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）

[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]

目次

[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？

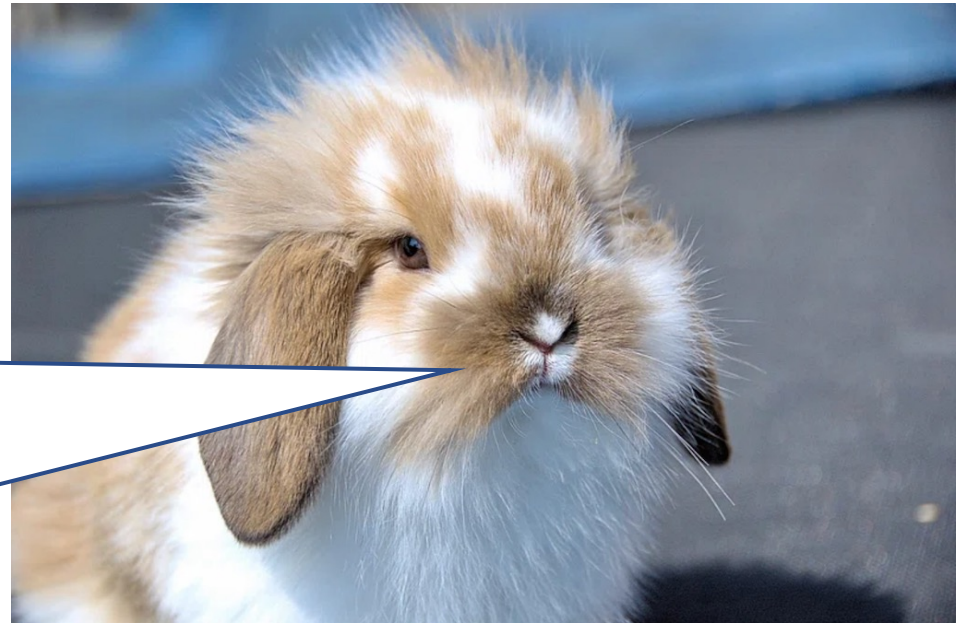
AFL（American Fuzzy Lop）[1]

・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]

