$

$k = 0$

grey-box concolic testing



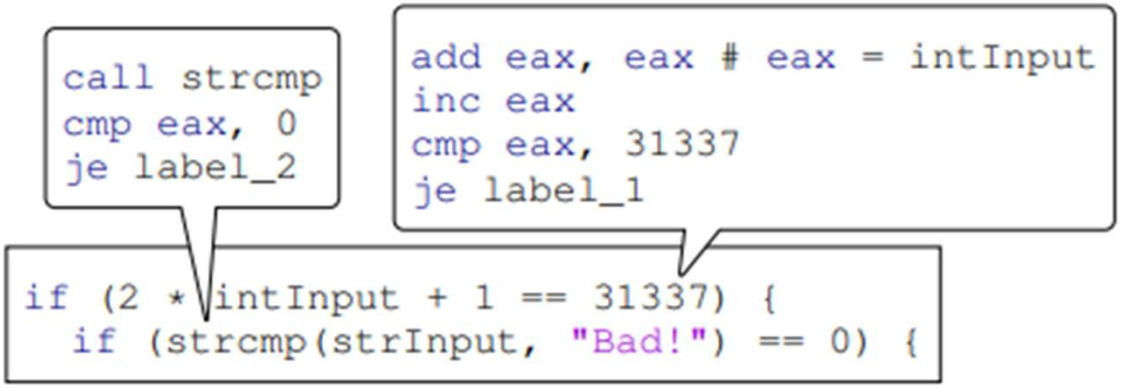
SPAWN

$S_p = \{0,0,0,0,0,0,0,0\}$ ⟶ $S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$N_{SPAWN} = 3$

$k = 0$

grey-box concolic testing



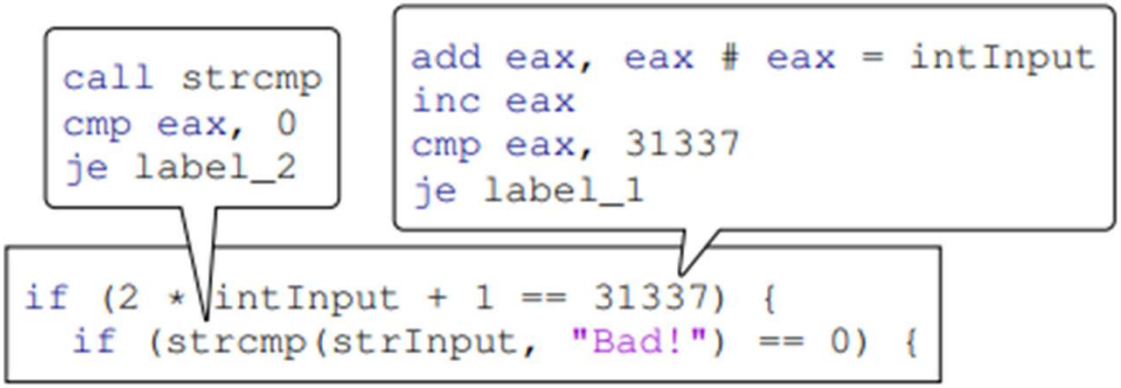
SPAWN

$S_p = \{0,0,0,0,0,0,0,0\}$ → $S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$N_{SPAWN} = 3$ → $\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

$k = 0$

grey-box concolic testing



$S_p = \{0,0,0,0,0,0,0,0\}$ → SPAWN

$N_{SPAWN} = 3$

$k = 0$

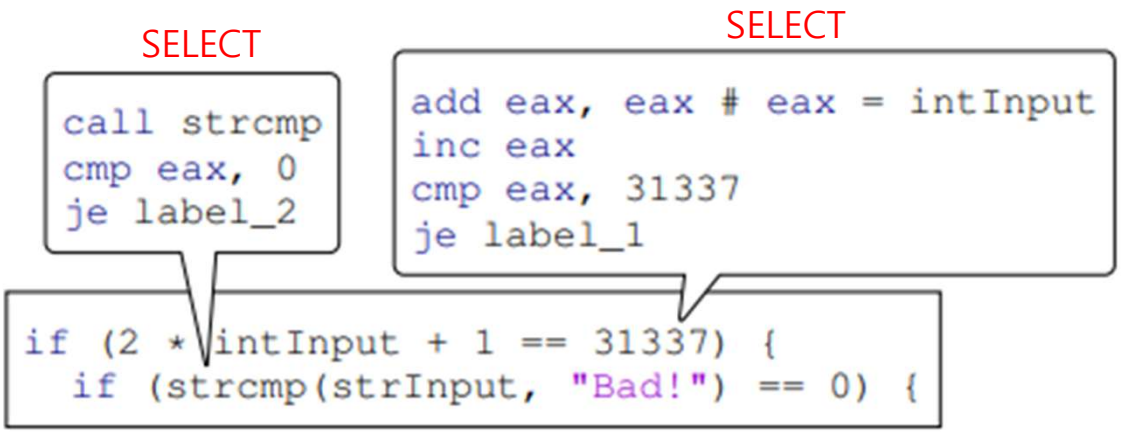
$S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

