

論文紹介

HTFuzz : Heap Operation Sequence
Sensitive Fuzzing
[ASE'22]

HTFuzz [3] の概要

ジャンル： typestate ガイドファジング

問題提起：

- UAF のような特定の順序で発生する脆弱性はコードカバレッジだけでは効率的に発見できない
- 従来の typestate ファザーは静的解析・事前情報に依存

提案手法：

- 実行時にメモリアクセスを追跡し、ヒープ操作シーケンスの多様性を高める
- 実行時にアクセスされるポインタの数を計測

結果：

- 従来のファザーより多くのヒープ操作シーケンスを発見
- 37件の新たな脆弱性を発見（うち 32件はヒープの時間的脆弱性）

目次

1. AFL [1] の概要
2. カバレッジガイドファザーの問題点
3. UAFL [2] の概要
4. HTFuzz [3] の概要



[1] <https://github.com/google/AFL>

[2] Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [ICSE '20]

[3] HTFuzz: Heap Operation Sequence Sensitive Fuzzing [ASE '22]

目次

1. AFL [1] の概要
2. カバレッジガイドファザーの問題点
3. UAFL [2] の概要
4. HTFuzz [3] の概要



[1] <https://github.com/google/AFL>

[2] Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [ICSE '20]

[3] HTFuzz: Heap Operation Sequence Sensitive Fuzzing [ASE '22]

ファジング (fuzzing)

ChatGPT に尋ねた結果：

ファジングとは、ソフトウェアやシステムの脆弱性を発見するために、ランダムまたは意図的に不正なデータや入力を生成してテストする手法です。

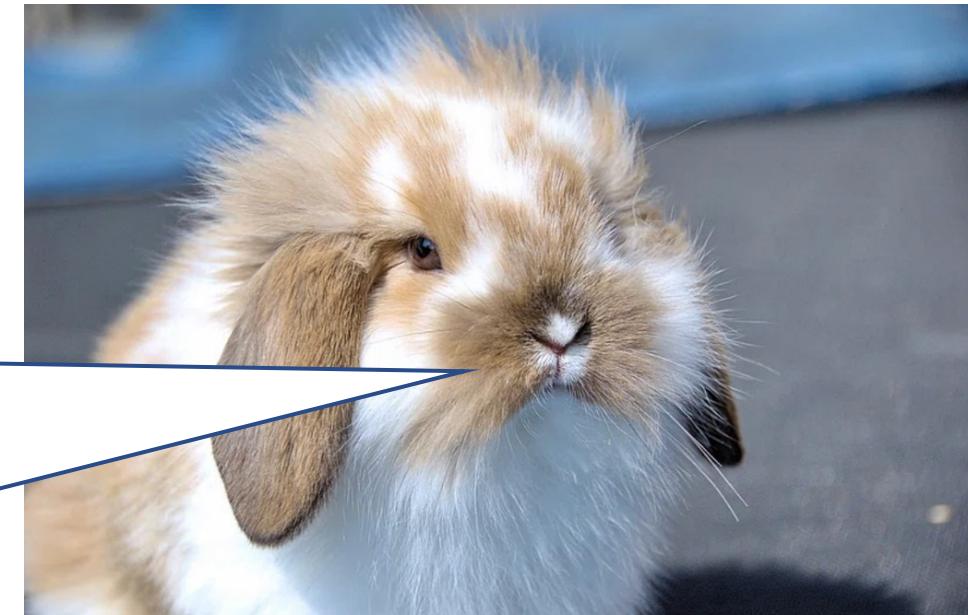
ファジング研究のゴール（の一つ）：

いかに効率的に脆弱性を発見できる入力を生成できるファザーを作れるか？

AFL (American Fuzzy Lop) [1]

- Google（の中の人）が開発したカバレッジガイドファザーのひとつ
- 新しい CFG エッジを通るテストケースを興味深いものとして保存
- テストケースを繰り返し変異させ、脆弱性を検出する

For many years after its release, AFL has been considered a "state of the art" fuzzer. AFL is considered "a de-facto standard for fuzzing", and the release of AFL contributed significantly to the development of fuzzing as a research area. AFL is widely used in academia; academic fuzzers are often forks of AFL, and AFL is commonly used as a baseline to evaluate new techniques. [6]



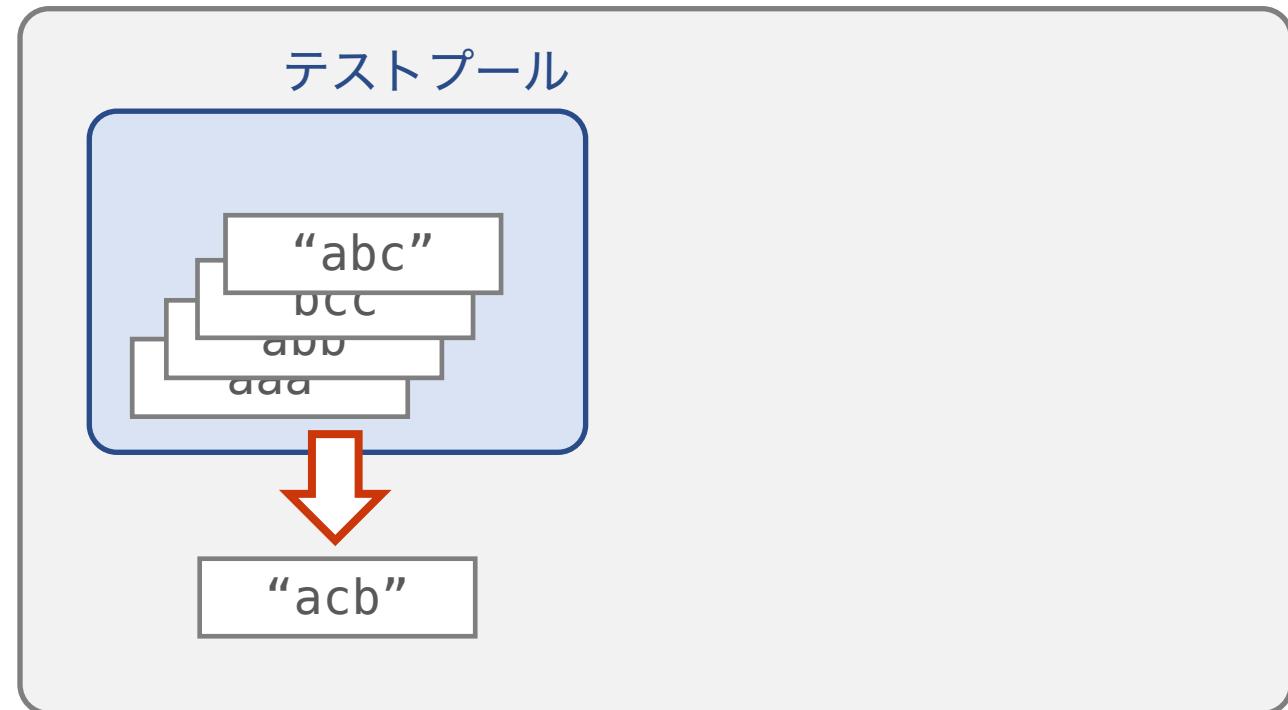
[1] <https://github.com/google/AFL>

[6] [https://en.wikipedia.org/wiki/American_Fuzzy_Lop_\(software\)](https://en.wikipedia.org/wiki/American_Fuzzy_Lop_(software))