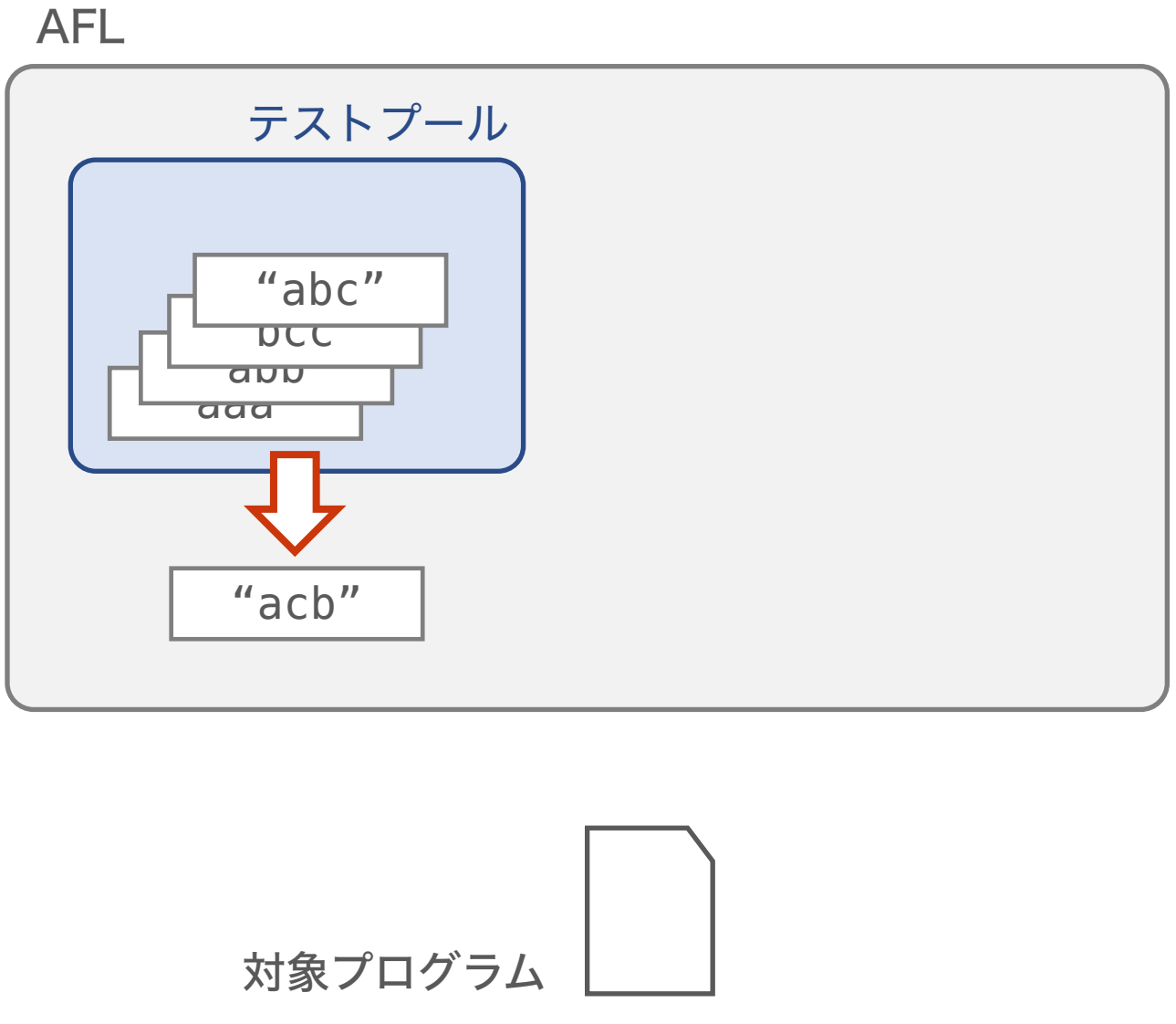


[1] https://github.com/google/AFL

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

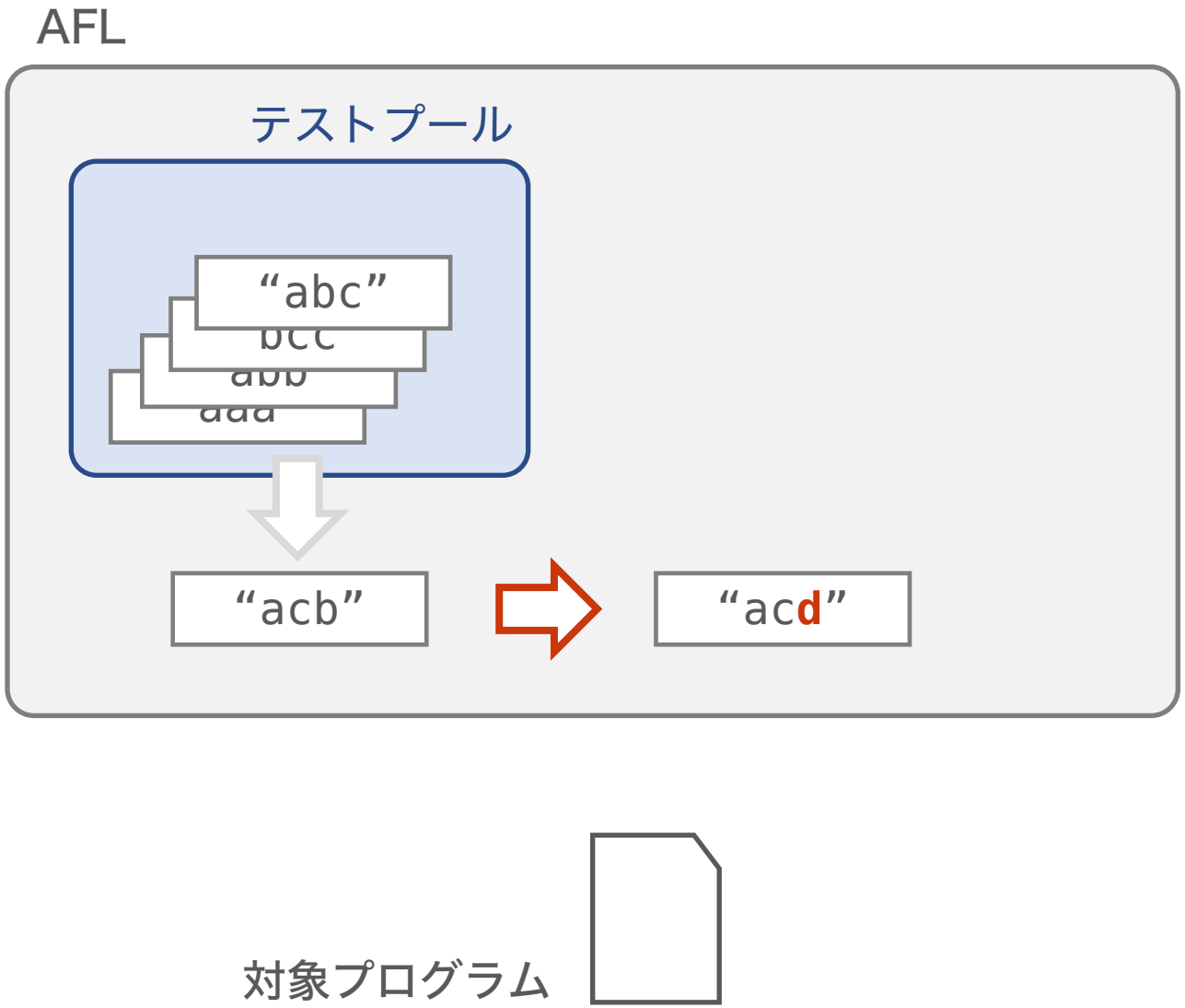
AFL の流れ

1. テストプールからシードを選択



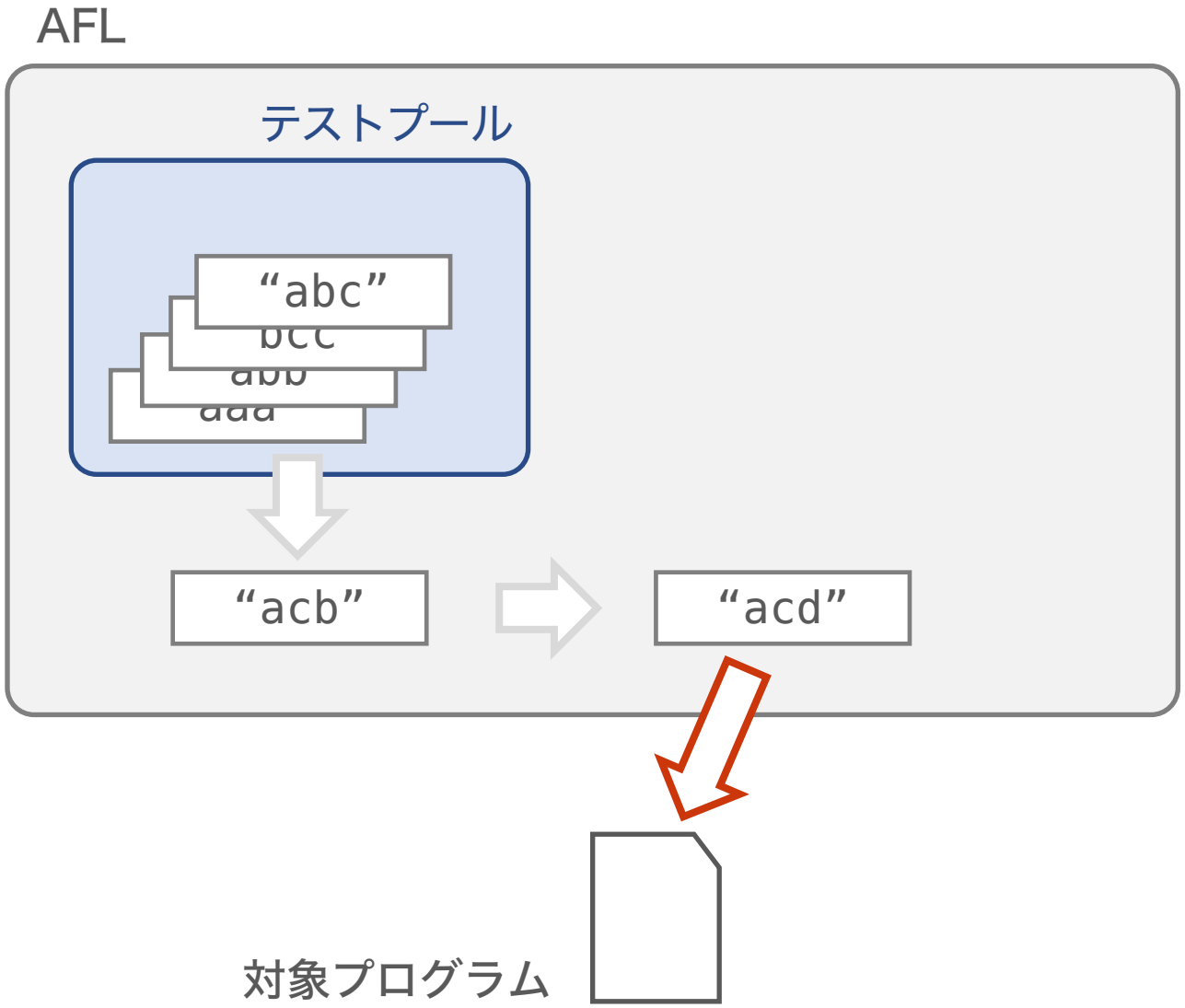
AFL の流れ

1. テストプールからシードを選択
2. シードを変異



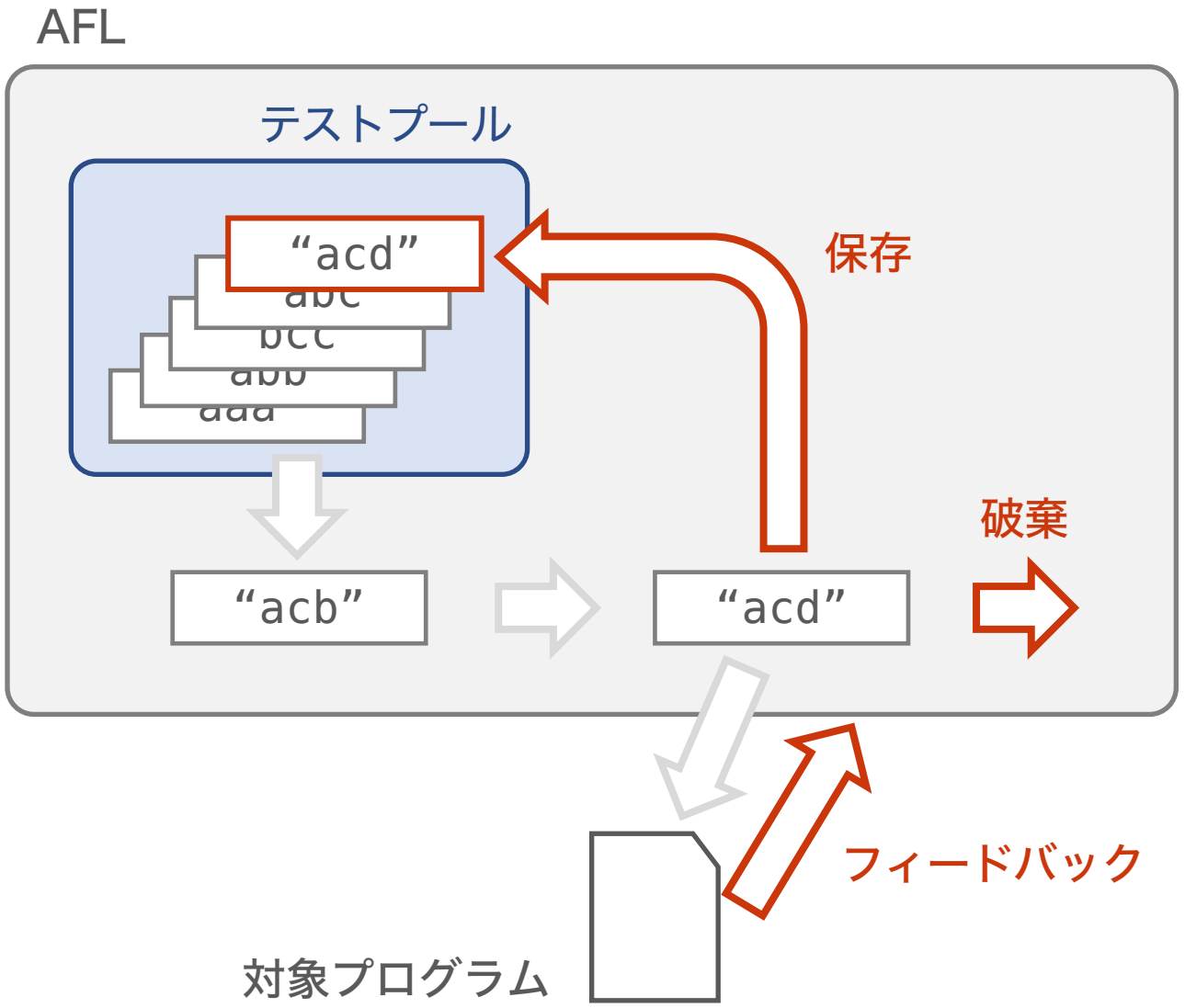