IDENTIFY

EAX : 21 EAX : 101 EAX : 181

grey-box concolic testing



$S_p = \{0,0,0,0,0,0,0,0\}$

$N_{SPAWN} = 3$

$k = 0$

SPAWN

$S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

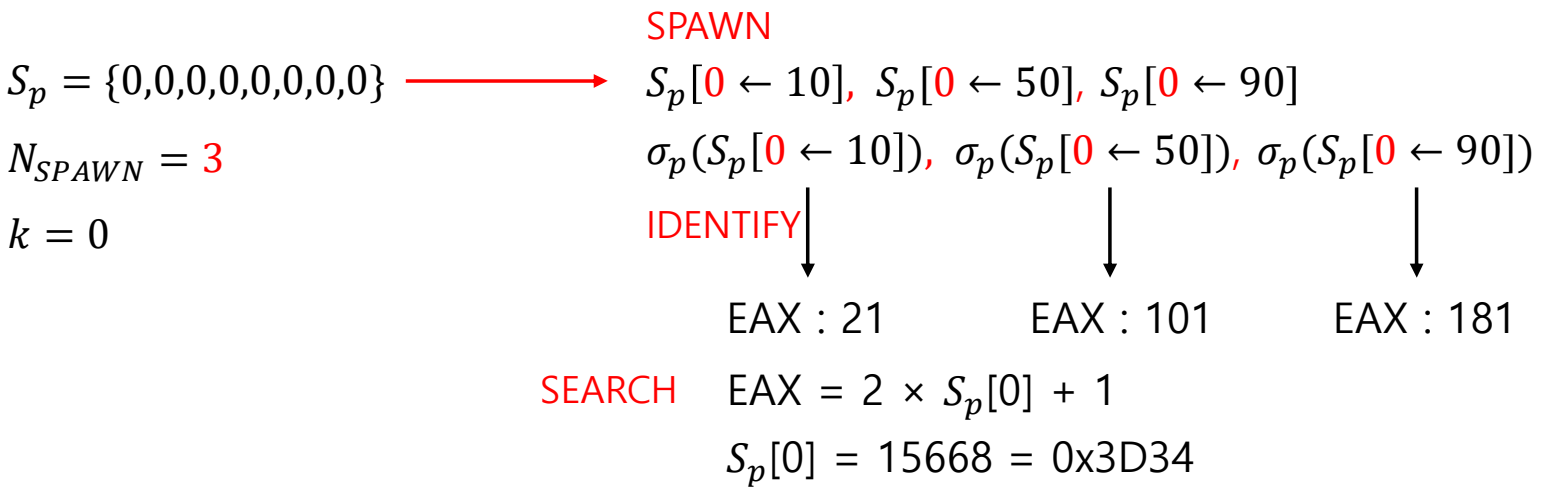
$\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