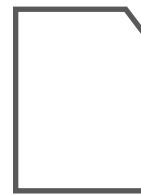
AFL の流れ

1. テストプールからシードを選択

AFL

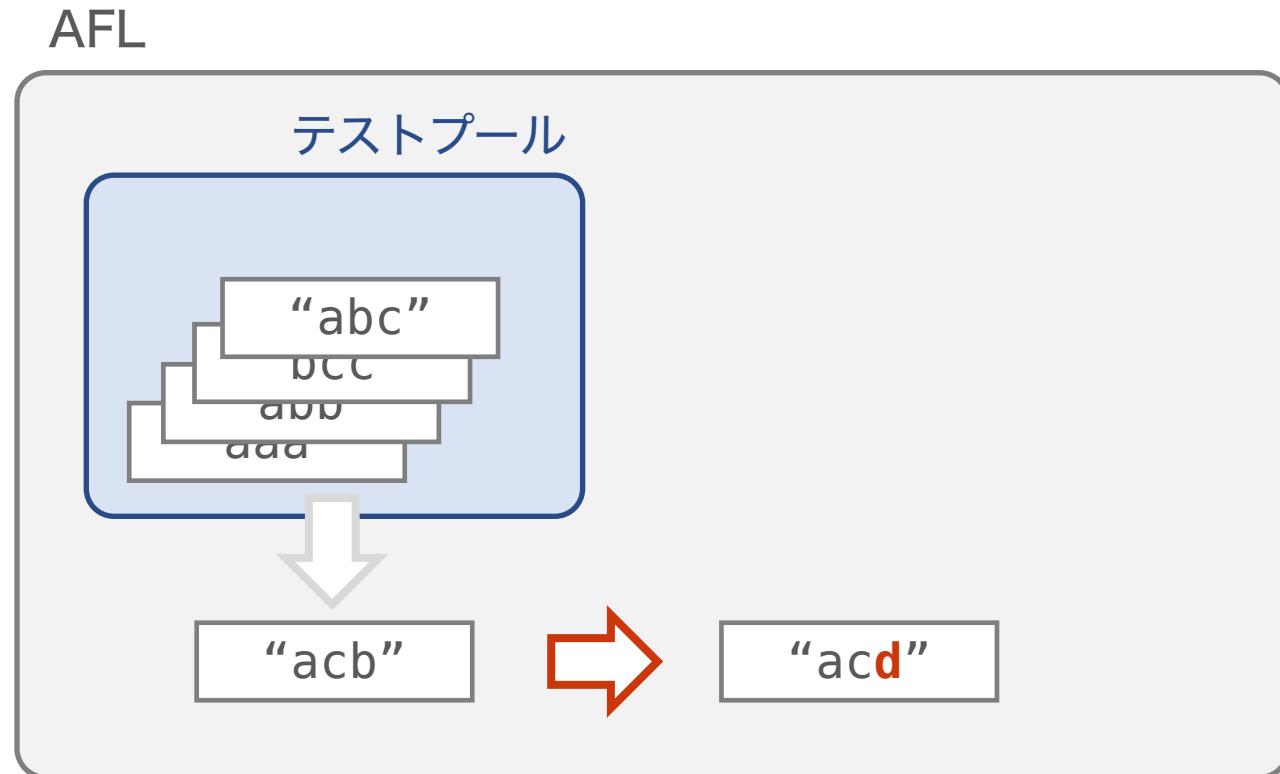


対象プログラム



AFL の流れ

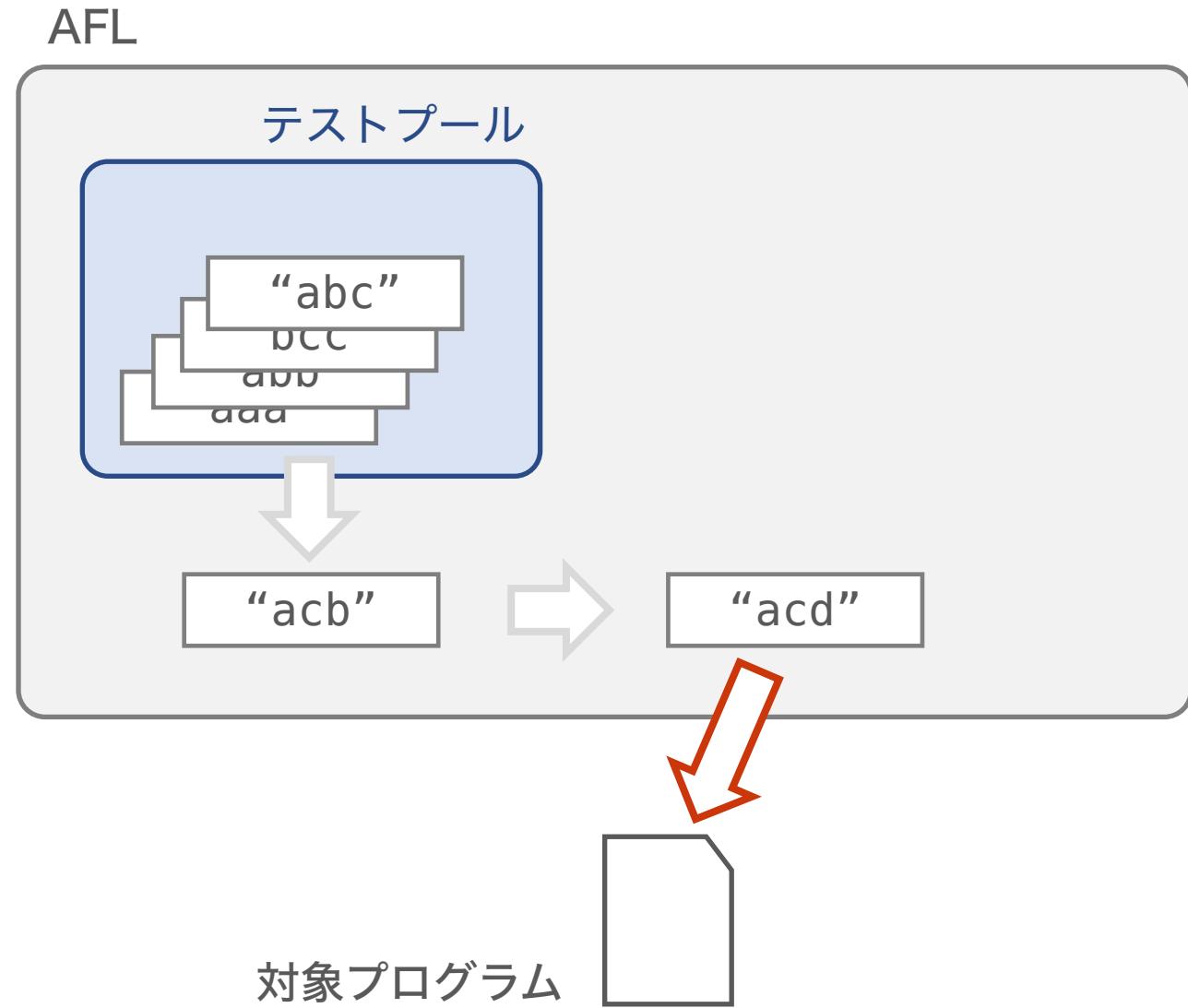
1. テストプールからシードを選択
2. シードを変異



対象プログラム

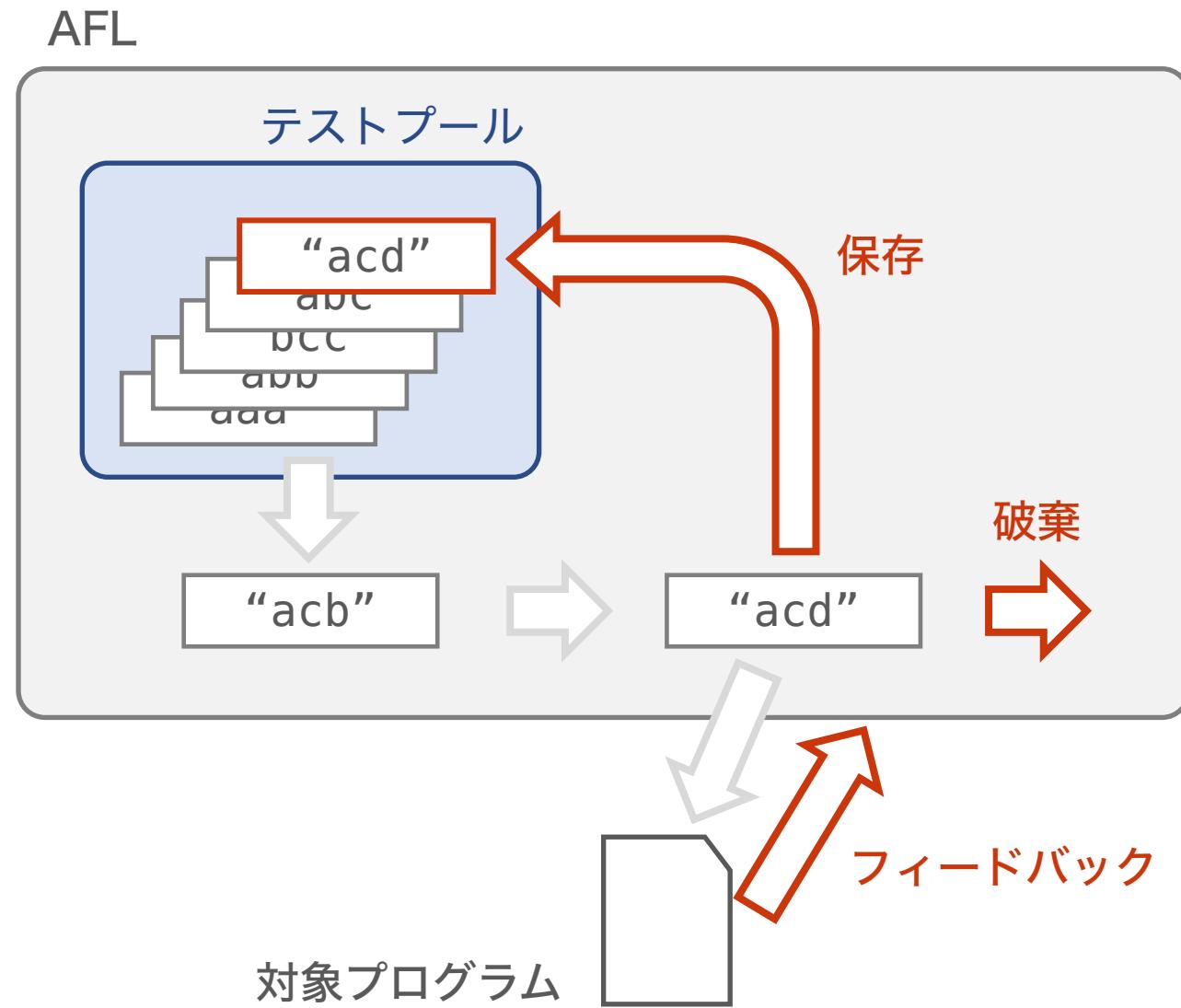
AFL の流れ

1. テストプールからシードを選択
2. シードを変異
3. 変異させたテストケースで
対象プログラムを実行



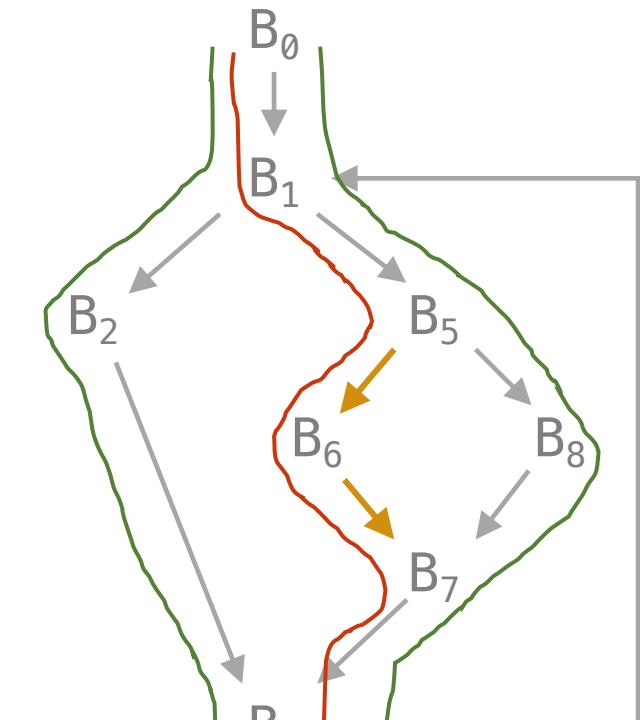
AFL の流れ

1. テストプールからシードを選択
2. シードを変異
3. 変異させたテストケースで
対象プログラムを実行
4. 実行から得られた
フィードバックから,
変異させたテストケースを
残すか破棄するかを決定



AFL のフィードバック

- CFG エッジのカバレッジを測定
→ 網羅率の高いテストは良いテスト
- それまでのテストケースで通っていない CFG エッジを通ったテストケースを興味深いものとして保存する
- より正確には、CFG エッジを新しい回数通ったテストケースを保存する
(1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128- 回で分割)



→ 通ったことのない CFG エッジ

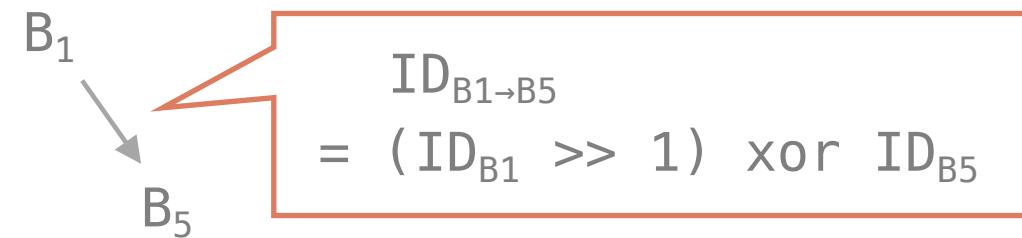
AFL のフィードバック

- 共有メモリ `shared_mem` を用いて CFG エッジを通ったか測定
- ランダムに基本ブロックに ID を割り当てる
- CFG エッジの ID は以下のように計算

$$\text{ID}_{A \rightarrow B} = (\text{ID}_A \gg 1) \text{ xor } \text{ID}_B$$

→ ID_A を右シフトするのは $\text{ID}_{A \rightarrow B}$ と $\text{ID}_{B \rightarrow A}$ を区別するため
- CFG エッジ $A \rightarrow B$ を通ったとき

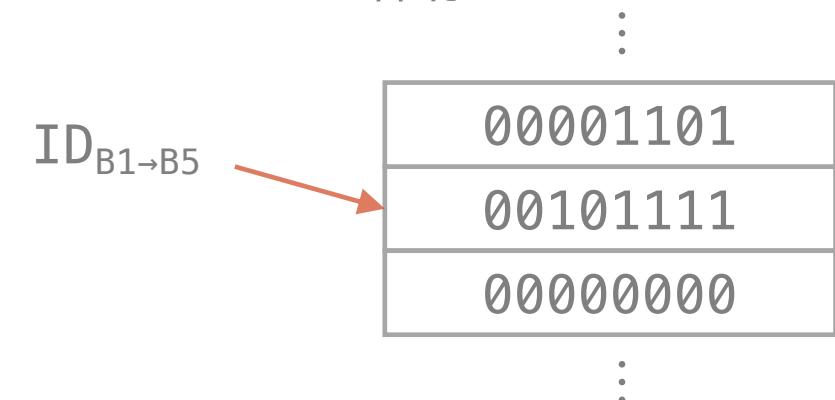
$$\text{shared_mem}[\text{ID}_{A \rightarrow B}]++$$



- 各実行で通過したとき

$$\text{shared_mem}[\text{ID}_{B1 \rightarrow B5}]++$$

- ファジングテスト全体でカバレッジを保存



AFL のフィードバック

- ランダムに基本ブロックに ID を割り当てる
- CFG エッジの ID は以下のように計算

$$ID_{A \rightarrow B} = (ID_A \gg 1) \text{ xor } ID_B$$

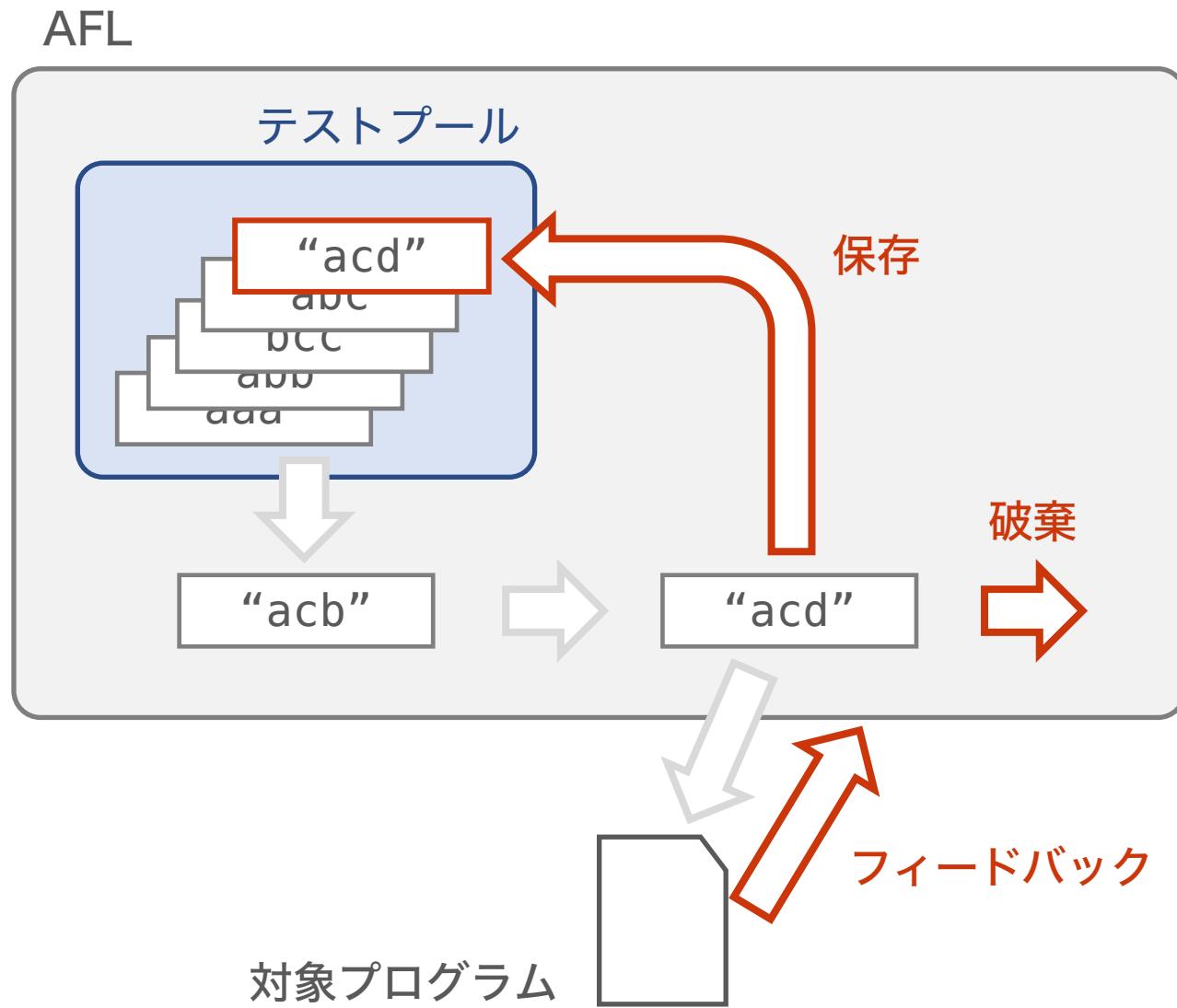
- ID は衝突しないのか？
 - CFG エッジの数が多くなると
衝突する可能性は高くなる
 - AFL は ID の衝突を無視する
- shared_mem[] は 64KB
 - 精度とオーバーヘッドのトレードオフ

Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	-

分岐の数と ID の衝突確率 [4]

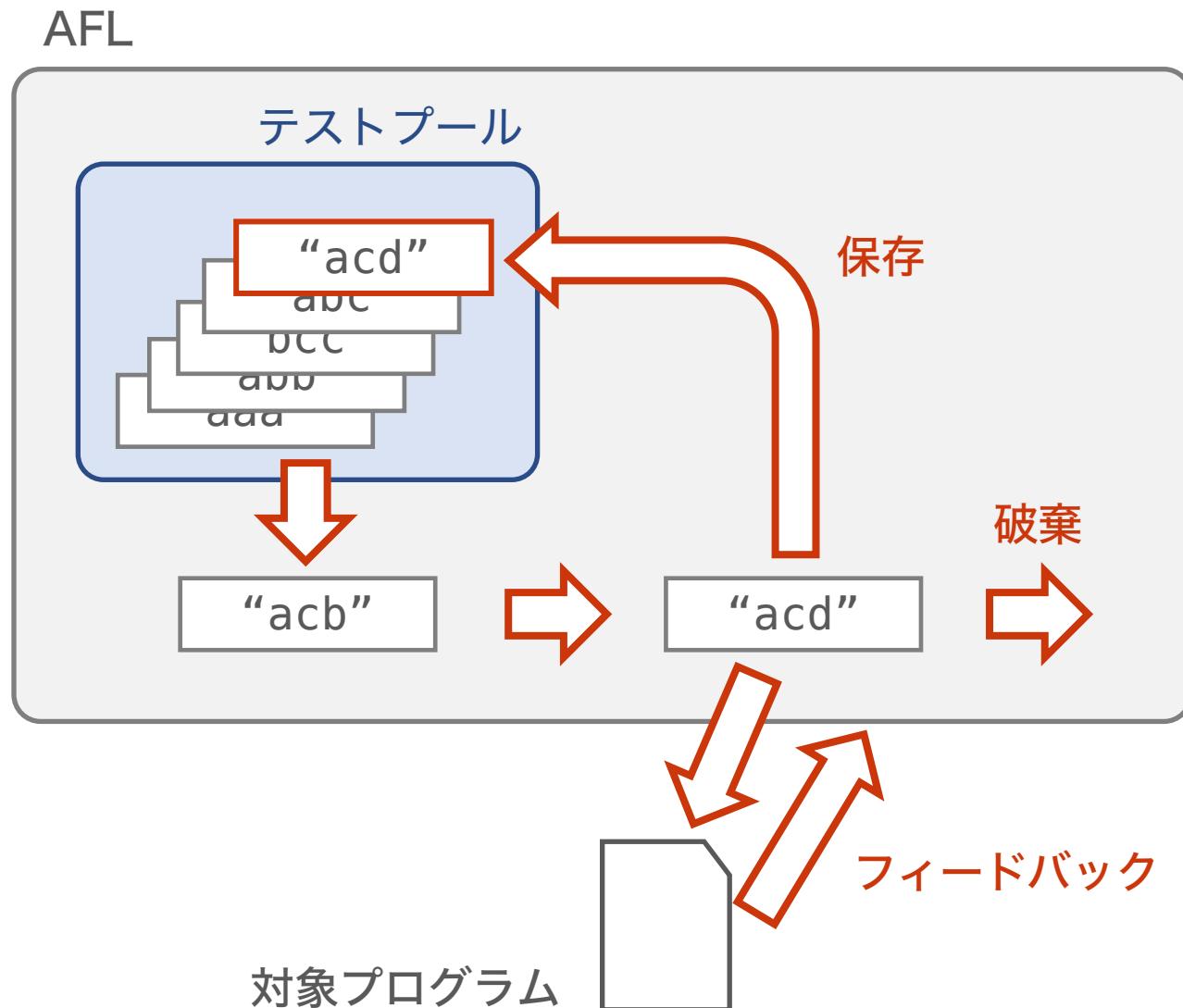
AFL の流れ

1. テストプールからシードを選択
2. シードを変異
3. 変異させたテストケースで
対象プログラムを実行
4. 実行から得られた
フィードバックから,
変異させたテストケースを
残すか破棄するかを決定



AFL の流れ

1. テストプールからシードを選択
2. シードを変異
3. 変異させたテストケースで
対象プログラムを実行
4. 実行から得られた
フィードバックから,
変異させたテストケースを
残すか破棄するかを決定
5. 1~4 を繰り返す