AFL の流れ

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



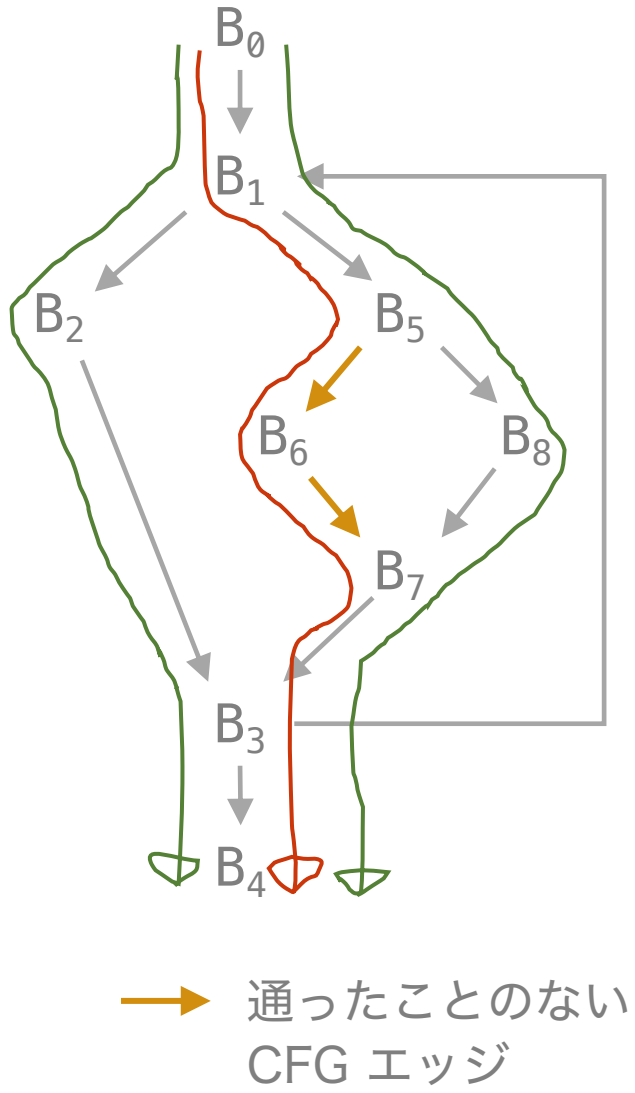
AFL の流れ

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)



AFL のフィードバック

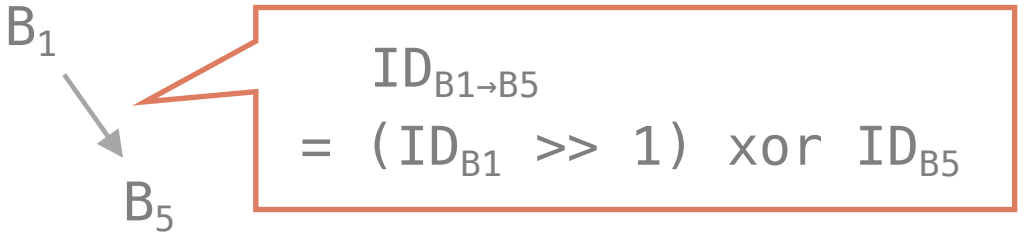
- 共有メモリ shared_mem を用いて
  CFG エッジを通ったか測定
- ランダムに基本ブロックに ID を割り当て
- CFG エッジの ID は以下のように計算