IDENTIFY

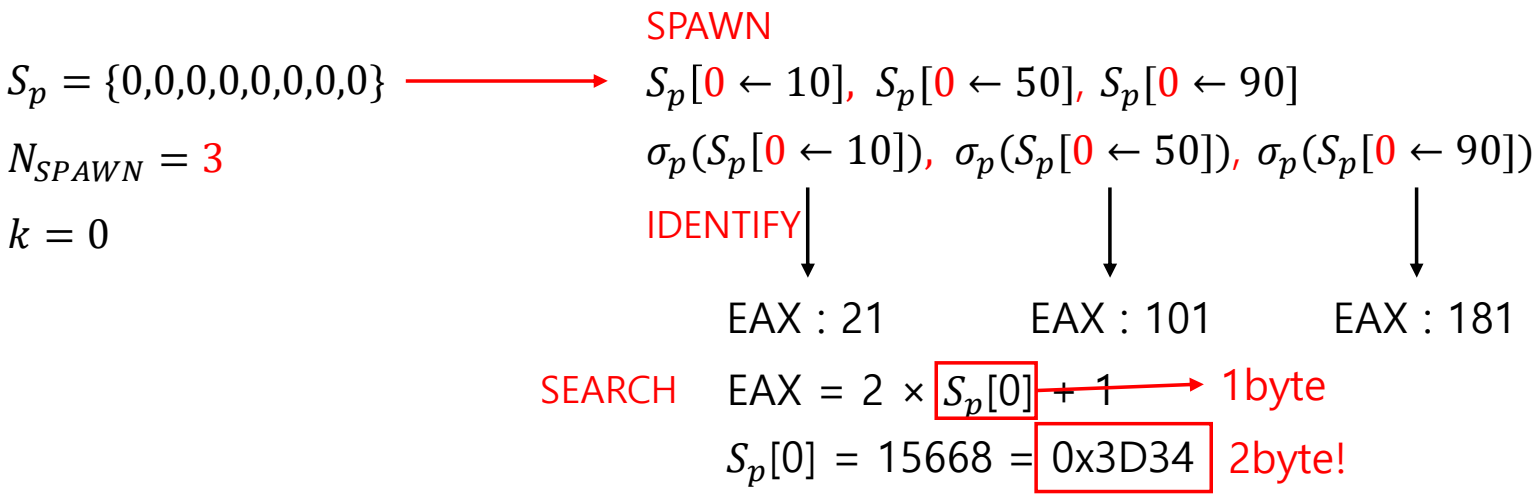
EAX : 21 EAX : 101 EAX : 181

EAX = 2 × $S_p[0]$ + 1

grey-box concolic testing



grey-box concolic testing



grey-box concolic testing

$S_p = \{0,0,0,0,0,0,0,0\}$ → SPAWN

$S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$N_{SPAWN} = 3$

$k = 0$

$\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

IDENTIFY

EAX : 21 EAX : 101 EAX : 181

SEARCH EAX = 2 × $S_p[0]$ + 1

$S_p[0]$ = 15668 = 0x3D34

이런 경우에, 8byte까지는 size up 로직 실행 – 2byte로 재시도

Little endian 고려하여 hex로 34 4d 00 00 00 00 00 00

{$S_p[0]$ → [0x34,0x34], $S_p[1]$ → [0x3D,0x3D]}

grey-box concolic testing

$S_p = \{0,0,0,0,0,0,0,0\}$ → SPAWN $S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$N_{SPAWN} = 3$

$k = 0$

$\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

IDENTIFY

EAX : 21 EAX : 101 EAX : 181

SEARCH EAX = 2 × $S_p[0]$ + 1

$S_p[0]$ = 15668 = 0x3D34

이런 경우에, 8byte까지는 size up 로직 실행 – 2byte로 재시도

Little endian 고려하여 hex로 34 4d 00 00 00 00 00 00

$\{S_p[0] \rightarrow [0x34,0x34], S_p[1] \rightarrow [0x3D,0x3D]\}$

34 34 3D 3D 00 00 00 00 ????!!!

0x3D3D3434 = 1027421236 (실제로 했던 실수…)

grey-box concolic testing

$S_p = \{0,0,0,0,0,0,0,0\}$ → SPAWN

$N_{SPAWN} = 3$

$k = 0$

$S_p[0 \leftarrow 10]$, $S_p[0 \leftarrow 50]$, $S_p[0 \leftarrow 90]$

$\sigma_p(S_p[0 \leftarrow 10])$, $\sigma_p(S_p[0 \leftarrow 50])$, $\sigma_p(S_p[0 \leftarrow 90])$