効果的なファザーを作るには

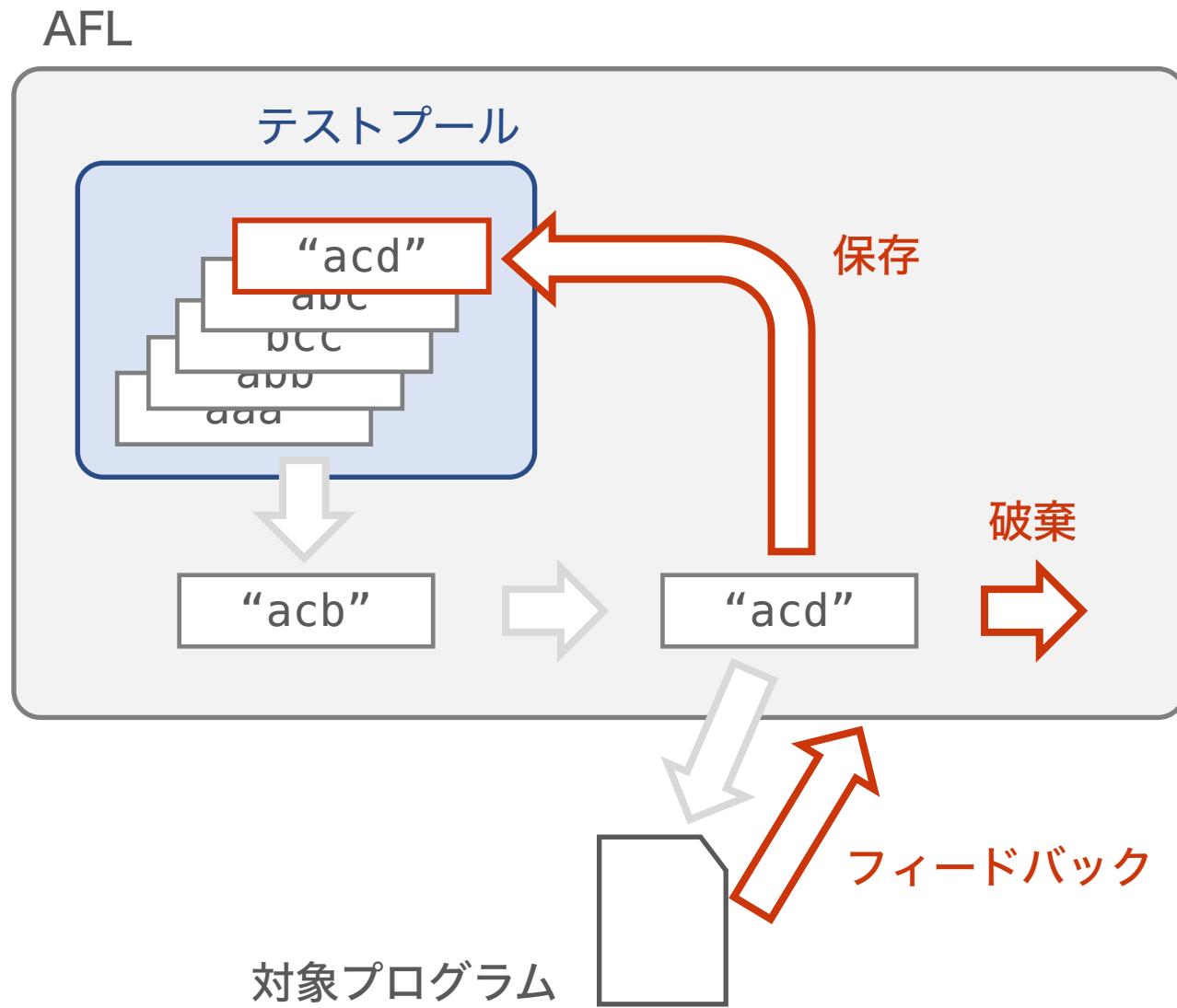
効率的に脆弱性を発見するファザーを作るために考えるべき戦略

- ・シードの **保存** 戰略
- ・シードの **選択** 戰略
- ・シードの **変異** 戰略

(その他にも、テストケースのサイズ削減、実行の方式など、
無駄を削減する方法はある)

シードの保存戦略

- ・どのテストケースをテストプールに残すか?
- AFL は新しい CFG エッジを通ったテストケースを興味深いものとして保存する



シードの選択戦略

- ・テストプール内のどのテストケースをシードとするか?
→ AFL はファイルサイズが小さく、実行時間が早いテストケースを優先的に選択する

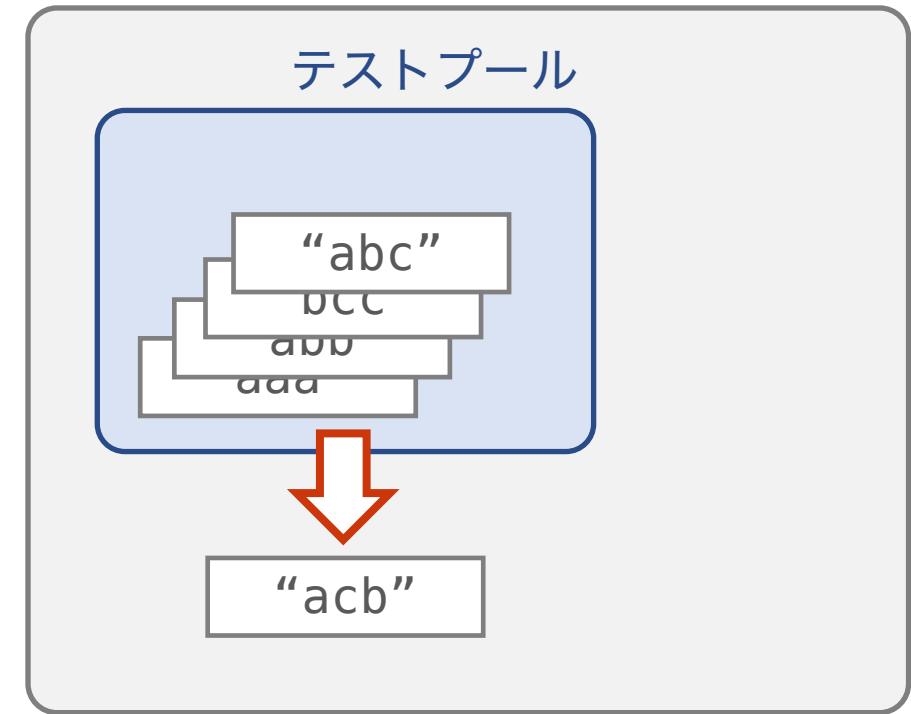
Win!
seedA

サイズ : 1MB
実行時間 : 2ms

seedB

サイズ : 2MB
実行時間 : 5ms

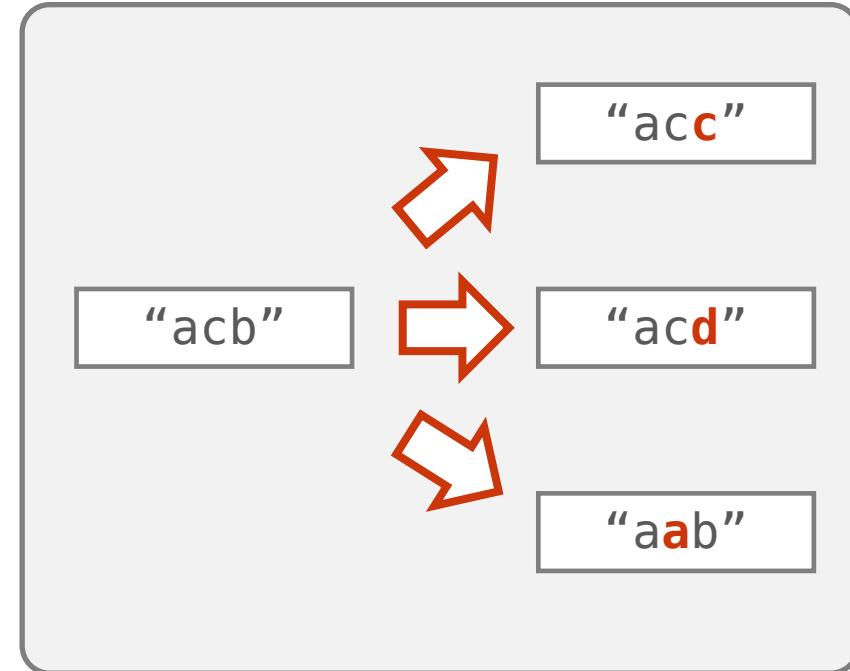
AFL



シードの変異戦略

- ・ シードを何回変異させるか？ (power scheduling)
 → AFL はファイルサイズが小さく、
 実行時間が早いシードを多く変異させる
- ・ シードのどのバイトを変異させるか？
 → AFL は各バイトを反転させて、実行パスが
 変化するかを観察
 → 実行パスが変化したバイトのみを変異させる
 (effector map : 興味深いバイトを記録)
- ・ シードをどのように変異させるか？ (バイト反転, 加減算, など)
 → AFL はランダム

AFL



目次

1. AFL [1] の概要
2. カバレッジガイドファザーの問題点
3. UAFL [2] の概要
4. HTFuzz [3] の概要



[1] <https://github.com/google/AFL>

[2] Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [ICSE '20]

[3] HTFuzz: Heap Operation Sequence Sensitive Fuzzing [ASE '22]

カバレッジガイドファザーの問題点

- ・ CFG エッジのカバレッジガイドファザーは単純に CFG エッジを通ったかどうかだけを記録する
- ・ 特定の順序で通ったかどうかは気にしない
例) A → B, B → C を通ったことは分かっても
A → B → C を通ったかどうかは分からぬ
- ・ 脆弱性には、一連の操作を特定の順序で行う必要があるものも
例) Use After Free (UAF) : malloc → free → use
→ CFG エッジカバレッジガイドファザーはこのような脆弱性を効率的に発見できない！

UAF の例

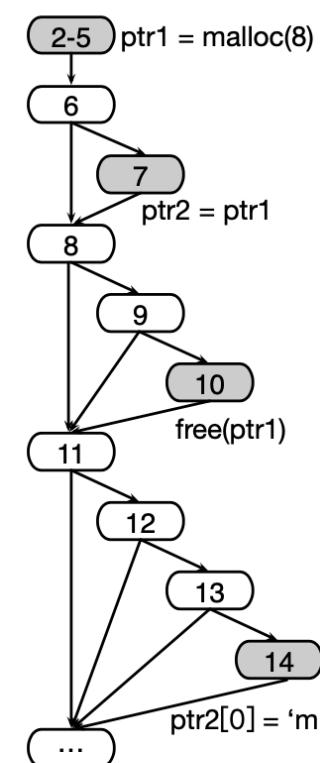
- $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ の順で通ったとき
UAF が発生する
`buf = "furseen"` など

```

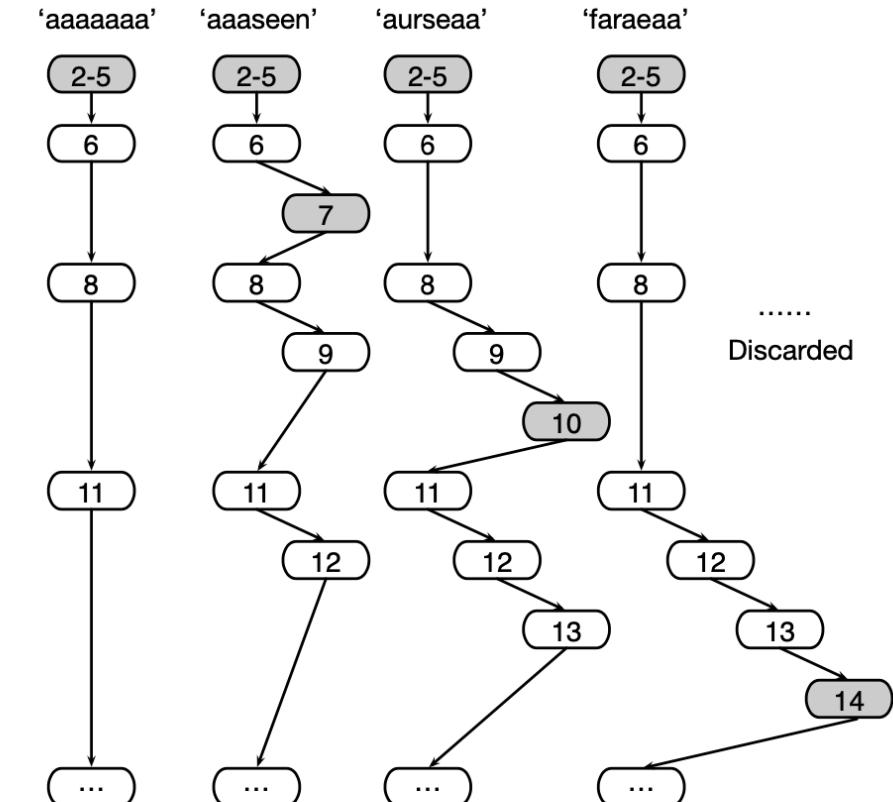
1 void main() {
2     char buf[7];
3     read(0, buf, 7);
4     char *ptr1 = malloc(8);
5     char *ptr2 = malloc(8);
6     if (buf[5] == 'e')
7         ptr2 = ptr1;
8     if (buf[3] == 's')
9         if (buf[1] == 'u')
10            free(ptr1);
11    if (buf[4] == 'e')
12        if (buf[2] == 'r')
13            if (buf[0] == 'f')
14                ptr2[0] = 'm';
15    ...
16 }
```

AFL の例

- シードに対しての変異が $6 \rightarrow 7, 9 \rightarrow 10, 13 \rightarrow 14$ のエッジを通る
- すべての CFG エッジを通ったため AFL はそれ以降の変異をすべて破棄する
→ 例えば、これ以降に $4 \rightarrow 7 \rightarrow 10$ を通るテストケースが生成されても、UAF の発見に近づくかもしれないが破棄される
- $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ を通るテストケースを生成するのは困難



(a) Control flow graph



(b) Fuzzing process in AFL

目次

1. AFL [1] の概要
2. カバレッジガイドファザーの問題点
3. UAFL [2] の概要
4. HTFuzz [3] の概要



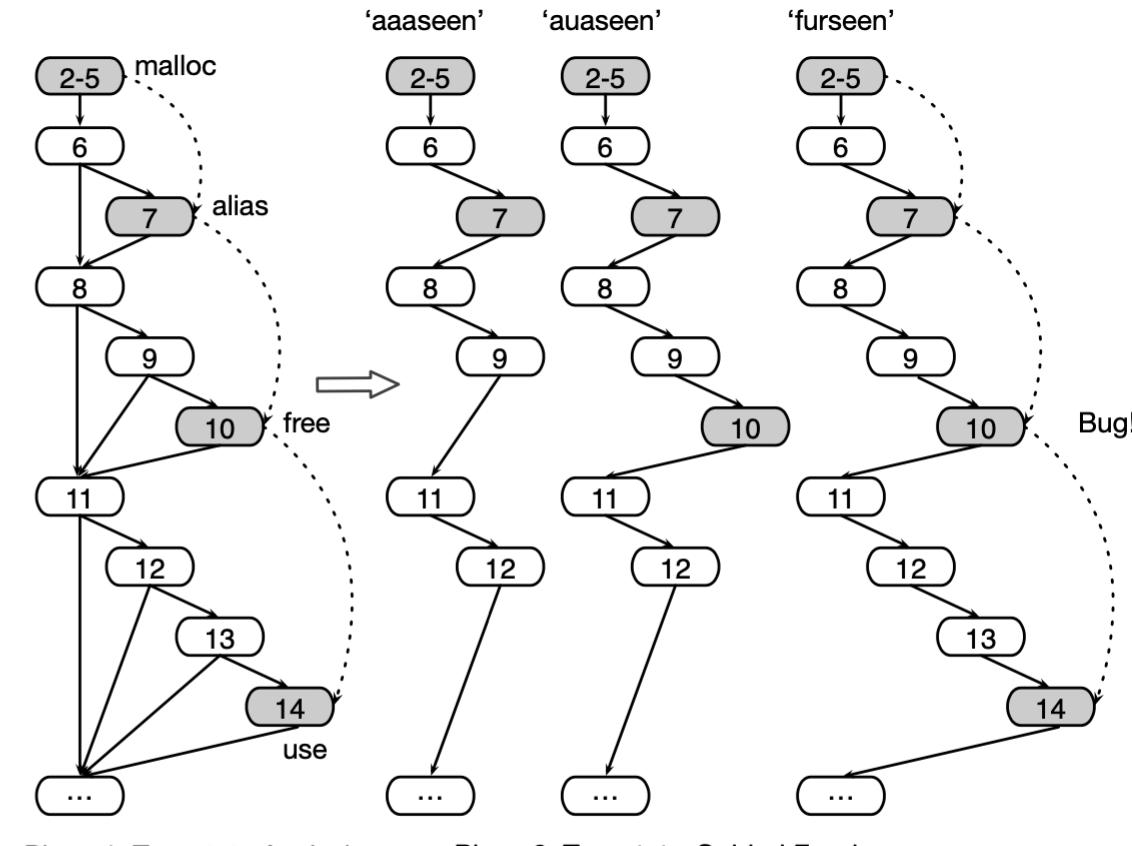
[1] <https://github.com/google/AFL>

[2] Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [ICSE '20]

[3] HTFuzz: Heap Operation Sequence Sensitive Fuzzing [ASE '22]

typestate ガイドファジング

- UAF は $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ で発生する
- AFL で破棄された $4 \rightarrow 7 \rightarrow 10$ を通る
テストケースは UAF の発見に近づく
ため残したい
- $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ を通るテスト
ケースを生成できるように
フィードバックを改良する