$$ID_{A \to B} = (ID_A >> 1) \text{ xor } ID_B$$

  → ID_A を右シフトするのは
  $ID_{A \to B}$ と $ID_{B \to A}$ を区別するため

- CFG エッジ A→B を通ったとき

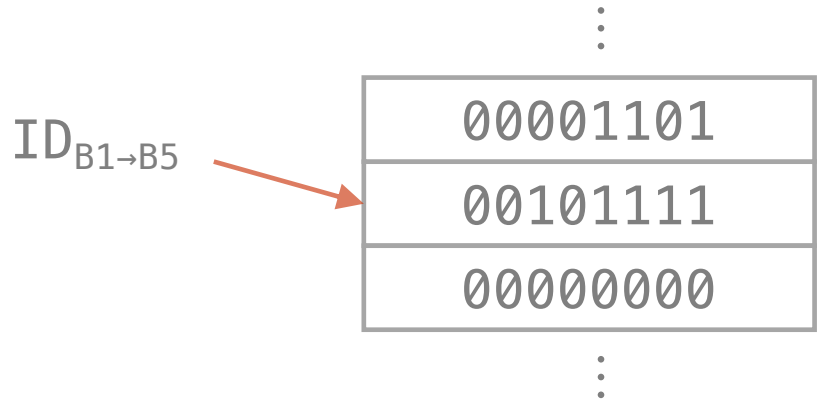
$$shared\_mem[ID_{A \to B}]++$$



- 各実行で通過したとき

$$shared\_mem[ID_{B1 \to B5}]++$$

- 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
- CFG エッジの ID は以下のように計算

$$ID_{A \to B} = (ID_A >> 1) \text{ xor } ID_B$$

- 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

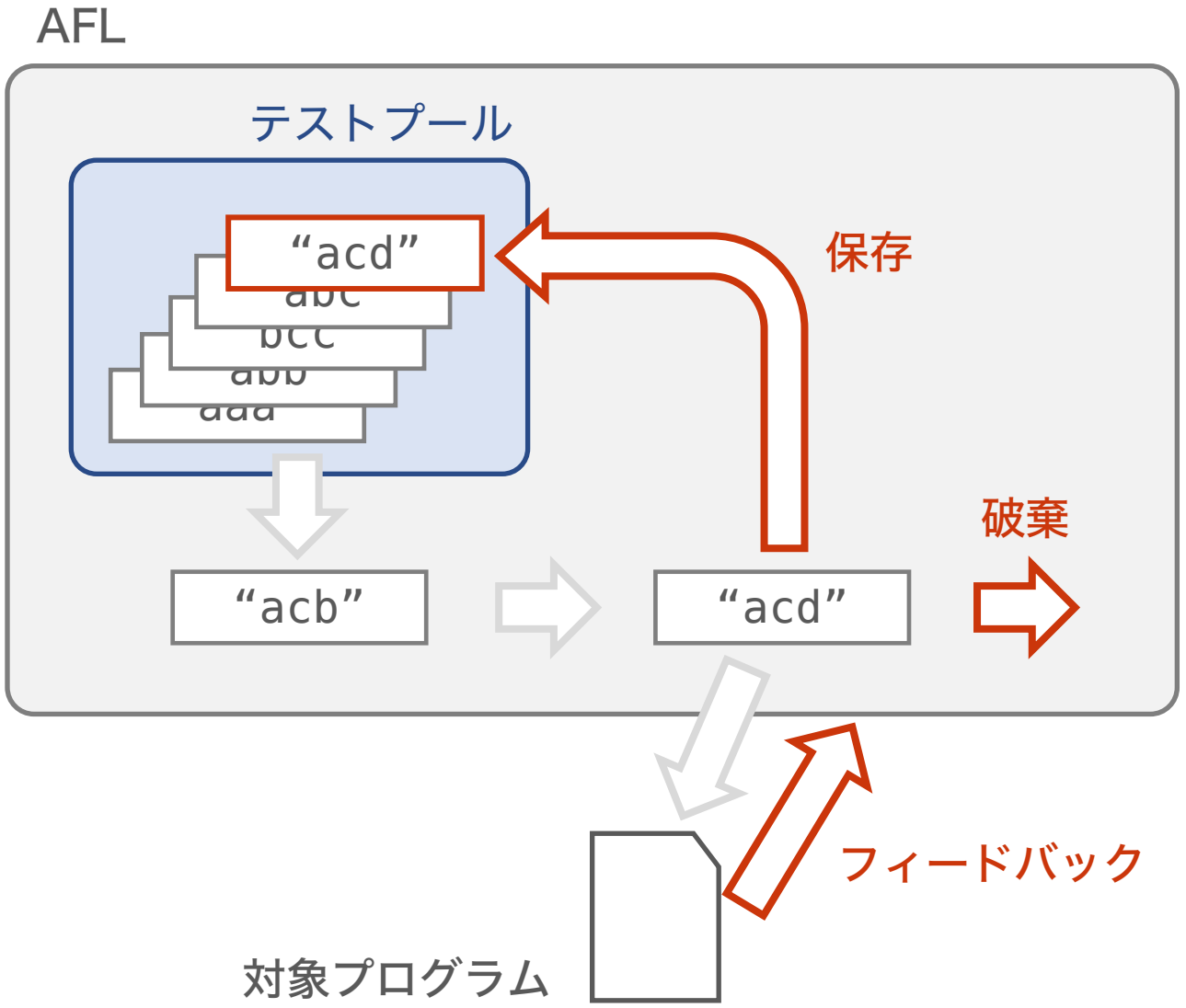
Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	-

分岐の数と ID の衝突確率 [4]