IDENTIFY

EAX : 21 EAX : 101 EAX : 181

SEARCH EAX = 2 × $S_p[0]$ + 1

$S_p[0]$ = 15668 = 0x3D34

이런 경우에, 8byte까지는 size up 로직 실행 – 2byte로 재시도

Little endian 고려하여 hex로 34 4d 00 00 00 00 00 00

{$S_p[0]$ → [0x34,0x34], $S_p[1]$ → [0x3D,0x3D]}

[0x34,0x34] : 반드시 0x34 이어야 한다

[0x10,0x50] : 0x10 ~ 0x50 사이의 값

grey-box concolic testing

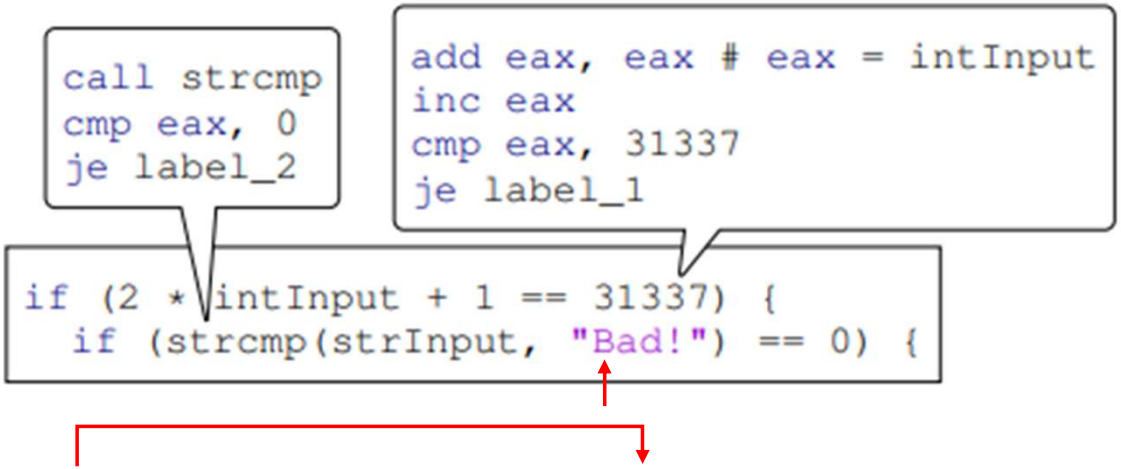
Algorithm 1: Grey-box Concolic Testing.

```
1  function GreyConc(p, s_p, k)
2      pc ← {} // Approximate path constraint
3      seeds ← ∅
4      execs ← SPAWN(p, s_p, k)
5      conds ← IDENTIFY(p, execs)
6      for cond in SELECT(conds) do
7          s'_p, c ← SEARCH(p, k, pc, execs, cond)
8          seeds ← seeds + s'_p
9          pc ← pc ∧ c // Merge two constraints
10     return seeds
```

intInput

{S_p[0] → [0x34,0x34], S_p[1] → [0x3D,0x3D]}

grey-box concolic testing



$S_p = \{0,0,0,0,0,0,0,0\}$ $S'_p = \{0x34,0x3D,0,0,0,0,0,0\}$

$N_{SPAWN} = 3$

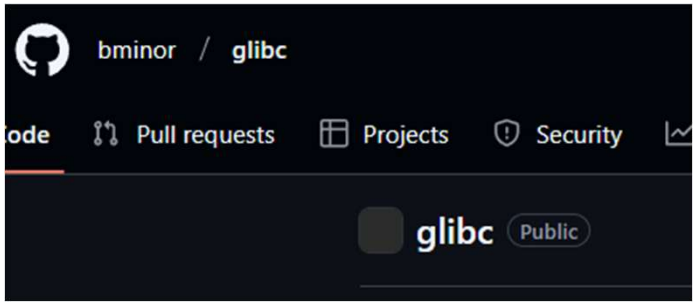
$k = 4$

$\sigma_p(S_p[4 \leftarrow 10])$, $\sigma_p(S_p[4 \leftarrow 50])$, $\sigma_p(S_p[4 \leftarrow 90])$

grey-box concolic testing

strcmp 함수를 반드시 이해해야 한다

```
printf("%d₩n", strcmp(strInput, "Bad!"));
```



glibc / string / strcmp.c

Code | Blame | 125 lines (108 loc) · 3.53 KB

```
90          /* Load the final word of P1 and align the final partial of P2.
91          w1 = *x1++;
92          w2 = MERGE (w2b, sh_1, 0, sh_2);
93        }
94
95      return final_cmp (w1, w2);
96    }
97
98    /* Compare S1 and S2, returning less than, equal to or
99       greater than zero if S1 is lexicographically less than,
100      equal to or greater than S2.  */
101   int
102   strcmp (const char *p1, const char *p2)
103   {
104     /* Handle the unaligned bytes of p1 first.  */
105     uintptr_t n = -(uintptr_t)p1 % sizeof(op_t);
106     for (int i = 0; i < n; ++i)
107       {
108         unsigned char c1 = *p1++;
109         unsigned char c2 = *p2++;
110         int diff = c1 - c2;
111         if (c1 == '\0' || diff != 0)
112           return diff;
113       }
114
115     /* P1 is now aligned to op_t.  P2 may or may not be.  */
116     const op_t *x1 = (const op_t *) p1;
117     op_t w1 = *x1++;
118     uintptr_t ofs = (uintptr_t) p2 % sizeof(op_t);
119     return ofs == 0
120       ? strcmp_aligned_loop (x1, (const op_t *)p2, w1)
121       : strcmp_unaligned_loop (x1, (const op_t *)(p2 - ofs), w1, ofs);
122   }
```

grey-box concolic testing

strcmp 함수를 반드시 이해해야 한다

```
printf("%s %d₩n", "A", strcmp("A", "Bad!"));
printf("%s %d₩n", "B", strcmp("B", "Bad!"));
printf("%s %d₩n", "C", strcmp("C", "Bad!"));
printf("%s %d₩n", "D", strcmp("D", "Bad!"));

printf("%s %d₩n", "B0", strcmp("B0", "Bad!"));
printf("%s %d₩n", "Ba", strcmp("Ba", "Bad!"));
printf("%s %d₩n", "Bb", strcmp("Bb", "Bad!"));
printf("%s %d₩n", "Bc", strcmp("Bc", "Bad!"));
```