(c) Fuzzing process in UAFL

U AFL [2]

- ・一連の操作を特定の順序で実行したときに発生する脆弱性の検出を目的としたファザー

例) Use After Free (UAF) : malloc → free → use

UAFL の2つのステップ

1. 静的 typestate 解析

UAF であれば, `malloc` → `free` → `use` となる

操作シーケンス（命令列）を静的に求める

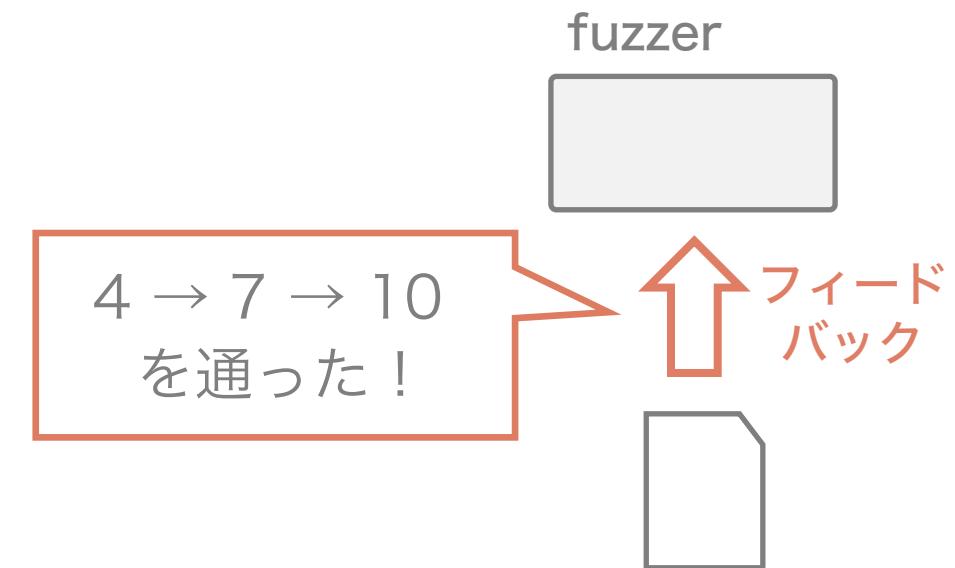
$4 \rightarrow 7 \rightarrow 10 \rightarrow 14$

2. typestate ガイドファジング

1で求めた操作シーケンスを実行する

テストケースを生成できるように

AFL を改良する



UAFL : 静的 typestate 解析

- typestate プロパティに違反する可能性のある操作シーケンスを静的に求める

例) Use After Free (UAF) :

`malloc → free → use` となる

操作シーケンスを静的に求める

- 右の例で $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ を求めたい

```

1 void main() {
2     char buf[7];
3     read(0, buf, 7);
4     char *ptr1 = malloc(8);
5     char *ptr2 = malloc(8);
6     if (buf[5] == 'e')
7         ptr2 = ptr1;
8     if (buf[3] == 's')
9         if (buf[1] == 'u')
10            free(ptr1);
11     if (buf[4] == 'e')
12         if (buf[2] == 'r')
13             if (buf[0] == 'f')
14                 ptr2[0] = 'm';
15     ...
16 }
```

UAFL : 静的 typestate 解析

UAF のアルゴリズム

- ・ すべての `malloc` に対して
 - ・ エイリアス解析からそのメモリを指すすべてのポインタを求め,
 - ・ そのポインタを使って `free`, `use` する命令を求め,
 - ・ $\text{malloc} \rightarrow \text{free} \rightarrow \text{use}$ が到達可能ならその命令列を UAF を引き起こす可能性のある操作シーケンスとして報告

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 

```

UAFL : 静的 typestate 解析

(アルゴリズム2行目)

malloc は 4行目と5行目

Algorithm 1: Typestate Analysis for Use-After-Free

input : A program P

output: A set of operation sequences S

```

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow find\_malloc(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow cal\_alias(m);$ 
5    $S_F \leftarrow find\_free(A, P);$ 
6    $S_U \leftarrow find\_use(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in$ 
      $S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\};$ 
8 return  $S;$ 

```



```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10       free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }

```

UAFL : 静的 typestate 解析

(01についてアルゴリズム4行目)

エイリアス解析から

01を指すポインタは ptr1, ptr2

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 

```



```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10       free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }

```

UAFL : 静的 typestate 解析

(01についてアルゴリズム5行目)

ptr1, ptr2 を使った free 文は
10行目

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 
```



```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10    free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

UAFL : 静的 typestate 解析

(01についてアルゴリズム6行目)

ptr1, ptr2 を使った use 文は
14行目

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 

```



```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10       free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }

```

UAFL : 静的 typestate 解析

(01についてアルゴリズム7行目)

$$s_m = 4, s_f = 10, s_u = 14$$

$s_m \rightarrow s_f \rightarrow s_u$ は到達可能

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 
```



```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10       free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

UAFL : 静的 typestate 解析

$4 \rightarrow 10 \rightarrow 14$ (エイリアス文も含めた
 $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$) の操作シーケンスは
UAF を引き起こす可能性がある

Algorithm 1: Typestate Analysis for Use-After-Free

```

input : A program  $P$ 
output: A set of operation sequences  $S$ 

1  $S \leftarrow \emptyset;$ 
2  $(S_M, M) \leftarrow \text{find\_malloc}(P);$ 
3 foreach  $(s_m, m) \in (S_M, M)$  do
4    $A \leftarrow \text{cal\_alias}(m);$ 
5    $S_F \leftarrow \text{find\_free}(A, P);$ 
6    $S_U \leftarrow \text{find\_use}(A, P);$ 
7    $S \leftarrow S \cup \{(s_m, s_f, s_u) | s_f \in S_F \wedge s_u \in$ 
      $S_U \wedge \text{is\_reachable}(s_m, s_f) \wedge \text{is\_reachable}(s_f, s_u)\};$ 
8 return  $S;$ 
```

```

1 void main() {
2   char buf[7];
3   read(0, buf, 7);
4   char *ptr1 = malloc(8); // 01
5   char *ptr2 = malloc(8); // 02
6   if (buf[5] == 'e')
7     ptr2 = ptr1;
8   if (buf[3] == 's')
9     if (buf[1] == 'u')
10       free(ptr1);
11   if (buf[4] == 'e')
12     if (buf[2] == 'r')
13       if (buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

UAFL：静的 typestate 解析

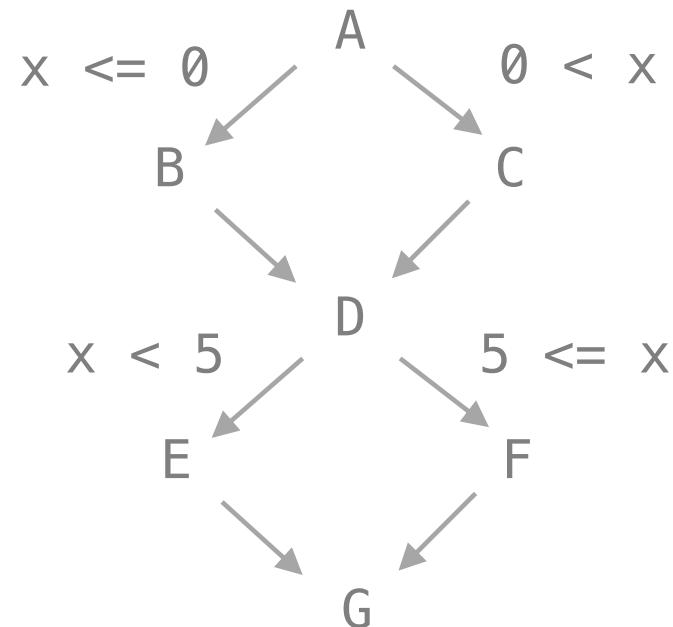
到達可能性

- ここで到達可能性の判定は path-insensitive に行う
path-insensitive : パスの条件を無視した解析 (\leftrightarrow path-sensitive)

Q. B から F は到達可能?
(B, D で x の値は変わらないとする)

path-sensitive :

path-insensitive :



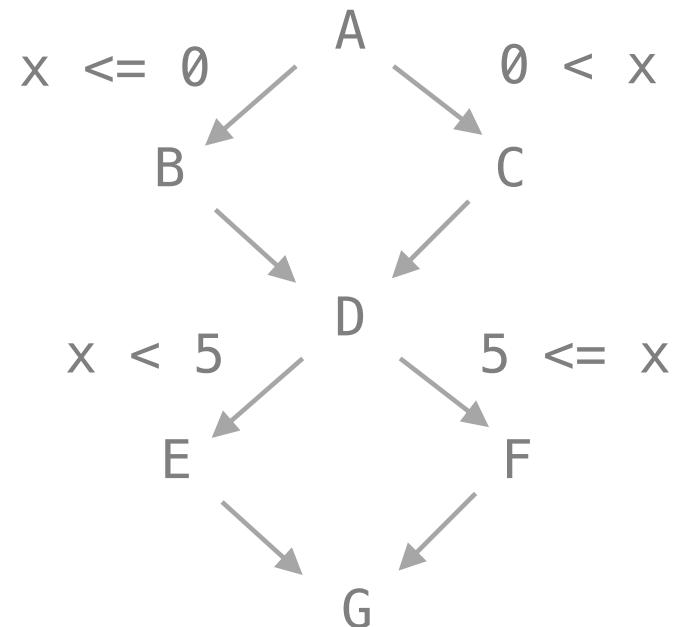
UAFL：静的 typestate 解析

到達可能性

- ここで到達可能性の判定は path-insensitive に行う
path-insensitive : パスの条件を無視した解析 (\leftrightarrow path-sensitive)

Q. B から F は到達可能?
(B, D で x の値は変わらないとする)

path-sensitive : 到達不可能
path-insensitive : 到達可能



UAFL：静的 typestate 解析

到達可能性

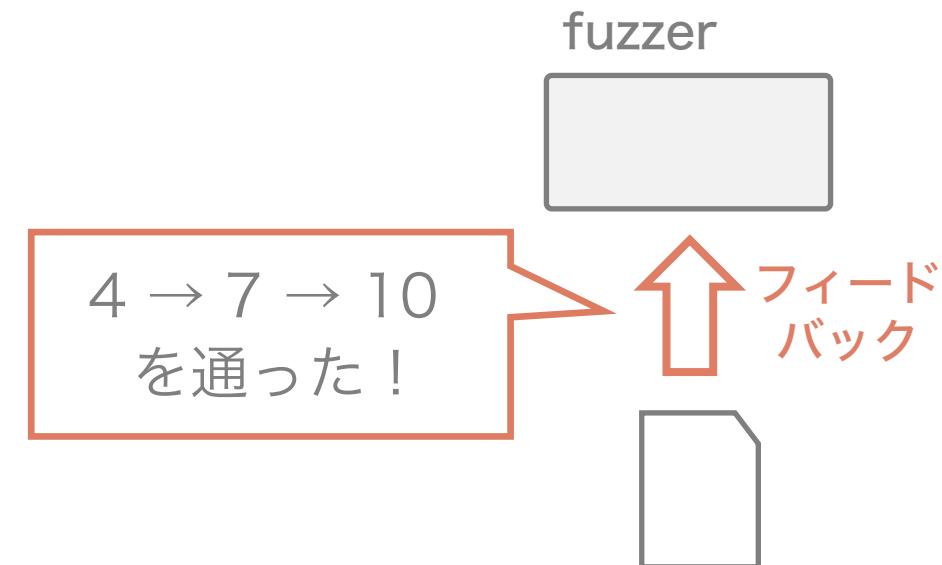
- ・ ここでの到達可能性の判定は path-insensitive を行う
path-insensitive : パスの条件を無視した解析 (\leftrightarrow path-sensitive)
- ・ 静的 typestate 解析で求められた操作シーケンスは
実行不可能なものも多い (false-positive が多い)
- ・ 本当にこの操作シーケンスが実行可能かどうかはファジングで調べる
 \rightarrow (無駄な操作シーケンスのガイドにファジングの多くの時間を費やす?)

UAFL : typestate ガイドファジング

- 静的 typestate 解析で特定した操作シーケンスを実行する入力を生成できるファザーを作りたい

操作シーケンス :

$4 \rightarrow 7 \rightarrow 10 \rightarrow 14$

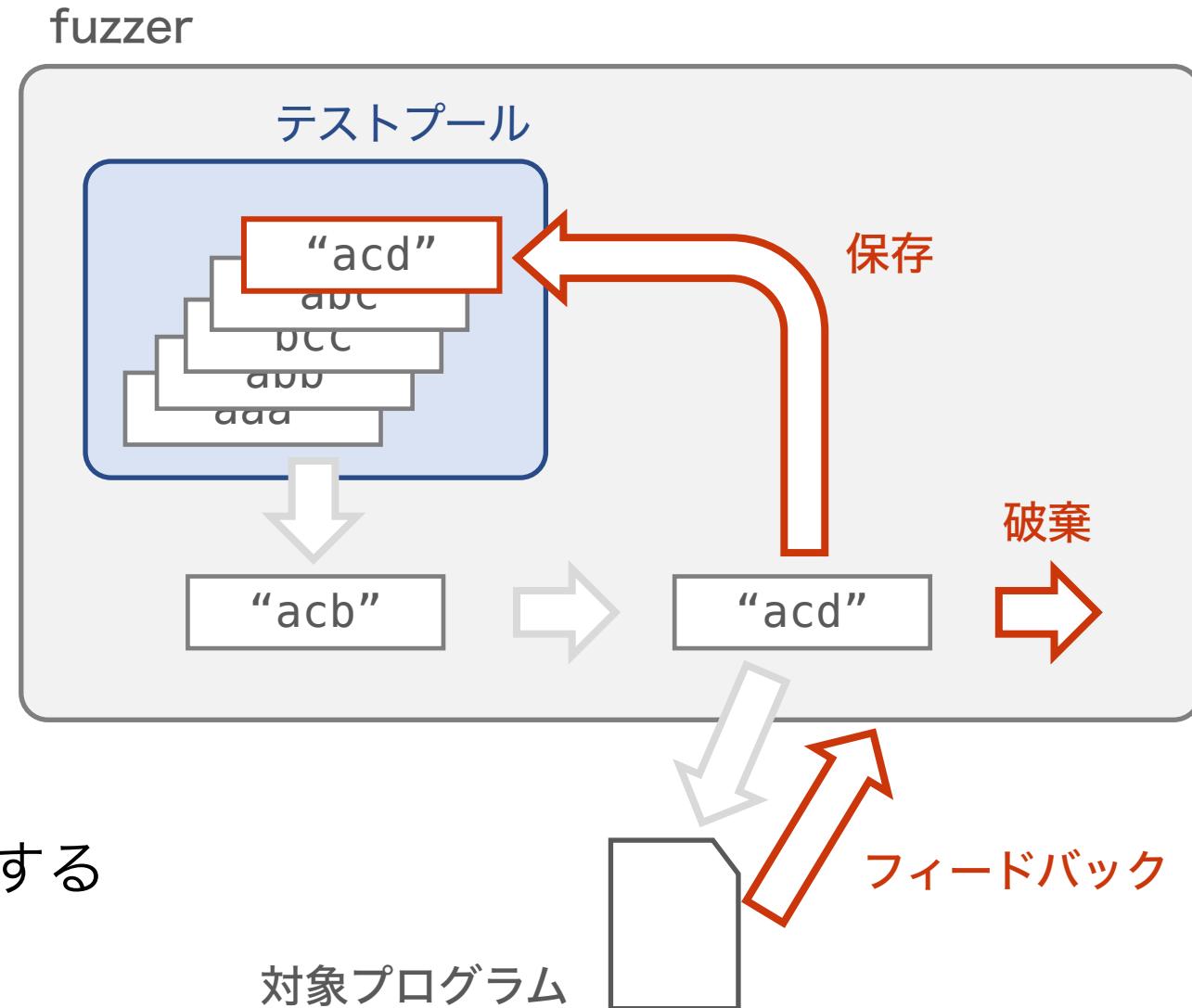


UAFL : シードの保存戦略

- $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ という操作シーケンスが与えられたとき

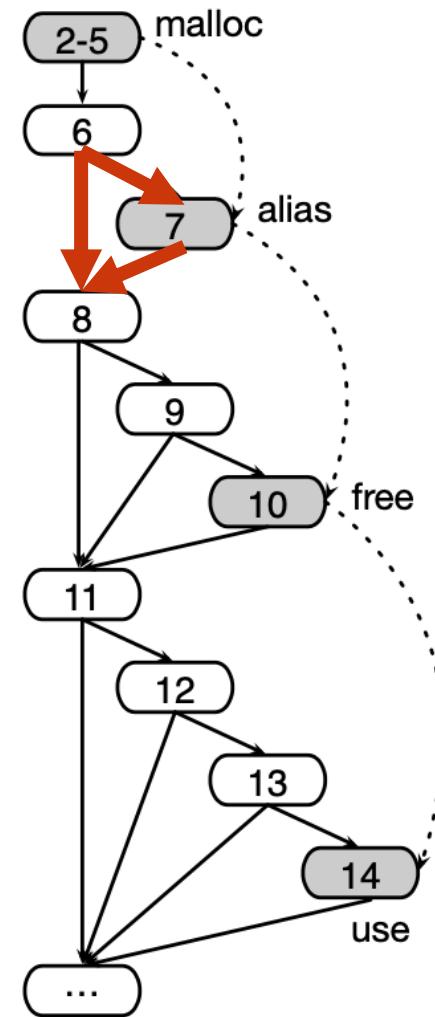
- $4 \rightarrow 7$
- $4 \rightarrow 7 \rightarrow 10$
- $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$