[4] https://github.com/google/AFL/blob/master/docs/technical_details.txt

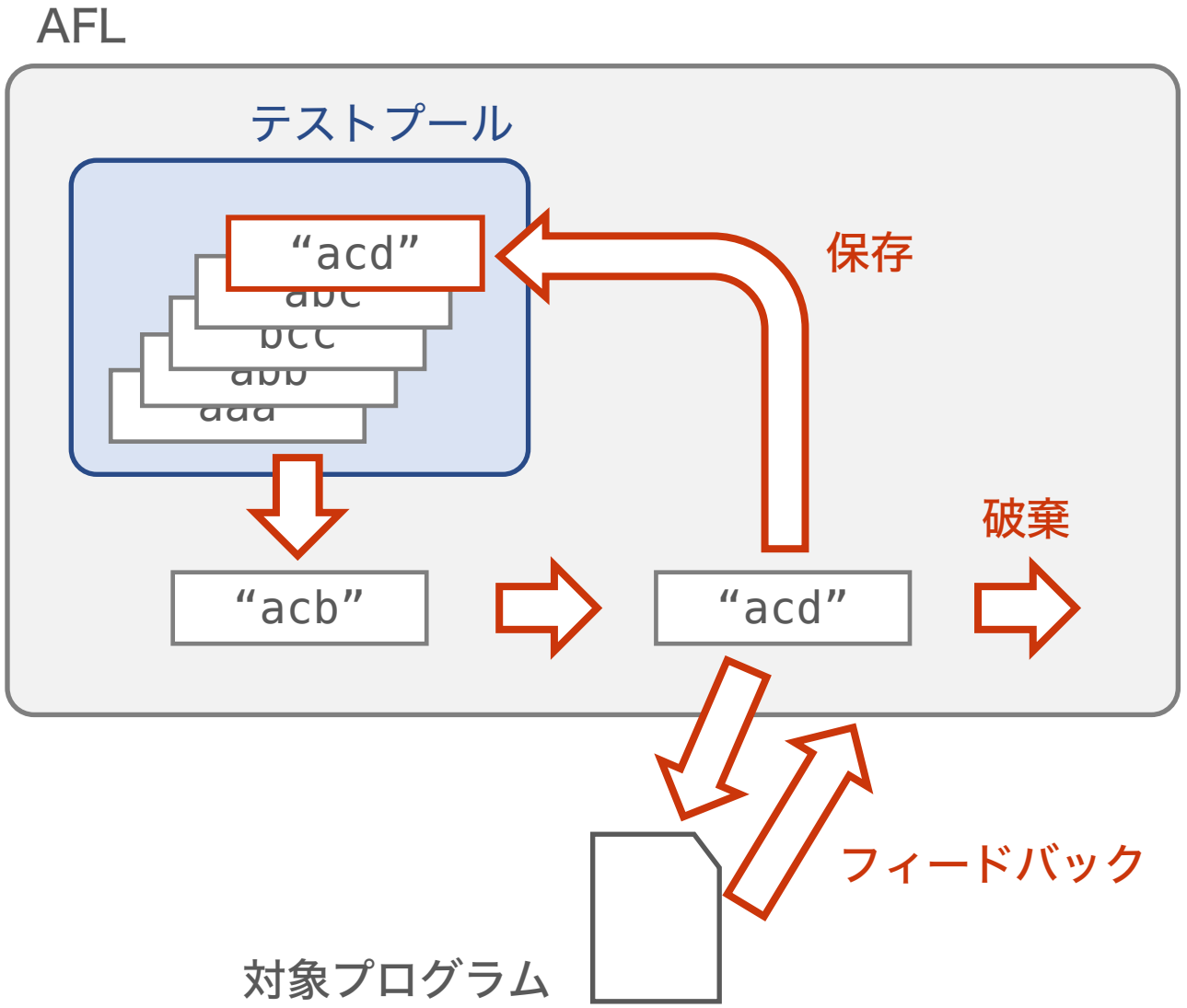
AFL の流れ

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

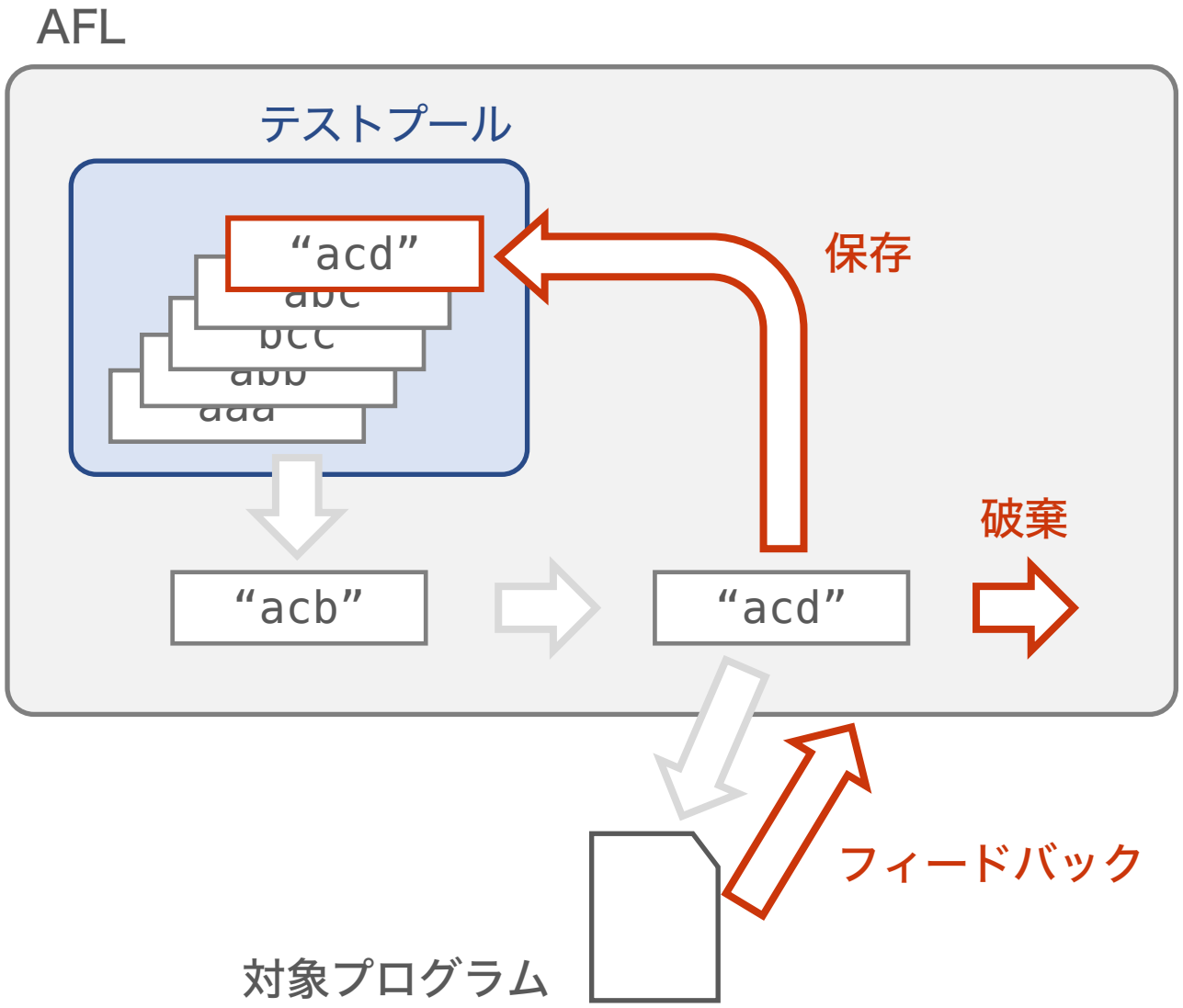
- シードの **保存** 戦略
- シードの **選択** 戦略
- シードの **変異** 戦略

(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？

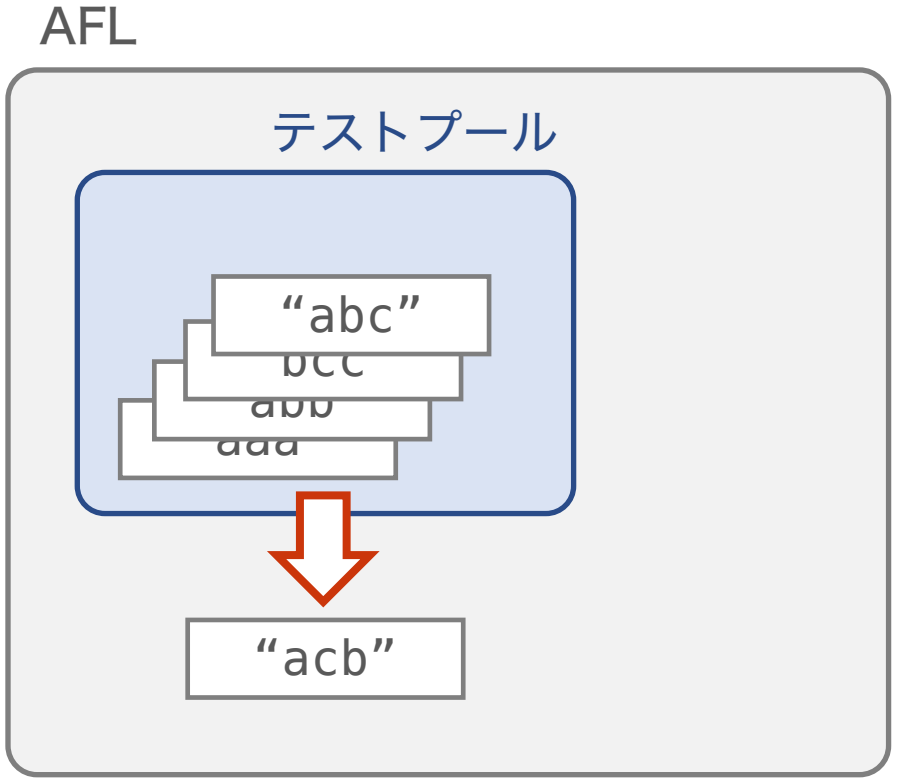
→ AFL は新しい CFG エッジを通ったテストケースを興味深いものとして保存する