Target : Bad!

```
A -1
B -1
C 1
D 1
B0 -1
Ba -1
Bb 1
Bc 1
Input is not equal to 'Bad!'
```

grey-box concolic testing

strcmp 함수를 반드시 이해해야 한다

Target : Bad!

```
A -1
B -97
C 1
D 2
```

```
char* A[2] = { "A",NULL };
char* B[2] = { "B",NULL };
char* C[2] = { "C",NULL };
char* D[2] = { "D",NULL };

printf("%s %d₩n", "A", strcmp(A[0], "Bad!"));
printf("%s %d₩n", "B", strcmp(B[0], "Bad!"));
printf("%s %d₩n", "C", strcmp(C[0], "Bad!"));
printf("%s %d₩n", "D", strcmp(D[0], "Bad!"));
```

strcmp는 두 문자열의 첫 번째 불일치 문자의 ASCII 차이를 반환하는 것이 원칙

grey-box concolic testing

strcmp는 두 문자열의 첫 번째 불일치 문자의 ASCII 차이를 반환하는 것이 원칙

그러나 어셈블리 레벨에서 분석해보면 컴파일러가 일부 코드를 최적화하여 실험을 실패함

```
printf("%s %d₩n", "D", strcmp("D", "Bad!"));
```

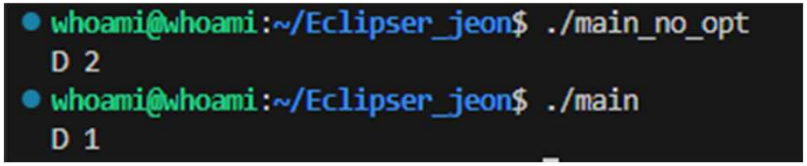
```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     mov    $0x1,%edx
   0x0000000000001156 <+13>:    lea    0xea7(%rip),%rax        # 0x2004
   0x000000000000115d <+20>:    mov    %rax,%rsi
   0x0000000000001160 <+23>:    lea    0xe9f(%rip),%rax        # 0x2006
   0x0000000000001167 <+30>:    mov    %rax,%rdi
   0x000000000000116a <+33>:    mov    $0x0,%eax
   0x000000000000116f <+38>:    call   0x1050 <printf@plt>
   0x0000000000001174 <+43>:    mov    $0x0,%eax
   0x0000000000001179 <+48>:    pop    %rbp
   0x000000000000117a <+49>:    ret
```

최적화 on

```
Dump of assembler code for function main:
   0x0000000000001169 <+0>:     endbr64
   0x000000000000116d <+4>:     push   %rbp
   0x000000000000116e <+5>:     mov    %rsp,%rbp
   0x0000000000001171 <+8>:     lea    0xe8c(%rip),%rax        # 0x2004
   0x0000000000001178 <+15>:    mov    %rax,%rsi
   0x000000000000117b <+18>:    lea    0xe87(%rip),%rax        # 0x2009
   0x0000000000001182 <+25>:    mov    %rax,%rdi
   0x0000000000001185 <+28>:    call   0x1070 <strcmp@plt>
   0x000000000000118a <+33>:    mov    %eax,%edx
   0x000000000000118c <+35>:    lea    0xe76(%rip),%rax        # 0x2009
   0x0000000000001193 <+42>:    mov    %rax,%rsi
   0x0000000000001196 <+45>:    lea    0xe6e(%rip),%rax        # 0x200b
   0x000000000000119d <+52>:    mov    %rax,%rdi
   0x00000000000011a0 <+55>:    mov    $0x0,%eax
   0x00000000000011a5 <+60>:    call   0x1060 <printf@plt>
   0x00000000000011aa <+65>:    mov    $0x0,%eax
   0x00000000000011af <+70>:    pop    %rbp
   0x00000000000011b0 <+71>:    ret
```

최적화 off

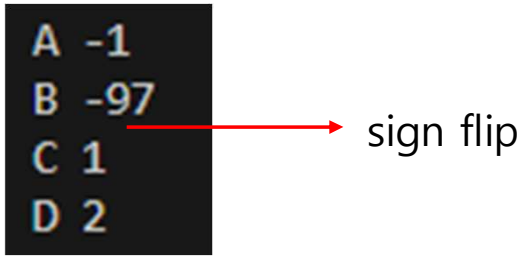
grey-box concolic testing



ㅇ0ㅇ (ctf 아이디어로 쓰면 좋을 듯...)