と順番にテストケースを保存できるようにし、最終的に操作シーケンスを満たすようにフィードバックを設計する



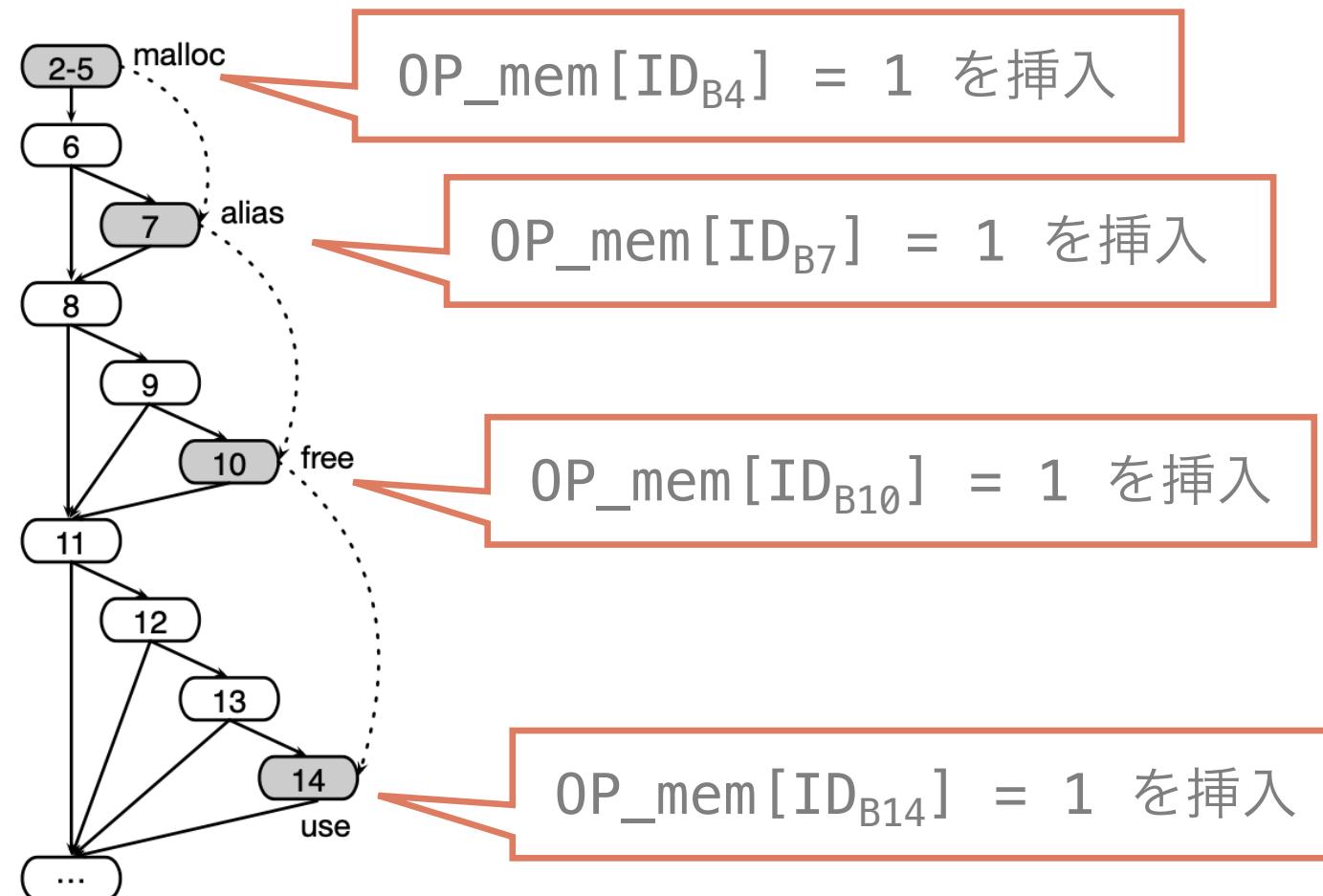
UAFL : シードの保存戦略

- ・操作シーケンス : 4 → 7 → 10 → 14
- ・直感的には、10行目を通ったときに
7行目を通ったかどうかを知りたい
- ・一回のプログラムの実行で以下の情報を記録
 - `OP_mem[]` : 操作シーケンス内の基本ブロックを
通ったか記録
 - `OPE_mem[]` : 操作シーケンスエッジをカバーしたか
記録



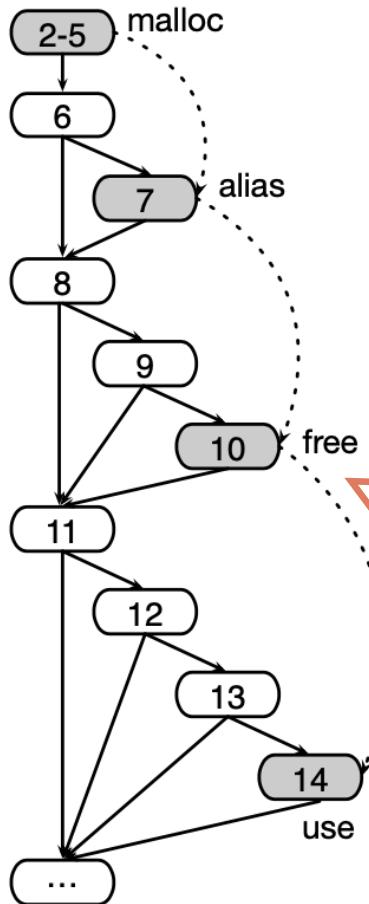
UAFL : シードの保存戦略

`OP_mem[]` : 操作シーケンス内の基本ブロックを通ったか記録



UAFL : シードの保存戦略

`OPE_mem[]` : 操作シーケンスエッジをカバーしたか記録



B₁₀ を通ったとき
 B₄, B₇ を通ったかどうかで場合分け

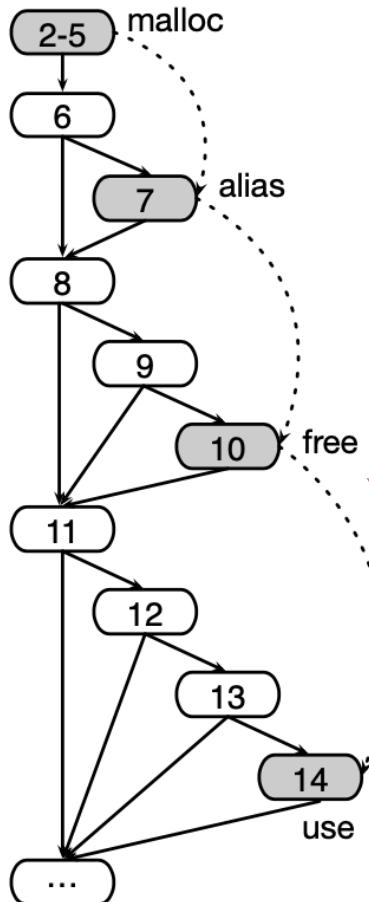
B₇ を通ったとき ($OP_mem[ID_{B7}] == 1$)
 $OPE_mem[ID_{B4} \ xor \ ID_{B7} \ xor \ ID_{B10}]++$

B₇ を通っていないとき ($OP_mem[ID_{B7}] != 1$)
 $OPE_mem[ID_{B4} \ xor \ ID_{B10}]++$

を挿入

UAFL : シードの保存戦略

`OPE_mem[]` : 操作シーケンスエッジをカバーしたか記録



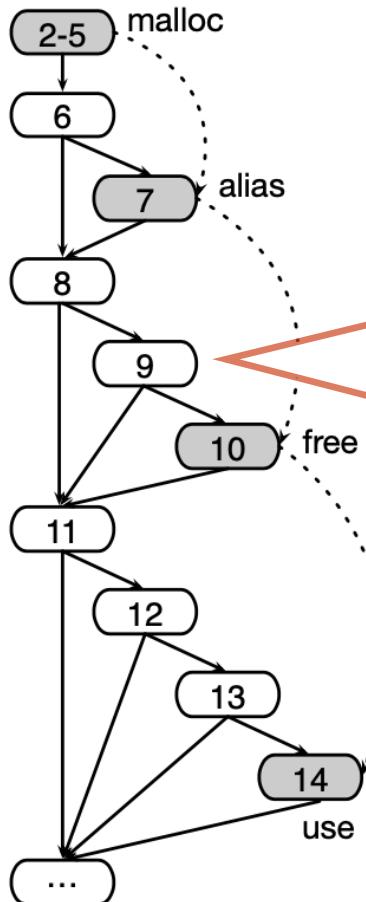
操作シーケンス $B_0 \rightarrow \dots \rightarrow B_i \rightarrow \dots \rightarrow B_n$ が与えられたとき, B_i に以下を挿入

```

tID = 0
for j in range(0, i):
    if OPE_mem[ID_Bj] == 1:
        tID = tID xor ID_Bj
    OPE_mem[tID xor ID_Bi]++
  
```

UAFL : シードの保存戦略

`OPE_mem[]` : 操作シーケンスエッジをカバーしたか記録



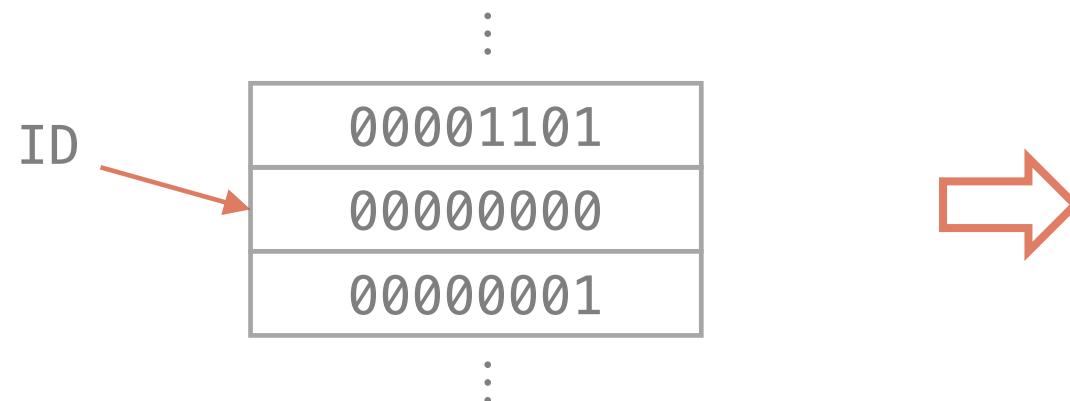
より正確には、 $7 \rightarrow 10$ の操作シーケンス
エッジをカバーするには、 $8 \rightarrow 9$ を
カバーする必要あり
(直接 $7 \rightarrow 10$ をカバーするのは難しい)
→ B_9 にも同様の計装を行う

UAFL : シードの保存戦略

`OPE_mem[]` : 操作シーケンスエッジをカバーしたか記録

`OPE_mem[ID] >= 1 :`

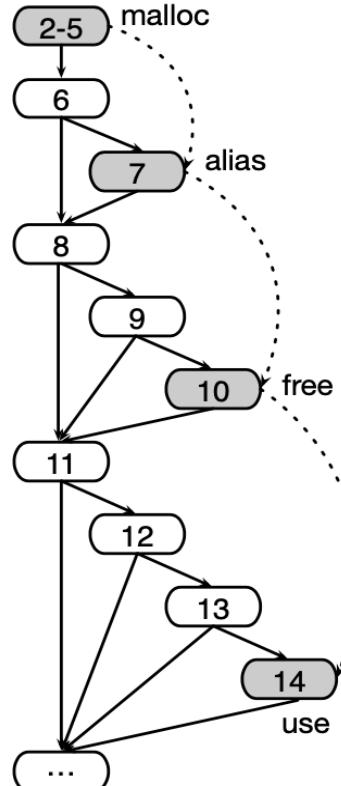
ファジングテスト全体のカバレッジ



新しい操作シーケンスエッジを
通過した興味深いテストケース
なので保存！

UAFL : シードの選択戦略

新しい操作シーケンスエッジをカバーした
テストケースを優先的に選択



操作シーケンス : $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$

新しく

- $4 \rightarrow 7 \rightarrow 9$
- $4 \rightarrow 7 \rightarrow 10$

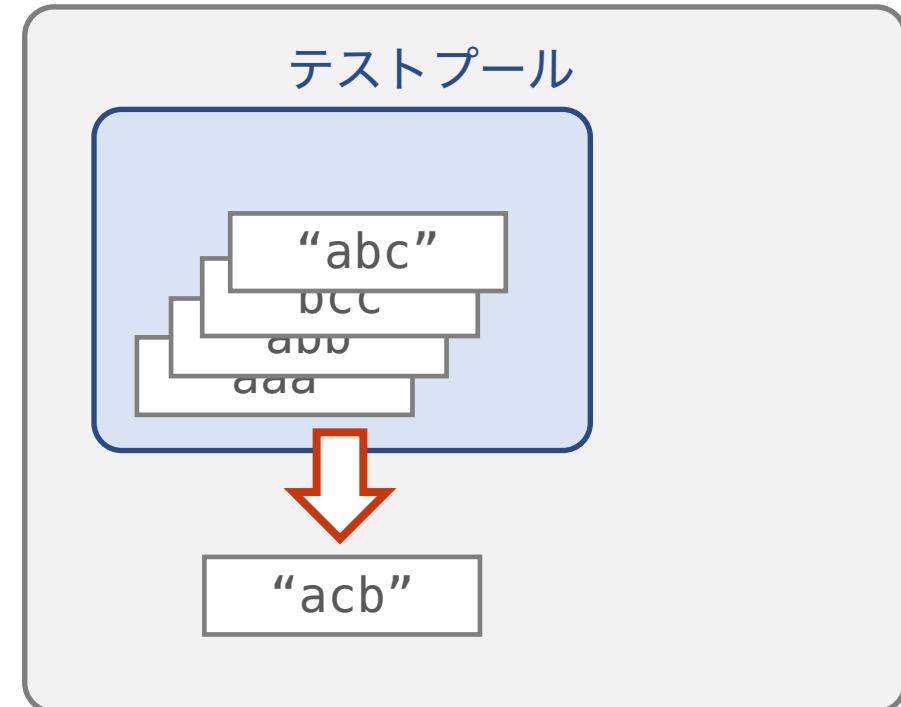
をカバーしたテストケースは保存



$4 \rightarrow 7 \rightarrow 10$

のテストケースを優先

fuzzer



UAFL：シードの変異戦略

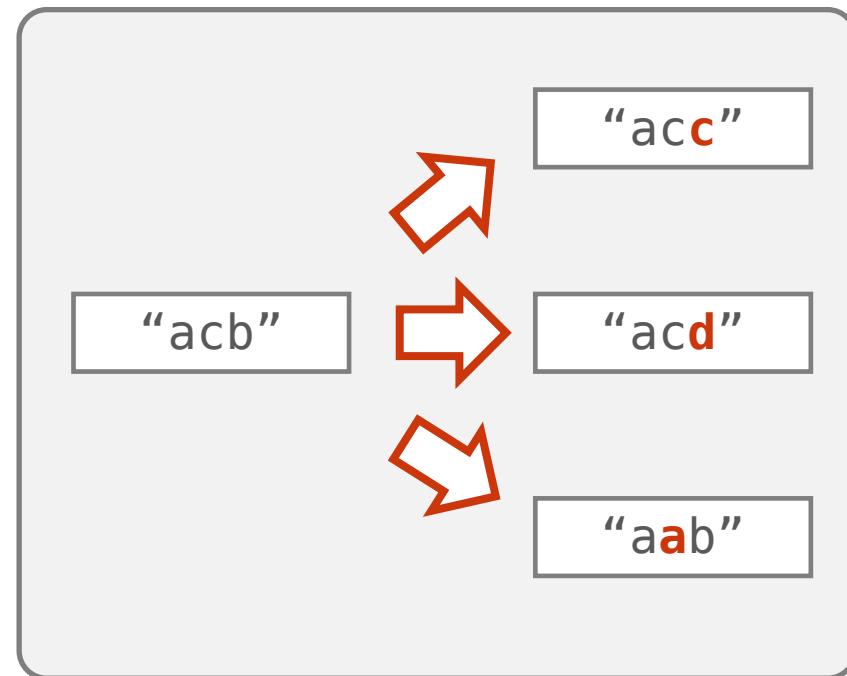
- シードを何回変異させるか？ (power scheduling)