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

シードの変異戦略

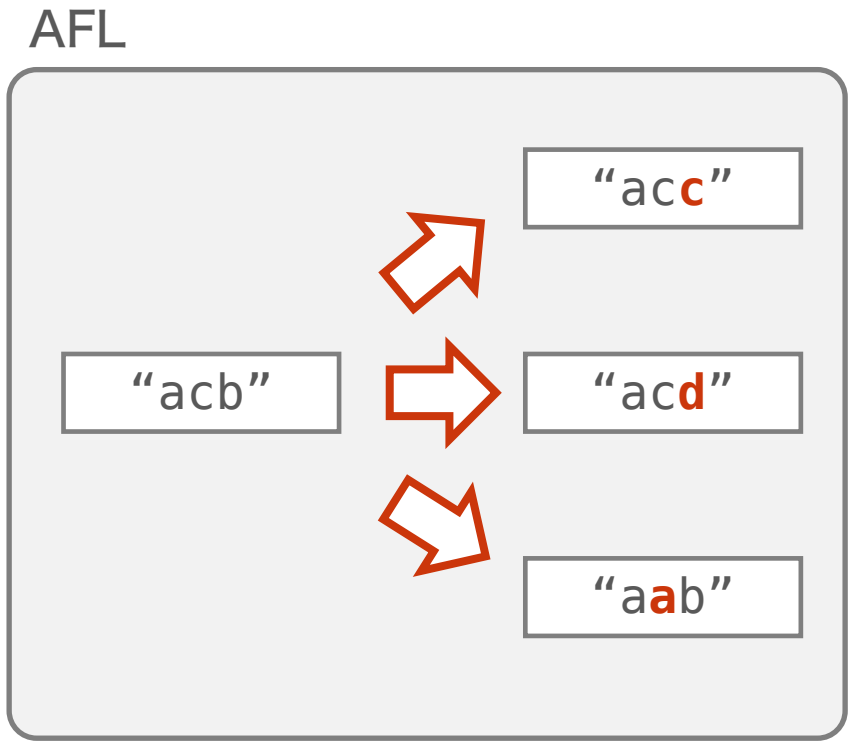
・シードを何回変異させるか？（power scheduling）

→ 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

目次

[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ない

- 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 → 7 → 10 → 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 }
```

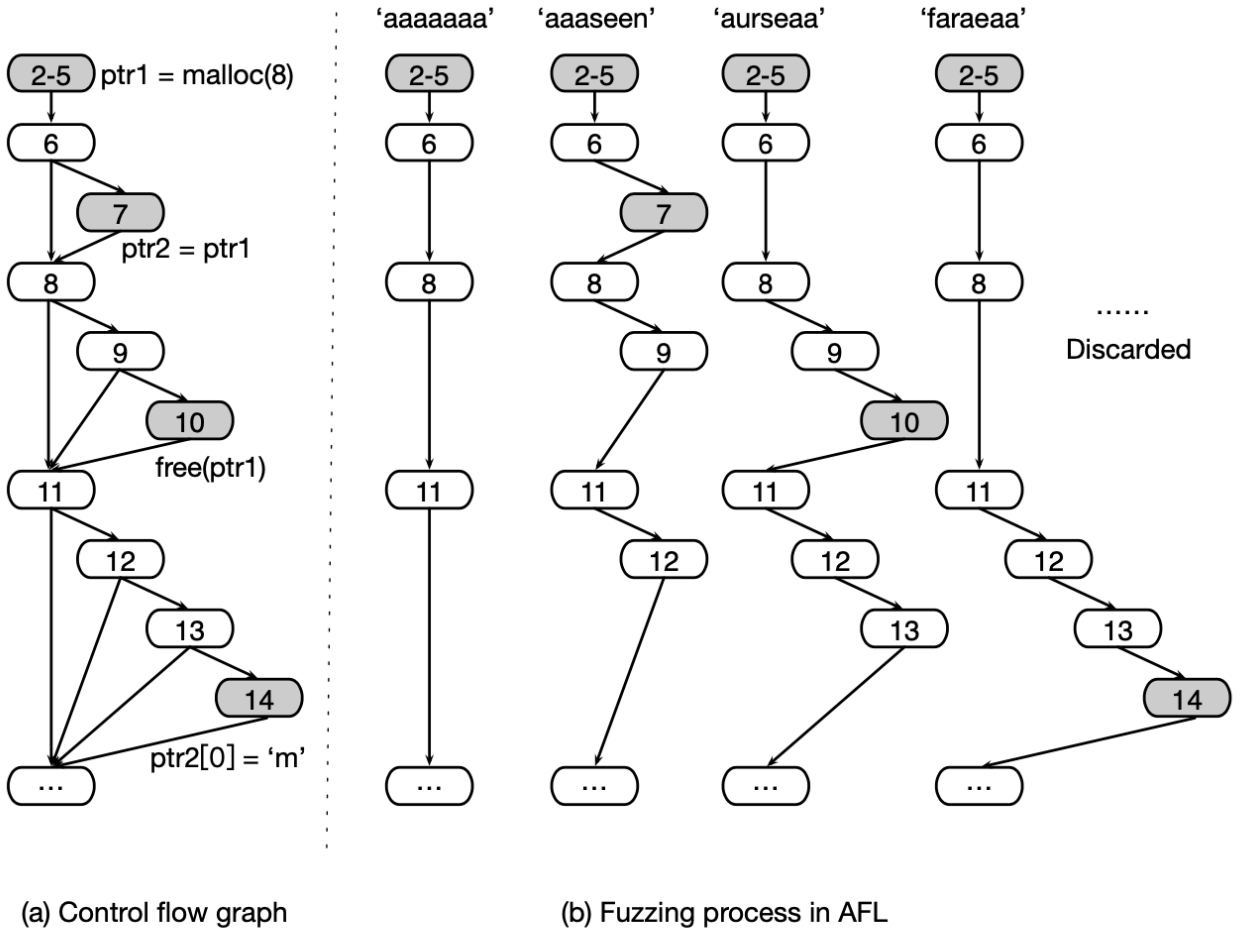
AFL の例

・シードに対しての変異が 6 → 7, 9 → 10, 13 → 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 → 7 → 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 → 7 → 10 → 14 を通る
テストケースを生成するのは困難



目次

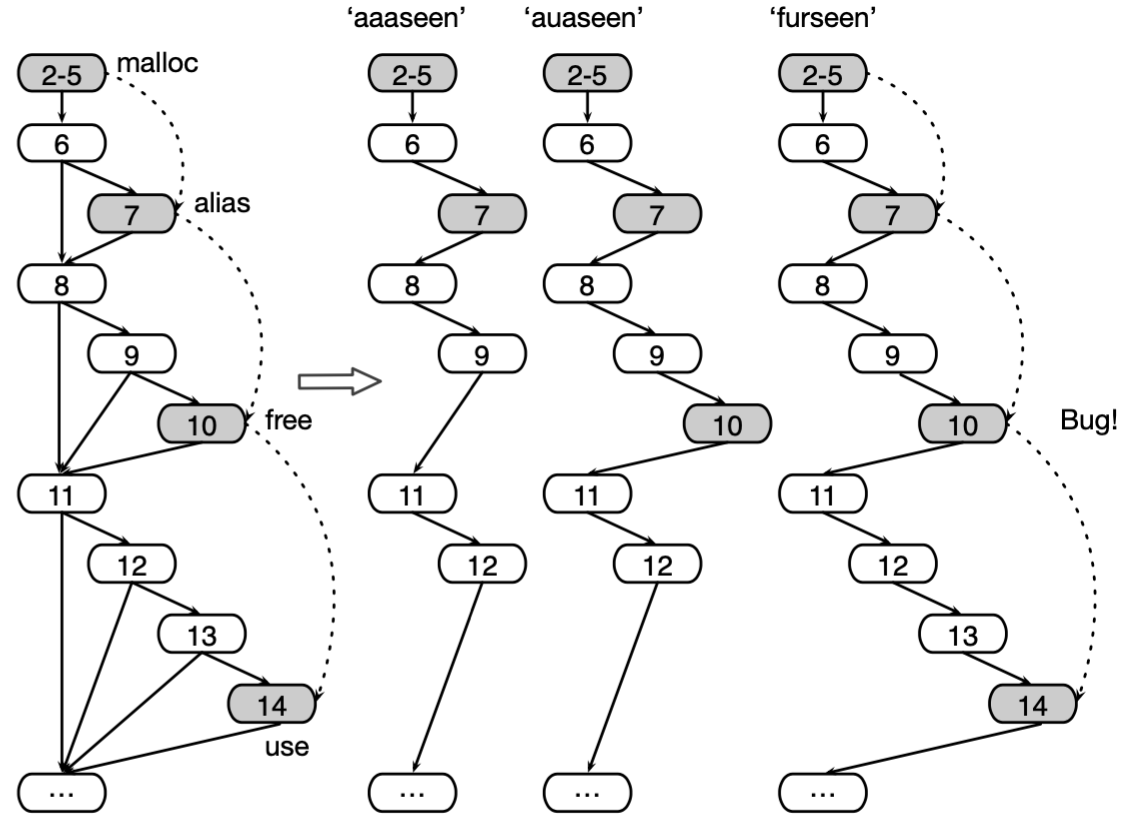
[1] https://github.com/google/AFL

[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 → 7 → 10 → 14 で発生する
- AFL で破棄された 4 → 7 → 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 → 7 → 10 → 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

UAFL [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`

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

UAFL の2つのステップ

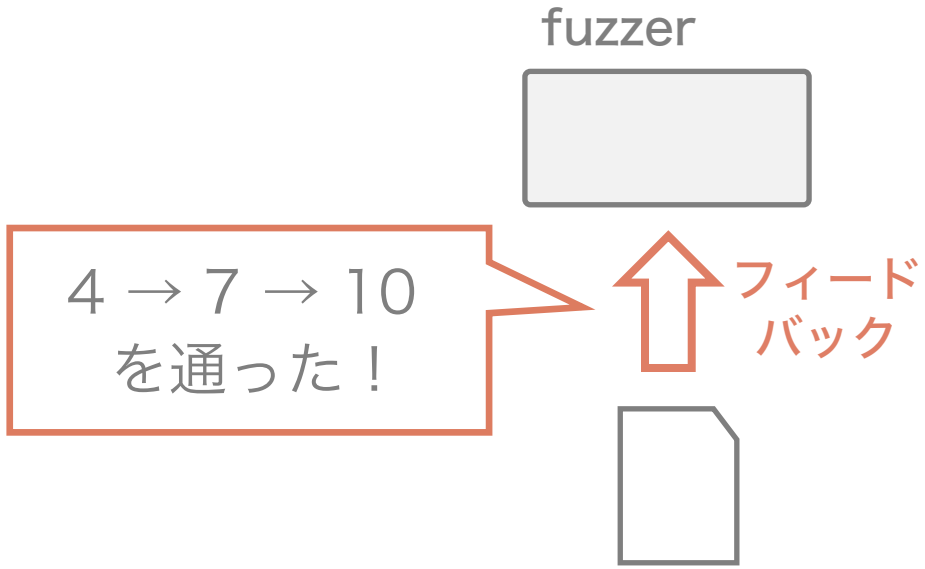
1. 静的 typestate 解析

   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 → 7 → 10 → 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 → 7 → 10 → 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,