grey-box concolic testing

$strcmp$ 함수를 반드시 이해해야 한다



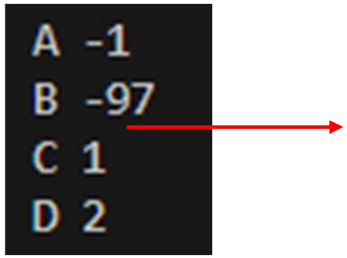
정답에 가까워 질 수록, 0에 가까워진다

$S'_p[4]$ = B~C 라는 것을 알 수 있다

$S'_p[4]$ = [0x42,0x43]

grey-box concolic testing

$strcmp$ 함수를 반드시 이해해야 한다



sign flip

정답에 가까워 질 수록, 0에 가까워진다

$S'_p[4]$ = B~C 라는 것을 알 수 있다

$S'_p[4]$ = [0x42,0x43]

그러나 rax == 0 인 condition은 찾을 수 없다. -> extend input field

[0x4200,0x4300]

이러한 과정을 반복하면 Bad!를 찾게 된다

grey-box concolic testing

SPAWN

Objective : S_p의 $S_p[k]$를 조작하여 비슷한 실행경로를 가지는 input을 생성하는 것

How?

$S_p[k].constr$ 을 기반으로 변이하여 $\sigma_p(i_1) \approx \sigma_p(i_2) \approx \cdots \approx \sigma_p(i_3)$ 인 $\{i_1, i_2, i_3\}$를 구한다
SMT 등의 solver를 사용하면 가능하지만, approximate path constraint로 overhead 적게 가능하다

ex) c가 [0x10,0x50]이라면,
기존에는 Solver를 통해 $S_p[k]$를 정확하게 0x10~0x50으로 계산하는 대신에 많은 시간과 자원이 요구되었다
Eclipser는 0x10~0x50 범위 내에서 랜덤하게 값을 고른다

grey-box concolic testing

SPAWN

Objective : S_p의 $S_p[k]$를 조작하여 비슷한 실행경로를 가지는 input을 생성하는 것

$\sigma_p(i_1) \approx \sigma_p(i_2) \approx \cdots \approx \sigma_p(i_3)$ 테스트 결과
새로운 경로가 발견되지 않았다면, IDENTIFY에서 어떤 분기문이 k에 영향을 받는지 분석

새로운 경로가 발견되었다면, 새로운 분기 탐색에 성공한 것으로 판단
해당 경로의 제약을 SEARCH 에서 분석

grey-box concolic testing

IDENTIFY

Objective : offset k와 $\sigma_p(S_p)$ 의 관계를 조사

How?

fine-grained taint analysis를 사용해도 되지만, $^%#^문제와 계산적 비용이 많이 들어간다

* fine-grained taint analysis가 무엇인지 (시간이 된다면)뒤에서 다룰 예정

grey-box concolic testing

SELECT

Objective : IDENTIFY 동안에 SEARCH에서 처리할 분기문을 결정

How?

Path explosion을 방지하기 위함

N_{solve}개를 랜덤하게 뽑는다.(SAGE, KLEE 도 마찬가지)

N_{solve}는 휴리스틱하게 결정되지만, 논문에서는 200으로 결정되었다.

```
| NSolve _ -> "Number of branches to flip in grey-box concolic testing " +
              "when an execution path is given. 'N_solve' parameter in " +
              "the paper."
| NSpawn _ -> "Number of byte values to initially spawn in grey-box " +
              "concolic testing. 'N_spawn' parameter in the paper."
```

```
let rec selectAndRepair opt branchTree =
  let selectN = opt.NSolve
  let size = sizeOf branchTree
  if selectN > size then reverse branchTree
  else
    let selectSet = randomSubset size selectN
    filterAndReverse selectSet branchTree
```

Ref : https://github.com/SoftSec-KAIST/Eclipser/blob/ba1d7a55c168f7c19ecceb788a81ea07c2625e45/src/GreyConcolic/BranchTree.fs#L311

grey-box concolic testing

SEARCH

Objective : cond(선택된 분기문)에 조건을 제약으로 표현해야함(resolve라고 표현)

How?

SMT solver를 사용하지 않기에, interval로 approximate한 제약을 찾는다

1. 현재 분기 조건을 공식화하고 solve한다 - Solving Branch Condition
2. 그에 대응하는 입력 필드를 인식한다 - Recognizing Input Field
3. 조건문을 통과할 수 있는 새로운 시드를 생성한다 - Seed Generation

grey-box concolic testing

SEARCH

Objective : cond(선택된 분기문)에 조건을 제약으로 표현해야함(resolve라고 표현)

How?