より多くの操作シーケンスエッジをカバーする
テストケースに多くの変異機会を割り当て

$$UAFL_energy(i) = AFL_energy(i) * \left(1 + \frac{c_OSe}{t_OSe} \right)$$

- $UAFL_energy(i)$: シード i の変異回数に比例
- $AFL_energy(i)$: AFL のエネルギー (入力ファイルの小ささや実行時間に影響)
- t_OSe : 操作シーケンスエッジの総数
- c_OSe : シード i によってカバーされる操作シーケンスの総数

fuzzer



UAFL : シードの変異戦略

- シードのどのバイトを変異させるか？

$7 \rightarrow 10$ をカバーするには `buf[3]`, `buf[1]` に
変異を集中させるべき

情報フロー (information flow) 解析 [5] を用いて
入力の各バイトが $7 \rightarrow 10$ の間の条件

```
8 if (buf[3] == 's')
9 if (buf[1] == 'u')
```

にどれだけ影響を与えるか計算

```
1 void main() {
2     char buf[7];
3     read(0, buf, 7);
4     char *ptr1 = malloc(8);
5     char *ptr2 = malloc(8);
6     if (buf[5] == 'e')
7         ptr2 = ptr1;
8     if (buf[3] == 's')
9         if (buf[1] == 'u')
10        free(ptr1);
11     if (buf[4] == 'e')
12        if (buf[2] == 'r')
13            if (buf[0] == 'f')
14                ptr2[0] = 'm';
15    ...
16 }
```

UAFL : シードの変異戦略

- シードのどのバイトを変異させるか?
 - 入力の各バイトを変異させつつ、入力バイトと条件式の変数間の情報フロー強度 (information flow strength) を計算
- 情報フロー強度が高いバイトほど高い変異確率を割り当てる

```

1 void main() {
2     char buf[7];
3     read(0, buf, 7);
4     char *ptr1 = malloc(8);
5     char *ptr2 = malloc(8);
6     if (buf[5] == 'e')
7         ptr2 = ptr1;
8     if (buf[3] == 's')
9         if (buf[1] == 'u')
10            free(ptr1);
11    if (buf[4] == 'e')
12        if (buf[2] == 'r')
13            if (buf[0] == 'f')
14                ptr2[0] = 'm';
15    ...
16 }
```

UAFL : シードの変異戦略

- シードのどのバイトを変異させるか？

例：入力の 3, 6 バイト目（0 始まり）と
8 行目の条件式の変数 buf[3] を考える

各バイトを 10 回ずつ変異させると
情報フロー強度は

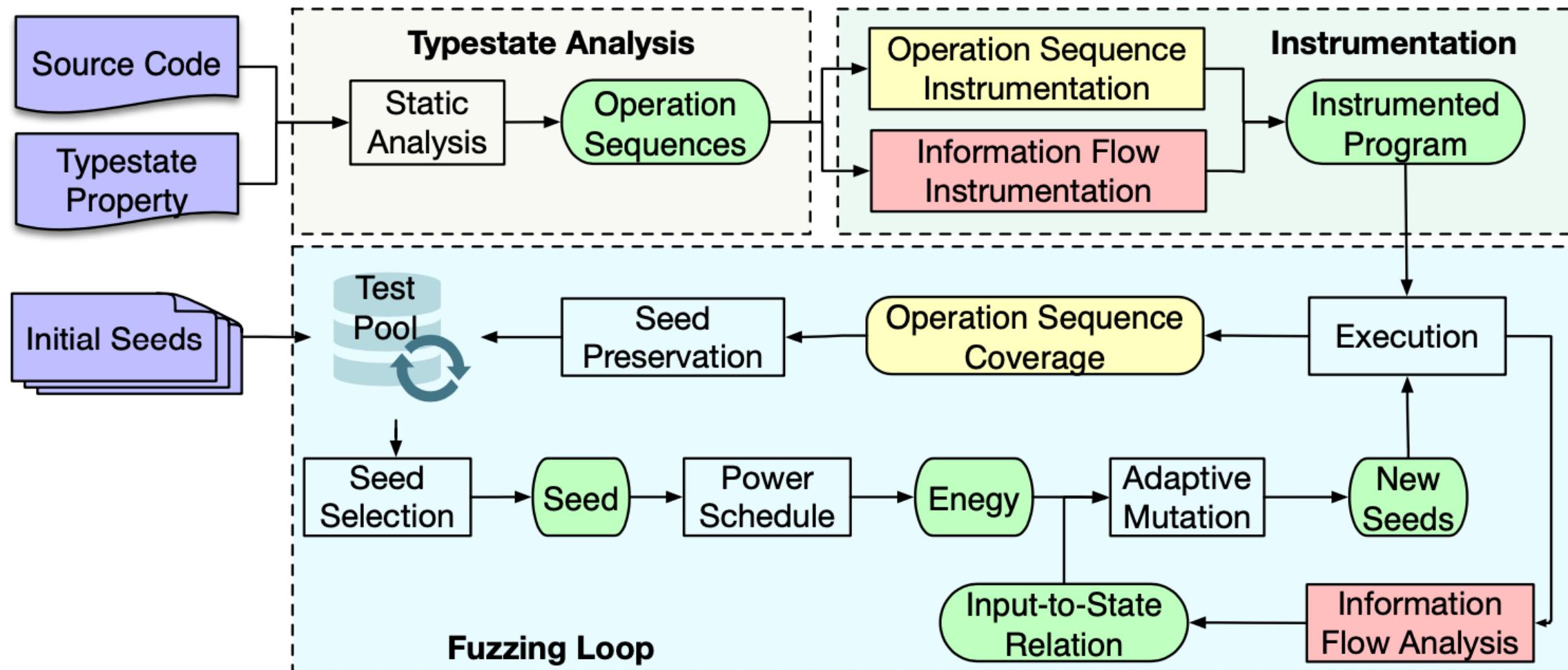
- 3 バイト目 : 3.3
- 6 バイト目 : 0

→ 3 バイト目に高い変異確率を割り当てる

```

1 void main() {
2     char buf[7];
3     read(0, buf, 7);
4     char *ptr1 = malloc(8);
5     char *ptr2 = malloc(8);
6     if (buf[5] == 'e')
7         ptr2 = ptr1;
8     if (buf[3] == 's')
9         if (buf[1] == 'u')
10            free(ptr1);
11    if (buf[4] == 'e')
12        if (buf[2] == 'r')
13            if (buf[0] == 'f')
14                ptr2[0] = 'm';
15    ...
16 }
```

UAFL のまとめ





日本酒呑みくらべ
on 電車

目次

1. AFL [1] の概要
2. カバレッジガイドファザーの問題点
3. UAFL [2] の概要
4. HTFuzz [3] の概要



HTFuzz [3] の概要

事前に脆弱な操作の候補を与えることなく, ヒープ操作シーケンスの多様性を高めるように設計された, ヒープ操作シーケンスガイドファザー

従来手法の問題点

- UAFL [2] や LTL-Fuzzer [7], UAFuzz [8] などの時間的ヒープ脆弱性に焦点を当てたファザーは, 静的解析 や 事前情報 から脆弱な操作の候補を絞るため, 精度に限界がある
 - 例えば, 関数ポインタ, スレッドインターリープなどを正確に静的に解析するのは困難

[2] Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [ICSE '20]

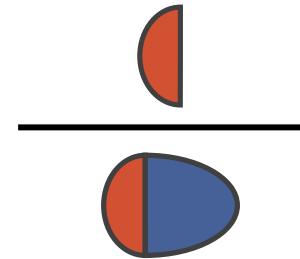
[3] HTFuzz: Heap Operation Sequence Sensitive Fuzzing [ASE '22]

[7] Linear-time Temporal Logic guided Greybox Fuzzing [ICSE '22]

[8] Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities [RAID '22]

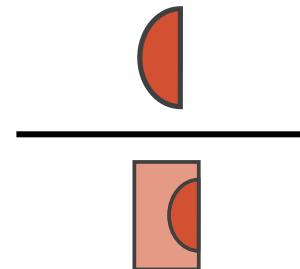
Precision と Recall

precision (適合率)

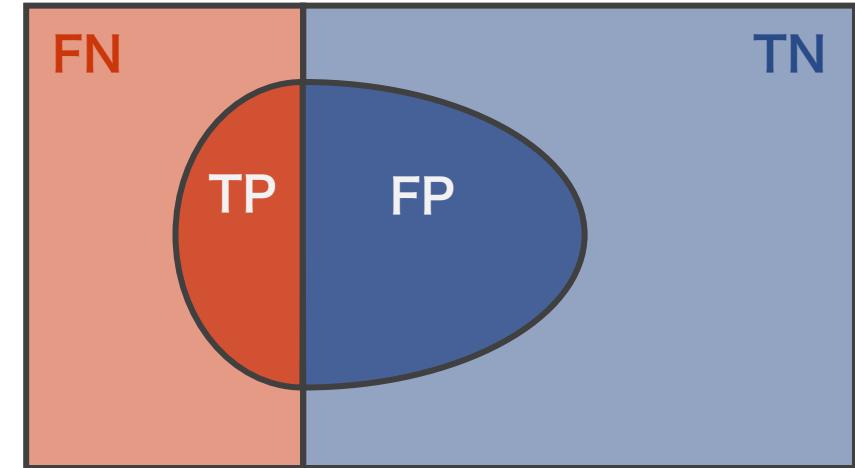


静的解析が脆弱と判定したもののうち
実際に脆弱な操作の割合

recall (再現率)



すべての脆弱な操作のうち
静的解析が脆弱と判定した操作の割合



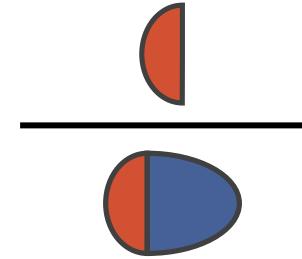
↑ 脆弱な操作 脆弱でない操作 ↑



: 静的解析が脆弱な操作と判定

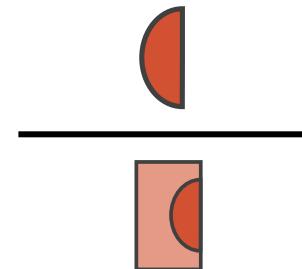
Precision と Recall

precision (適合率)

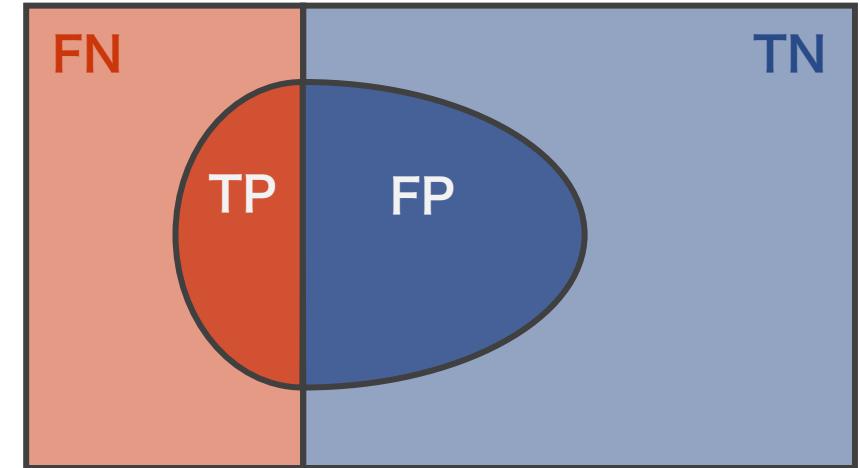


静的解析が脆弱と判定したもののうち
実際に脆弱な操作の割合

recall (再現率)



すべての脆弱な操作のうち
静的解析が脆弱と判定した操作の割合



↑ 脆弱な操作 脆弱でない操作 ↑

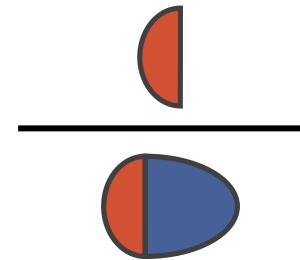


: 静的解析が脆弱な操作と判定

Q. 静的解析が sound のとき
100% となるのは？

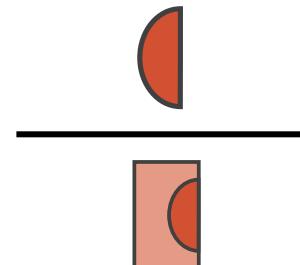
Precision と Recall

precision (適合率)

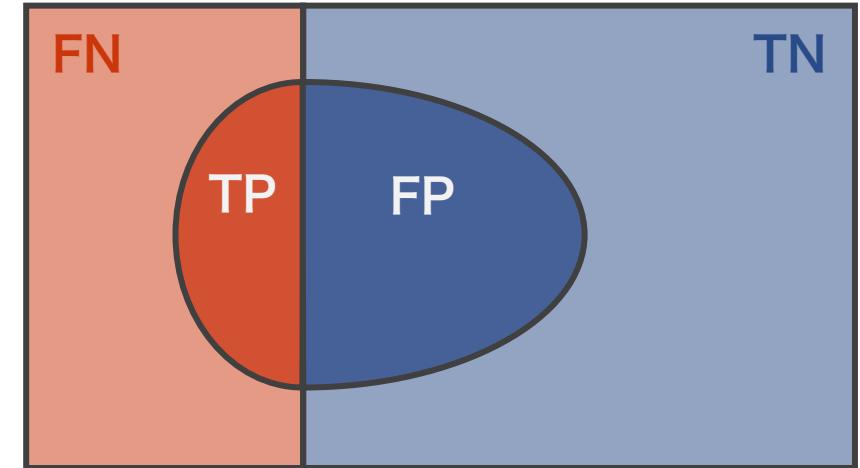


静的解析が脆弱と判定したもののうち
実際に脆弱な操作の割合

recall (再現率)



すべての脆弱な操作のうち
静的解析が脆弱と判定した操作の割合



↑ 脆弱な操作 脆弱でない操作 ↑



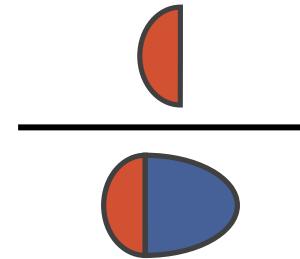
: 静的解析が脆弱な操作と判定

Q. 静的解析が sound のとき
100% となるのは？

A. recall

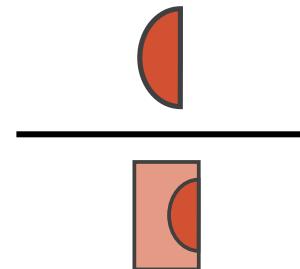
Precision と Recall

precision (適合率)



静的解析が脆弱と判定したもののうち
実際に脆弱な操作の割合

recall (再現率)



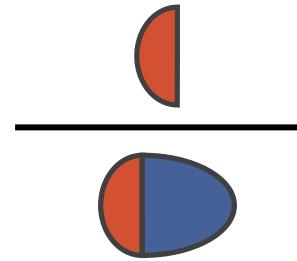
すべての脆弱な操作のうち
静的解析が脆弱と判定した操作の割合

← HTFuzz が問題に挙げている
のは recall の方

静的解析で見逃す脆弱な操作が
存在するという主張

Precision と Recall

precision (適合率)



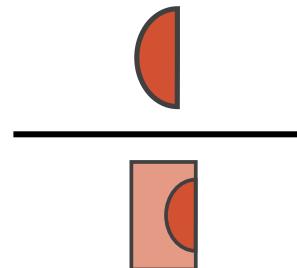
静的解析が脆弱と判定したもののうち
実際に脆弱な操作の割合

← precision も問題では？

UAFL の静的解析では多くの
false-positive が存在

→ 多くの無駄な候補によって
ファザーが脆弱ではない操作に
ガイドされる

recall (再現率)



すべての脆弱な操作のうち
静的解析が脆弱と判定した操作の割合

← HTFuzz が問題に挙げている
のは recall の方

静的解析で見逃す脆弱な操作が
存在するという主張

HTFuzz

事前に脆弱な操作の候補を与えることなく, ヒープ操作シークエンスの多様性を高めるように設計された, ヒープ操作シークエンスガイドファザー

- 1. 制御フローの多様性
脆弱な操作は特定のパスを通ったときに発生する
- 2. データフローの多様性
複雑なポインタ管理が行われた方が脆弱な操作が起きやすい

HTFuzz

事前に脆弱な操作の候補を与えることなく, ヒープ操作シークエンスの多様性を高めるように設計された, ヒープ操作シークエンスガイドファザー

3つのポイント

- ・シードの保存戦略：新しいヒープ操作シークエンスフィードバックを導入し, 制御フローの多様性を高める
- ・シードの選択戦略：実行中にアクセスされるポインタをカウントし, データフローの多様性を高める
- ・シードの変異戦略：コードカバレッジと操作シークエンスの2つのフィードバックを連携させる

HTFuzz : シードの保存戦略

新しいヒープ操作シーケンスフィードバックを導入し, 制御フローの多様性を高める

理想：各オブジェクトが確保されてからの操作をすべて追跡し,
新しい操作シーケンスが見つかったら保存する

→ オーバーヘッドが高すぎる

ヒープ操作とアドレスを関連づけるために, 動的なティント伝播を行ったが,
非効率なファザーとなった

HTFuzz : シードの保存戦略

新しいヒープ操作シーケンスフィードバックを導入し, 制御フローの多様性を高める

理想：各オブジェクトが確保されてからの操作をすべて追跡し,
新しい操作シーケンスが見つかったら保存する

→ オーバーヘッドが高すぎる

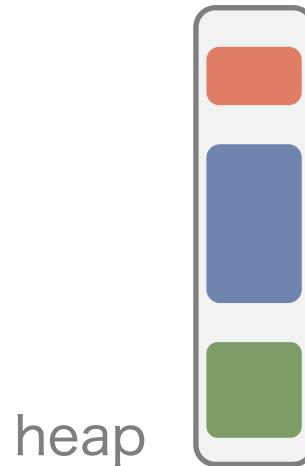
実際の設計：

- ・ ヒープのどこへの操作かは見ない
- ・ メモリアクセスポイントで最新のヒープ操作を複数個記録

新しいヒープ操作のシーケンスが見つかったら保存する

HTFuzz : シードの保存戦略

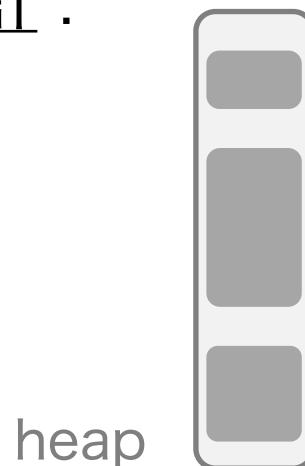
理想 :



$0 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow \underline{2} \rightarrow \dots$

$\text{HeapOpSeq}[2] = 0 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2$

実際の設計 :

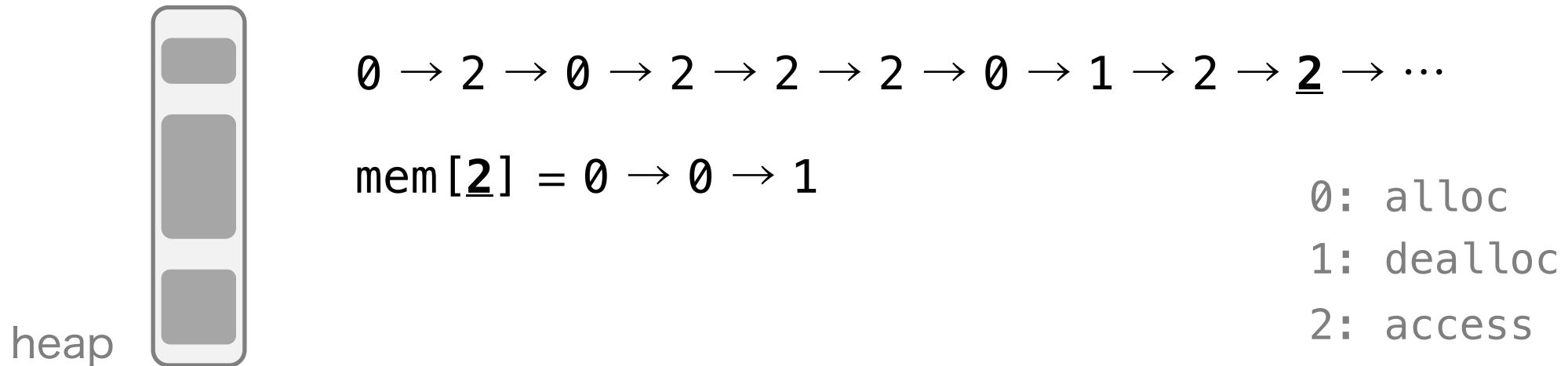


$0 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow \underline{2} \rightarrow \dots$

$\text{HeapOpSeq}[2] = 0 \rightarrow 0 \rightarrow 1$

0 : alloc
 1 : deallocate
 2 : access

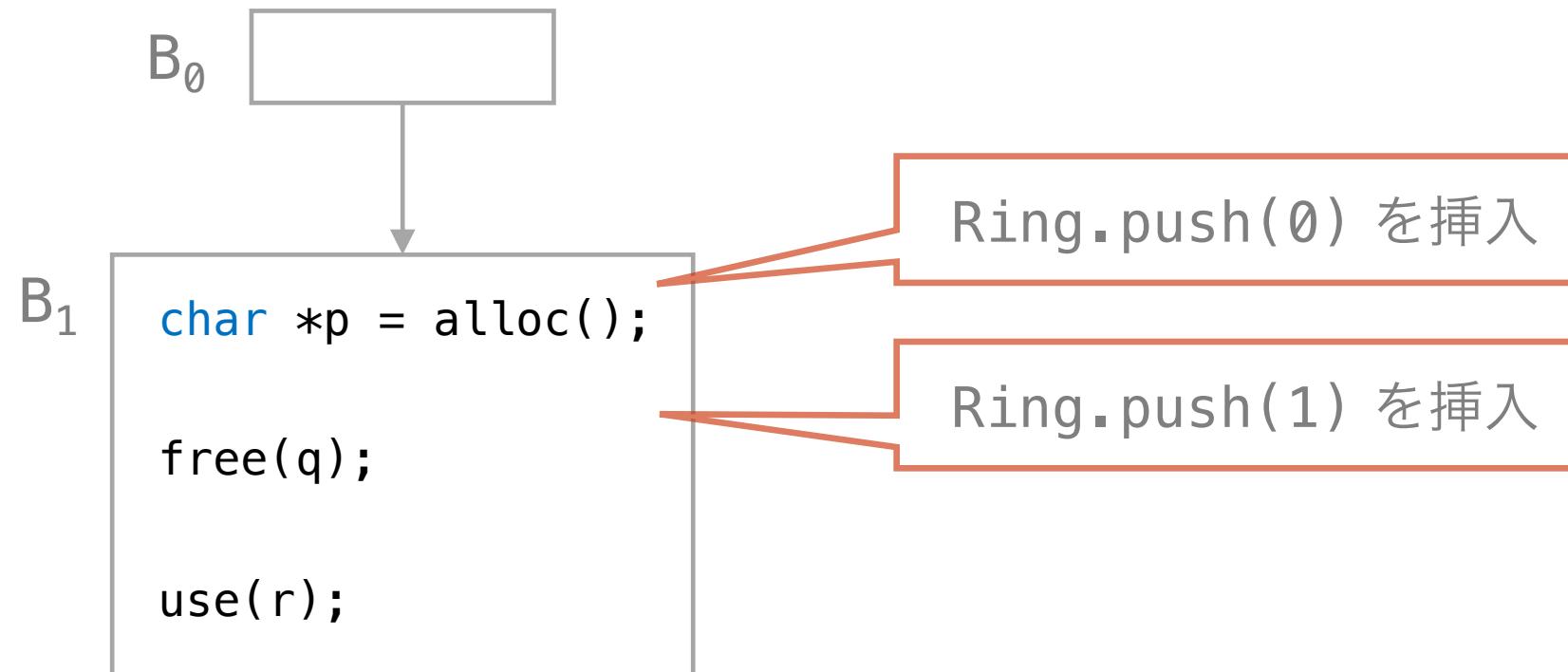
HTFuzz : シードの保存戦略



- alloc, dealloc 操作を長さ L のリングバッファで記録
- ヒープアクセスがあったときに、現在のリングバッファの状態をアクセスのプログラム位置をインデックスとして記録
- 新しいリングバッファの状態が見つかったとき、テストケースを保存
→ HTFuzz はコードカバレッジベースの興味深いテストケースも保存

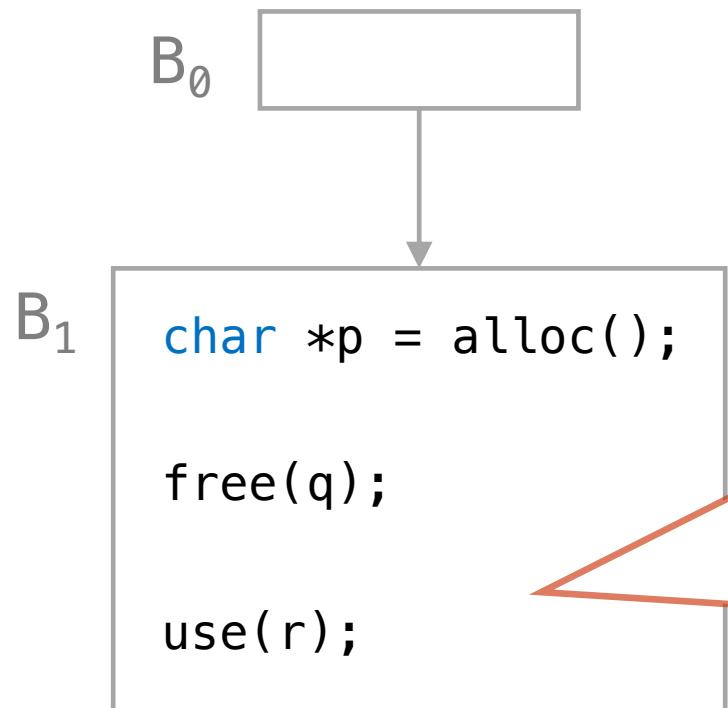
HTFuzz : フィードバックの計装

alloc・dealloc 操作のリングバッファへの保存



HTFuzz : フィードバックの計装

ヒープ操作シーケンスの bitmap への保存



- ・リングバッファの状態を整数値に変換
例 : $1 \rightarrow 0 \rightarrow 1$ なら 5
- ・各 BB で最初のアクセス命令だけに計装

$\text{hash} = \text{Encodeing}(\text{Ring})$
 $\text{HeapOpSeq}[\text{ID}_{B_0 \rightarrow B_1} \text{ xor } \text{hash}]++$
 を挿入

HTFuzz : シードの選択戦略

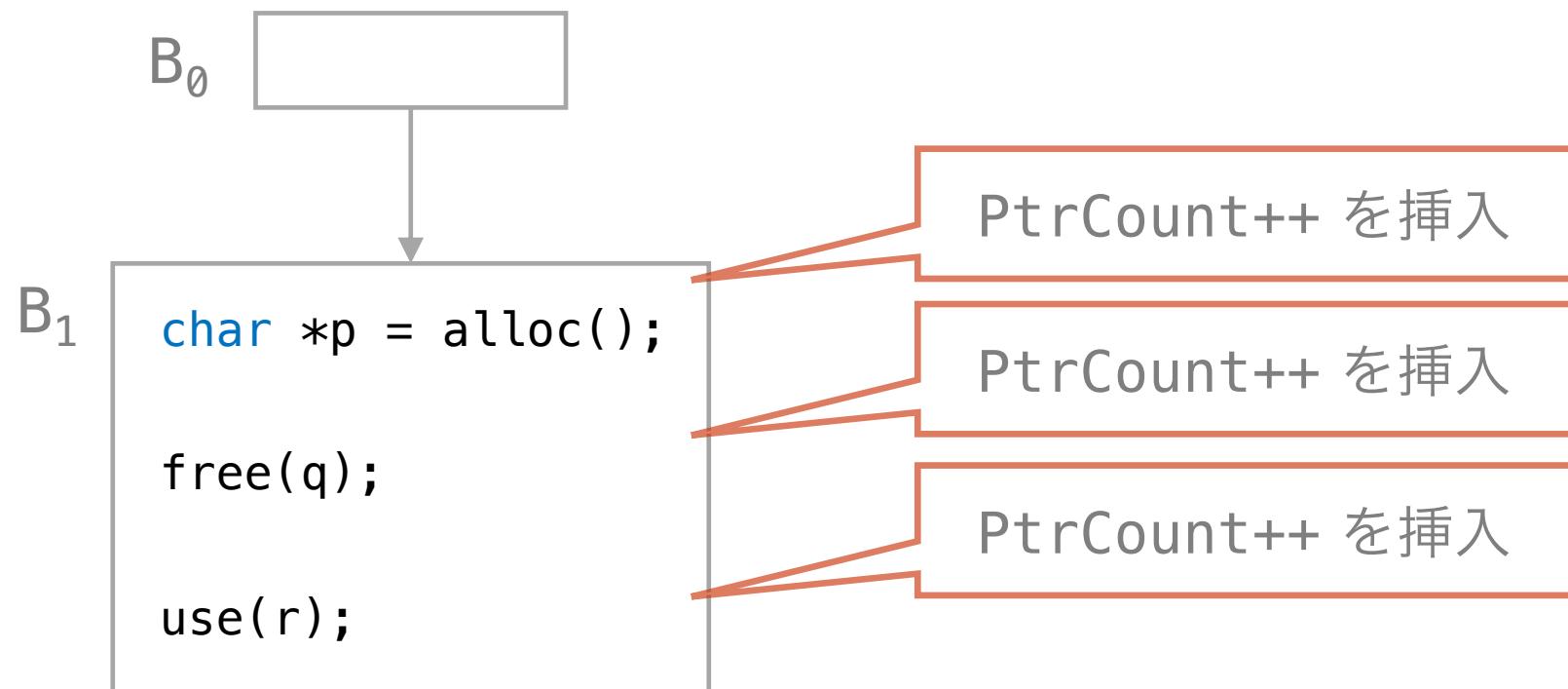
データフローの多様性 :

複雑なポインタ管理が行われた方が脆弱な操作が起きやすい

- ・正確にポインタを追跡するのはオーバーヘッドの問題から困難
 - ポインタがアクセスされた回数を記録
 - アクセス回数が多いシードが優先的に選択される

HTFuzz : フィードバックの計装

ポインタアクセス回数の記録



HTFuzz : シードの変異戦略

コードカバレッジと操作シーケンスの2つのフィードバックを連携させる

Effector map

- ・ シードのどのバイトを変異させるか?
 - AFL は各バイトを反転させて、実行パスが変化するかを観察
 - 実行パスが変化したバイトのみを変異させる
(effector map : 興味深いバイトを記録)
- ・ effector map はコードカバレッジに最適化
 - 新しい操作シーケンスに寄与する興味深いバイトを無視するかも
- ・ effector map を操作シーケンスカバレッジにも寄与するように変更

HTFuzz : シードの変異戦略

コードカバレッジと操作シーケンスの2つのフィードバックを連携させる

MOPT

- ・ シードをどのように変異させるか？（バイト反転, 加減算, など）
→ AFL はランダム
- ・ MOPT [9] : フィードバックから変異演算子を動的にスケジューリング
- ・ MOPT も操作シーケンスフィードバックに適応するように変更