SMT solver를 사용하지 않기에, interval로 approximate한 제약을 찾는다

1. 현재 분기 조건을 공식화하고 solve한다 - **Solving Branch Condition**
2. 그에 대응하는 입력 필드를 인식한다 - Recognizing Input Field
3. 조건문을 통과할 수 있는 새로운 시드를 생성한다 - Seed Generation

grey-box concolic testing

Solving Branch Condition

Linear 한 경우

$$\frac{oprnd(i_1)-oprnd(i_2)}{i_1-i_2} = \frac{oprnd(i_2)-oprnd(i_3)}{i_2-i_3}$$

방정식 풀어서 관계를 정리할 수 있다

Monotonic(단조 증가) 한 경우

```
1  # mov ebx, f(input)
2  cmp ebx, 20
3  je label
```

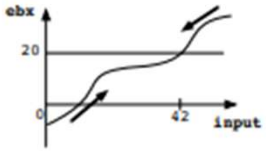


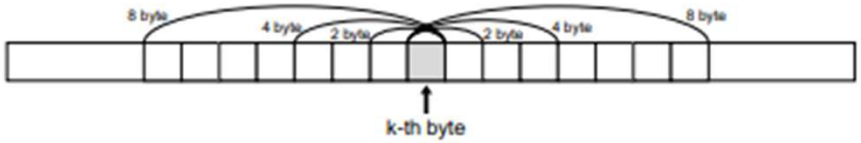
Fig. 4. Monotonic input-output relationship.

ebx와 input이 monotonic 관계에 있다

구간에 따른 표현으로, binary search로 solve할 수 있다

grey-box concolic testing

Recognizing Input Field



$s_p[k]$의 size를 결정할 때,

Size를 1에서 시작하여 8을 threshold로 증가하며 해를 찾는다

grey-box concolic testing

Seed Generation

새로운 seed를 생성하는 과정에서, 제약조건 c들을 만족시켜야 한다.

따라서 누적된 제약 조건인 pc를 구현하여 사용한다.

$S_p[i].constr = pc \wedge c$

Eclipser

grey-box concolic testing

Linear, Monotonic 관계에서 분기를 잘 분석해낸다
그러나, 그 외 복잡한 분기문은 해결하지 못한다

이를 해결하기 위해 Grey-box Fuzzing과 병행한다(번갈아 실행)

Grey-box Fuzzing은 전통적인 퍼징 기법으로, 무작위 입력 변이를 통해 실행 경로를 탐색.
Grey-box Concolic Testing은 제안한 방식으로 실행 경로를 분석하고, 논리적으로 새로운 입력을 생성.

Eclipser

grey-box concolic testing

Algorithm 2: Main Algorithm of Eclipser.

```
   // p: PUT, seeds: initial seeds, t: time limit
1  function Eclipser(p, seeds, t)
2      Q ← InitQueue(seeds)
3      T ← ∅
4      while getTime() < t do
5          R_G, R_R ← Schedule()
6          Q, T ← GreyConcolicLoop(p, Q, T, R_G)
7          Q, T ← RandomFuzzLoop(p, Q, T, R_R)
8      return T
```

입력값

p: 테스트할 프로그램

t: 실행 시간 제한(Time Limit)

seeds: 초기 테스트 입력(Seed Set)

출력값

T: 퍼징 중에 생성된 테스트 케이스(Test Case Set)

Eclipser

grey-box concolic testing

Algorithm 2: Main Algorithm of Eclipser.

```
   // p: PUT, seeds: initial seeds, t: time limit
1  function Eclipser(p, seeds, t)
2      Q ← InitQueue(seeds)
3      T ← ∅
4      while getTime() < t do
5          R_G, R_R ← Schedule()
6          Q, T ← GreyConcolicLoop(p, Q, T, R_G)
7          Q, T ← RandomFuzzLoop(p, Q, T, R_R)
8      return T
```

1. seed를 Q에 삽입
2. t시간까지 while 루프
3. Schedule 을 통해 리소스 할당

   Grey-box Concolic Testing : R_G

   Grey-box Fuzzing : R_R
4. GreyConcolicLoop와 RandomFuzzLoop 전략을 번갈아 사용, 결과를 T에 저장

Eclipser

Q는 seed의 우선순위를 평가하여 저장된다

Code coverage 기반으로 seed를 평가 – coverage 증가(새로운 path)하는 seed에 높은 우선순위
- coverage 증가가 없는 경우, drop(제거)

오프셋 k 를 조정하여 입력을 다양하게 변이
k 번째 바이트 이외에 k-1, k+1 번째 바이트도 큐에 추가
k 값도 동적으로 조정

$\mathrm{R_G}, \mathrm{R}_R$사이에 seed 공유

Eclipser

Resource Scheduling

새로운 path(커버리지의 증가)를 지향

현재 전략이 허용된 실행 횟수를 초과하면 다른 전략으로 전환된다(switching)