HTFuzz : 手法のまとめ

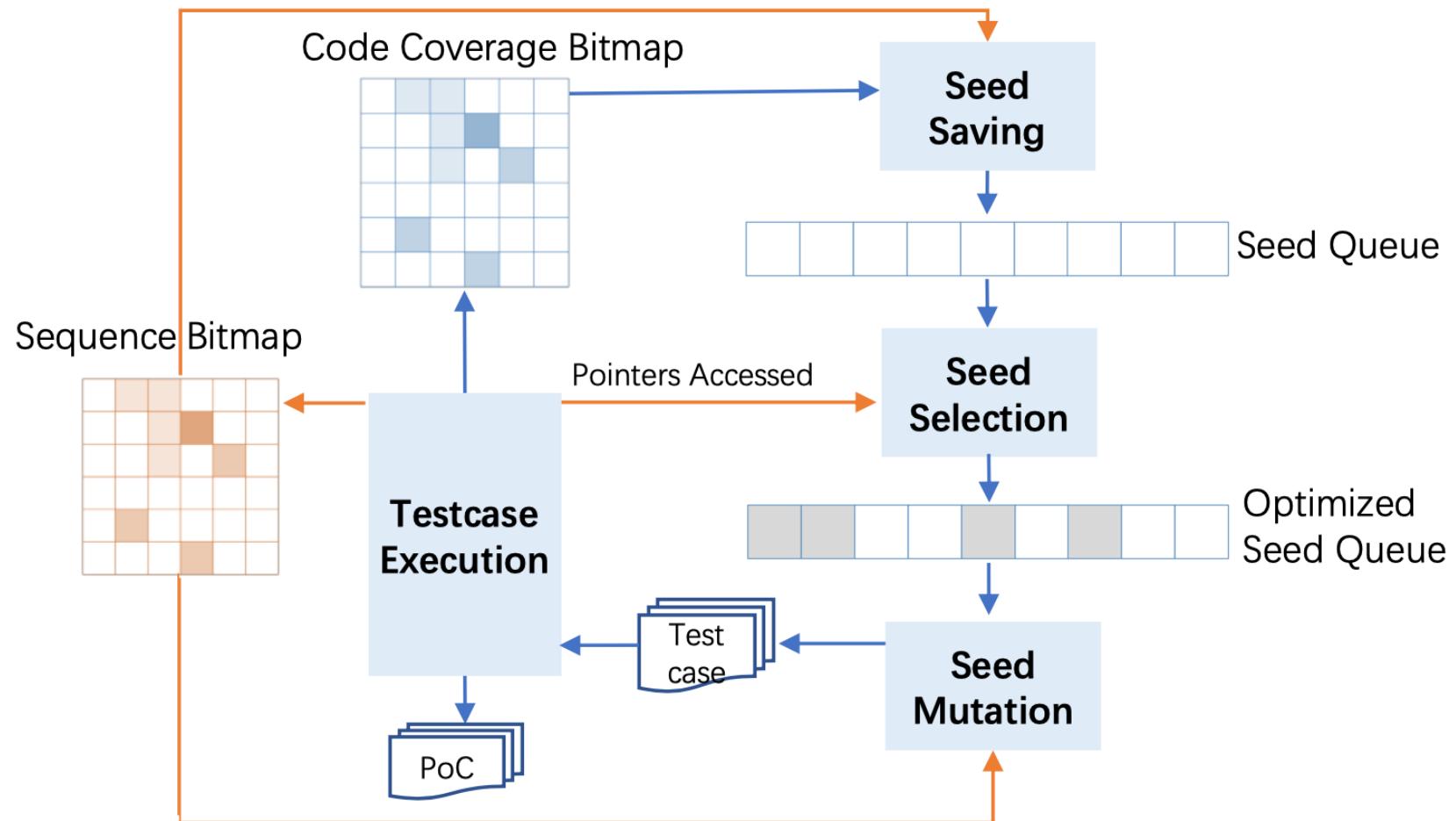


Figure 3: Framework of HTFuzz.

HTFuzz : 評価

RQ1 : HTFuzz は HT-Vuls を発見するのに有効か？

RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

RQ3 : HTFuzz は他のファズラーと比較してどうか？

HT-Vuls : heap-based temporal vulnerability (ヒープ時間的脆弱性)

HTFuzz：評価

比較するファザー

- 操作シーケンス情報を使用するファザー, ベースラインファザー,
操作シーケンス情報を部分的に使用するファザー

Work	Seq	Seed Saving	Seed Selection	Seed Mutation
AFL	○	new branches	file_size*time	multi-mu-ops
Angora	○	context-aware branch	gradient decent	AFL's
MOPT	○	AFL's	AFL's	PSO algorithm
AFL-sen-wa	●	memory-write-aware branch	AFL's	AFL's
AFL-sen-ma	●	memory-access-aware branch	AFL's	AFL's
TortoiseFuzz	●	coverage accounting	coverage accounting	AFL's
Memlock	●	memory usage peak value	AFL's	AFL's
PathAFL	●	path coverage	neighbour coverage	AFL's
Ankou	●	combinatorial branch difference	combinatorial distance	AFL's
UAFL	●	typestate sequence	sequence coverage	seq-aware mutation
LTL-fuzzer	●	typestate sequence	sequence similarity	AFL's
UAFuzz	●	cut-edge coverage	sequence similarity	AFL's
HTFuzz	●	heap operation sequence	pointer accessed	seq-aware PSO algorithm

HTFuzz : 評価

Performance metrics

- ・発見した HT-Vuls の数
- ・ヒープ操作シーケンス量
シーケンスビットマップからその量をカウント
- ・コードカバレッジ
`gcov` を用いて行カバレッジで比較する

HTFuzz : 評価

Configuration parameters

- ・各プログラムは72時間テスト
- ・各実験を8回実施
- ・すべての実験は以下の環境で行う
 - ・OS : Ubuntu LTS 16.04
 - ・CPU : Intel Xeon CPU E5-2630 v3 プロセッサ (2.40GHZ, 32Core)
 - ・メモリ : 32GB

HTFuzz : RQ1

RQ1 : HTFuzz は HT-Vuls を発見するのに有効か？

- HTFuzz は 74 個の HT-Vuls を含む 92 個の脆弱性を発見
- そのうち 37 個が新しい脆弱性で、32 個が新しい HT-Vuls
- 約半数 (34/74) は 10 時間以内に発見

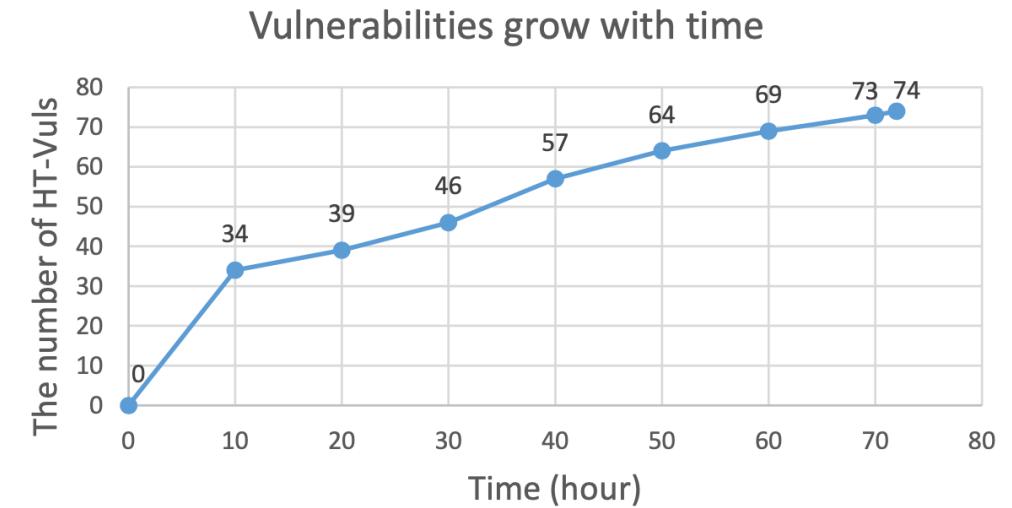


Figure 4: Unique HT-Vuls accumulated over time.

HTFuzz : RQ1

RQ1 : HTFuzz は HT-Vuls を発見するのに有効か？

	HTFuzz	AFL	Memlock	AFL-sen-ma	AFL-sen-mw	PathAFL	Tofuzz	MOPT	Angora	Ankou	Total
0day	37-32-0	24-22-0	9-7-0	9-9-0	9-9-0	21-18-0	20-18-0	21-19-0	8-8-0	26-22-0	37-32-0
0day-non-CVE	-	3-0-0	3-0-0	-	-	2-0-0	1-0-0	-	3-2-2	6-1-1	11-3-3
1day	55-42-0	37-26-4	29-20-2	16-12-0	11-9-0	28-23-4	30-20-2	46-32-1	27-25-10	49-36-2	87-66-24
Sum	92-74-0	64-48-4	41-27-2	25-21-0	20-18-0	51-41-4	51-38-2	67-51-1	42-37-14	81-59-3	135-101-27

X-Y-Z : X : 発見したすべての脆弱性の数

Y : 発見した HT-Vuls の数

Z : 発見した HT-Vuls の中で HTFuzz が見逃した数

- HTFuzz は他のファーザーと比較して HT-Vuls の発見に効果的
→ 特に 0day 脆弱性は他のファーザーが発見したすべての HT-Vuls を発見している

HTFuzz : RQ2

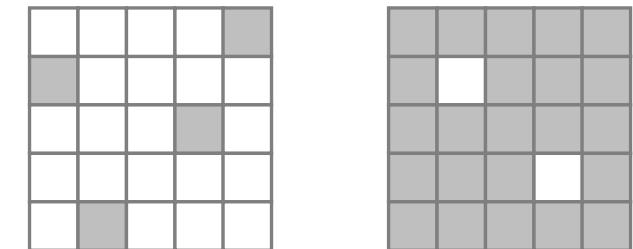
RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

ハイパーパラメータの設定

- ヒープ操作シーケンスビットマップのサイズ 2^K

→ 高いビットマップ密度を保持したい

しかし, 密度が高すぎてもハッシュが衝突する



- リングバッファの長さ L

→ L が大きいほど操作シーケンスへの感度は高くなる

しかし, オーバーヘッドが大きくなる

$1 \rightarrow 0 \rightarrow 1$

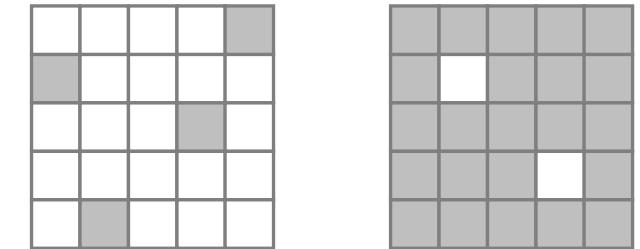
$0 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

HTFuzz : RQ2

RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

ハイパーパラメータの設定

- ・ ヒープ操作シーケンスビットマップのサイズ 2^K
 - ビットマップサイズが $K > 16$ となると
実行オーバーヘッドが急激に増加
(CollAFL[10] でも示される)
 - $K = 16$ と固定して, L の値を変化



HTFuzz : RQ2

RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

ハイパーパラメータの設定

- ・ リングバッファの長さ L

$$1 \rightarrow \emptyset \rightarrow 1$$

$$\emptyset \rightarrow \emptyset \rightarrow 1 \rightarrow \emptyset \rightarrow 1$$

K = 16

L = 3 のとき

最良の結果

Program	AFL		16-16		16-8		16-4		16-3		16-1	
	Den	Bug	Den	Bug	Den	Bug	Den	Bug	Den	Bug	Den	Bug
jpegoptim	0.34	1	0.49	1	0.49	1	0.40	1	0.39	1	0.35	1
lrzip	2.07	2	19.3	2	11.1	3	4.30	2	3.42	8	2.41	2
yasm	13.2	13	100	2	99.9	7	55.1	7	40.1	13	18.0	12
mjs	6.19	13	100	8	95.2	6	34.5	10	20.6	13	8.23	13
binutils	7.59	3	100	3	75.2	3	15.1	3	14.1	2	7.53	2
boolector	7.87	1	65.4	1	19.4	1	11.9	1	9.97	1	8.02	1
gpac	8.64	4	100	3	86.3	3	36.3	4	21.7	3	11.0	3
gpac_latest	13.2	3	100	3	85.7	2	41.2	4	30.5	4	16.5	3
GMagick	9.58	2	8.25	2	9.54	2	9.23	2	9.80	2	9.55	2
IMagick	14.4	1	8.36	0	14.9	0	9.24	0	14.6	1	9.82	0
Avg/Sum	8.30	43	60.2	25	49.8	28	21.7	34	16.5	48	9.14	39

HTFuzz : RQ2

RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

HTFuzz の要素の比較

- HTFuzz は AFL の3つの要素を変更
 - シードの保存戦略
 - シードの選択戦略
 - シードの変異戦略
- それぞれの要素がどれだけファザーの有効性に影響するか？

HTFuzz : RQ2

RQ2 : HTFuzz の要素は HT-vuls の発見の効率にどの程度貢献するか？

HTFuzz の要素の比較

- AFL-S : ヒープシーケンスフィードバックの保存戦略を持つ
- AFL-SP : AFL-S をベースにアクセスされたポインタの選択戦略を追加
- AFL-SM : AFL-S をベースに MOPT の変異戦略を追加

Program	HTFuzz			AFL			AFL-S			AFL-SP			AFL-SM		
	U	L-Cov	HSeq	U	L-Cov	Hseq	U	L-Cov	Hseq	U	L-Cov	Hseq	U	L-Cov	Hseq
Sum/Avg/Avg	74	25.56%	52,149	48	25.49%	28,608	51	25.70%	39,045	61	25.53%	37,055	62	26.41%	52,901

U : 発見した脆弱性数, L-Cov : コードカバレッジ, Hseq : ヒープシーケンス量

コードカバレッジは AFL-SM が高いが, 発見した脆弱性数は HTFuzz がトップ
 → 3つの要素はどれも重要

HTFuzz : RQ3

RQ3 : HTFuzz は他のファザーと比較してどうか？

HT-Vuls に焦点を当てたファザー UAFL, UAFuzz との比較

- UAFL, UAFuzz はオープンソースではない
→ 論文中で示された UAFL, UAFuzz と同じベンチマークに対して
これらのファザーが発見した HT-Vuls を HTFuzz も発見できるかで比較
- HTFuzz は UAFL, UAFuzz が発見したすべての HT-Vuls を発見
→ 論文で書かれた脆弱性のみ
UAFL は論文には書かれていない別の HT-Vuls も発見している [11]
- HTFuzz は UAFL, UAFuzz が見逃した 7 個の HT-Vuls も発見

HTFuzz : RQ3

RQ3 : HTFuzz は他のファザーと比較してどうか？

HT-Vuls に焦点を当てたファザー UAFL, UAFuzz との比較

- ・ HTFuzz は UAFL, UAFuzz が見逃した 7 個の HT-Vuls も発見

→ 見逃された原因是静的解析の限界

- ・ 関数ポインタ
- ・ コールバック
- ・ スレッドインターリーブ

→ HTFuzz では特定の操作が偶然見つかった場合に脆弱性を発見することができる

HTFuzz : RQ3

RQ3 : HTFuzz は他のファザーと比較してどうか？

他のファザーとの比較

- ・発見した HT-Vuls 数, ヒープ操作シーケンス量とともに他のファザーと比較して HTFuzz は有効

発見した HT-Vuls 数 (P-value : Mann-Whitney U-test の p 値)

Program	HTFuzz		AFL		Memlock		AFL-sen-ma		AFL-sen-mw		PathAFL		Tofuzz		MOPT		Angora		Ankou	
	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg
Sum/Sum	74	48.88	48	31.88	27	19	21	13.00	18	12.00	41	34.83	38	26.25	51	41.00	37	19.88	59	40.88
P-value																				

ヒープ操作シーケンス量

Program	HTFuzz	AFL	AFL-sen-ma	AFL-sen-mw	Memlock	PathAFL	Tofuzz	MOPT	Angora	Ankou
Avg	52,149	28,608	19,908	19,609	25,810	23,603	25,260	35,514	17,497	26,339

HTFuzz [3] のまとめ

ジャンル： typestate ガイドファジング

問題提起：

- UAF のような特定の順序で発生する脆弱性はコードカバレッジだけでは効率的に発見できない
- 従来の typestate ファザーは静的解析・事前情報に依存

提案手法：

- 実行時にメモリアクセスを追跡し、ヒープ操作シーケンスの多様性を高める
- 実行時にアクセスされるポインタの数を計測

結果：

- 従来のファザーより多くのヒープ操作シーケンスを発見
- 37件の新たな脆弱性を発見（うち 32件はヒープの時間的脆弱性）