효율성f를 계산하여 리소스를 동적으로 할당한다

$f = \frac{N_{path}}{N_{exec}}$ 이며, 실행 대비 발견한 경로의 비율이다

Eclipser

Approximate Path Constraint

경로 제약(Path Constraints)을 정밀한 논리식이 아닌 "구간(Interval)"로 근사하여 표현

$[l, u]$ $(interval\ constrain: l \leq x \leq u)$

l:하한값

u:상한값

이를 통해 입력 값이 구간 내에 존재하도록 제한

교집합을 통해 제약 조건을 계산

$[l_1, u_1] \wedge [l_1, u_1] = [\max(l_1, l_2), \min(u_1, u_2)]$

*근사를 활용하는 자세한 로직은 몇개 더 있음

Eclipser

Implementation

F#과 C로 구현

QUMU binary instrumentation을 위해 c를 사용

F#으로 메인 알고리즘 구현

Grey-Box Fuzzing을 위해 AFL mutation module(verified!)사용

Greedy Set-Cover 알고리즘으로 seed 최적화

QEMU 기반의 execution feedback

Evaluation

Eclipser의 설정 값(Configuration Parameter)이 성능에 미치는 영향?

N_{SPAWN}, N_{EXEC}, t는 휴리스틱하게 결정된다

N_{SPAWN} 이 너무 작으면 실행 경로 탐색이 부족하고, 너무 크면 탐색 속도가 저하될 수 있음.

N_{EXEC} 이 높을수록 실행 속도가 빨라지지만, 불필요한 반복 실행이 많아질 가능성 있음.

*이게 왜 Evaluation인지..?

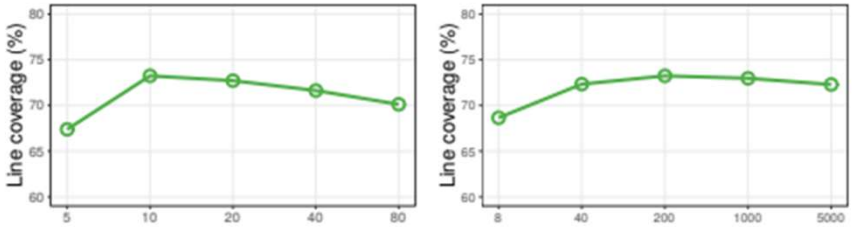
Evaluation

Grey-box Concolic Testing이 일반적인 테스트 케이스 생성 알고리즘이 될 수 있는가?
White-box 퍼징과 비교하면 어떤가?

평가 기준
커버리지(Coverage): 각 퍼저가 탐색한 코드 영역의 크기 비교.
버그 탐색 능력: 실제 취약점(버그) 탐색 성능 비교.
속도: 실행 횟수 대비 탐색된 분기(branch) 개수 평가.

White-box: klee
Greybox-concolic: Eclipser
Greybox: Eclipser

Evaluation



(a) N_{spawn} (in log scale) (b) N_{solve} (in log scale)

Fig. 6. The impact of N_{spawn} and N_{solve}.

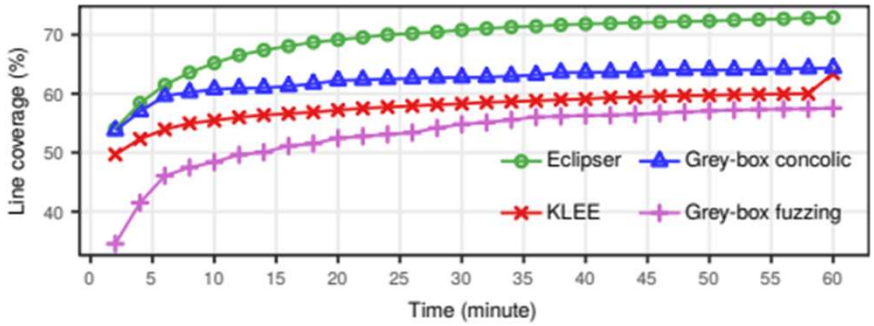


Fig. 7. Line coverage achieved by Eclipser and KLEE over time for coreutils.

Evaluation

테스트 대상: GNU coreutils 8.27 (총 107개 프로그램)

제외된 프로그램:
시스템에 영향을 미칠 수 있는 프로그램 (kill, rm 등 8개)

총 95개의 프로그램을 테스트
실행 시간: 각 프로그램당 1시간 테스트
KLEE 실행 시 공식적인 명령어 옵션 사용 (공정한 비교를 위해 KLEE 공식 웹사이트의 설정을 따름).
Eclipser 실행 시 입력 크기 제한을 동일하게 설정하여 비교 실험

Evaluation

퍼징 방식	탐색된 코드 라인 수	탐색률(%)
Grey-box Concolic Testing (Eclipser)	20,737	64.36%
White-box Fuzzing (KLEE)	20,445	63.45%
Grey-box Fuzzing 단독	18,540	57.54%

Grey-box Concolic Testing 단독으로도 KLEE와 경쟁 가능하지만,

Grey-box Fuzzing을 함께 사용하면 추가적인 코드 커버리지 증가 가능