  - `malloc` → `free` → `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 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
8 **return** S;

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 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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7行目)

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

$s_m \to s_f \to s_u$ 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 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u\rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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

4 → 10 → 14 (エイリアス文も含めた
4 → 7 → 10 → 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

$S \leftarrow \emptyset$;
$(S_M, M) \leftarrow find\_malloc(P)$;
foreach $(s_m, m) \in (S_M, M)$ **do**
  $A \leftarrow cal\_alias(m)$;
  $S_F \leftarrow find\_free(A, P)$;
  $S_U \leftarrow find\_use(A, P)$;
  $S \leftarrow S \cup \{\langle s_m, s_f, s_u \rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
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

・ここでの到達可能性の判定は path-insensitive に行う

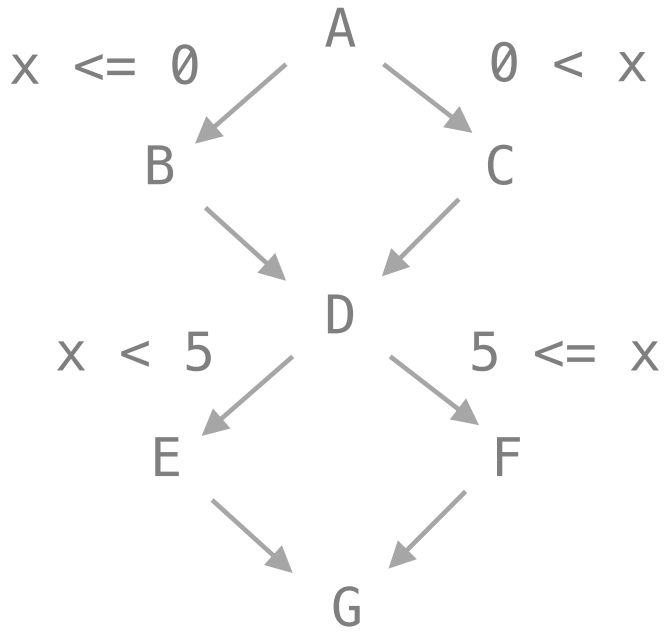
path-insensitive : パスの条件を無視した解析（↔ path-sensitive）

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

・ここでの到達可能性の判定は path-insensitive に行う

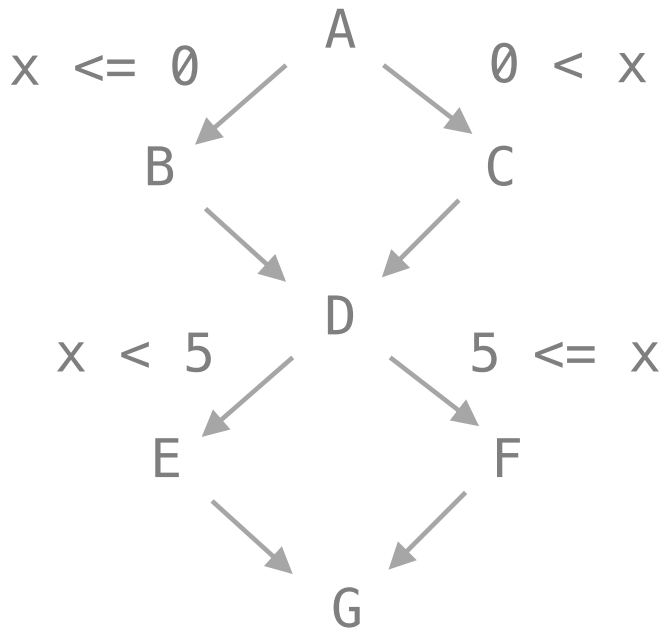
path-insensitive : パスの条件を無視した解析（↔ path-sensitive）

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 に行う

path-insensitive : パスの条件を無視した解析（↔ 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

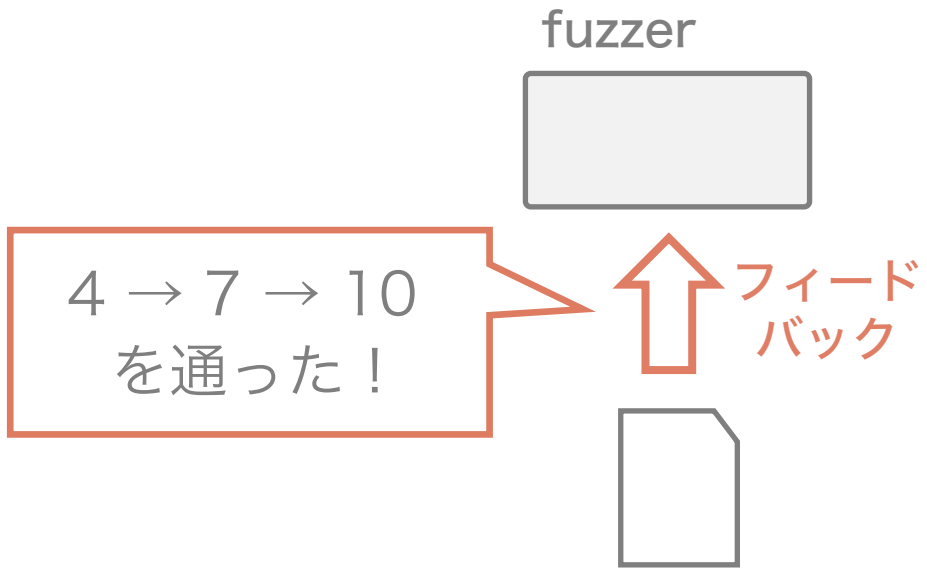
→ (無駄な操作シーケンスのガイドにファジングの多くの時間を費やす？)

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 → 7 → 10 → 14

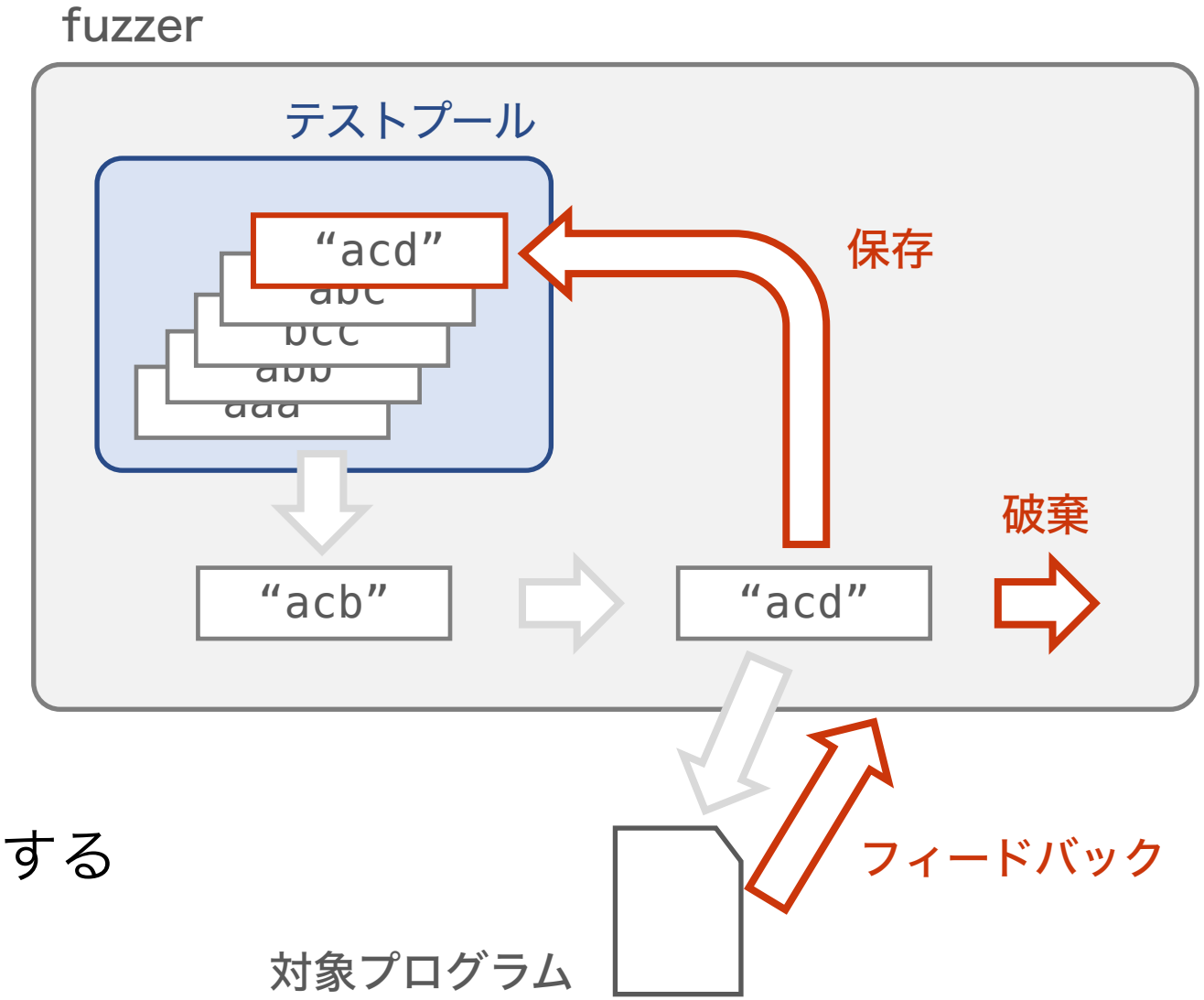


UAFL : シードの保存戦略

・4 → 7 → 10 → 14 という
操作シーケンスが与えられたとき

- ・4 → 7
- ・4 → 7 → 10
- ・4 → 7 → 10 → 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

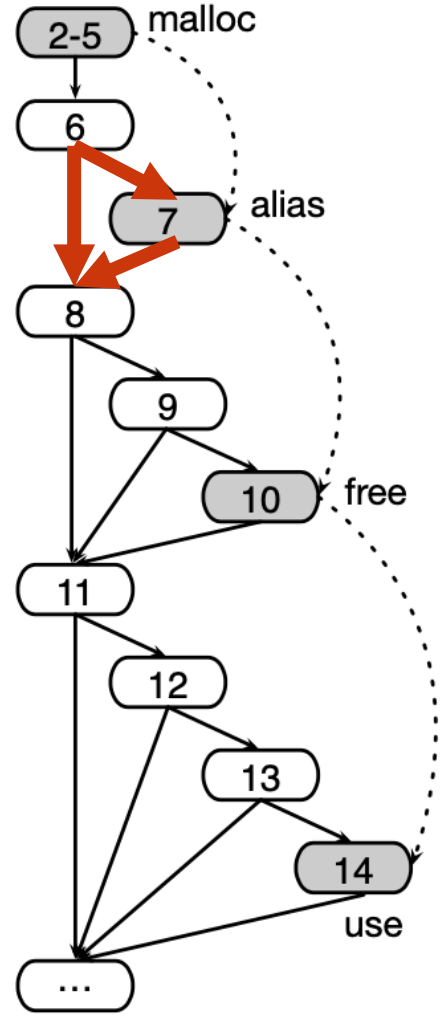
・操作シーケンス：4 → 7 → 10 → 14

・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

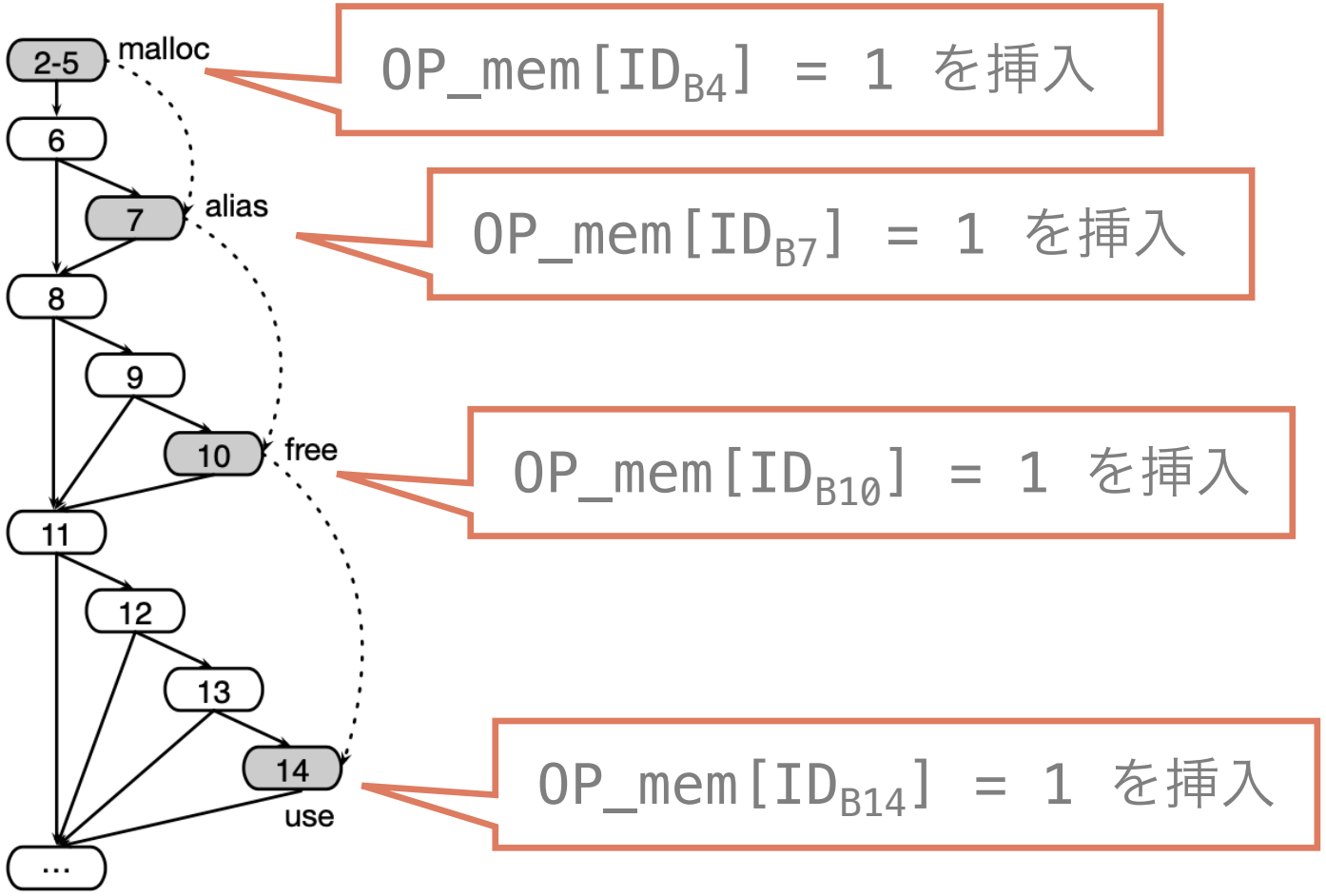
OP_mem[]：操作シーケンス内の基本ブロックを
通ったか記録

OPE_mem[]：操作シーケンスエッジをカバーしたか
記録



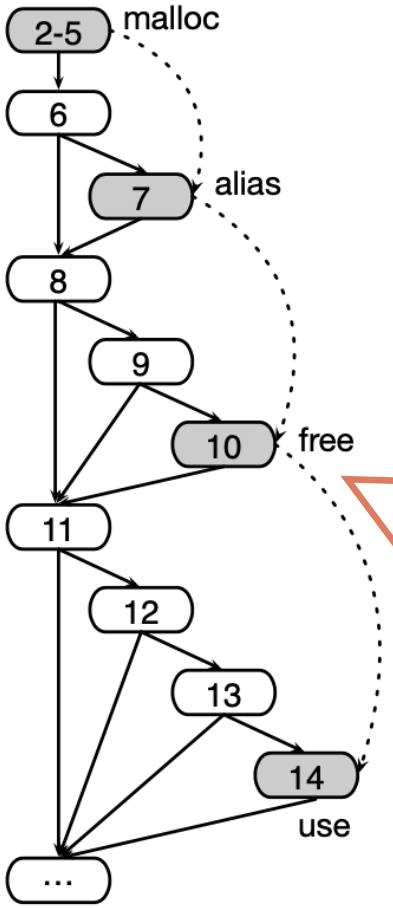
UAFL：シードの保存戦略

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



B_{10} を通ったとき
B_4, B_7 を通ったかどうかで場合分け

B_7 を通ったとき（OP_mem[ID_{B7}] == 1）

OPE_mem[ID_{B4} xor ID_{B7} xor ID_{B10}]++

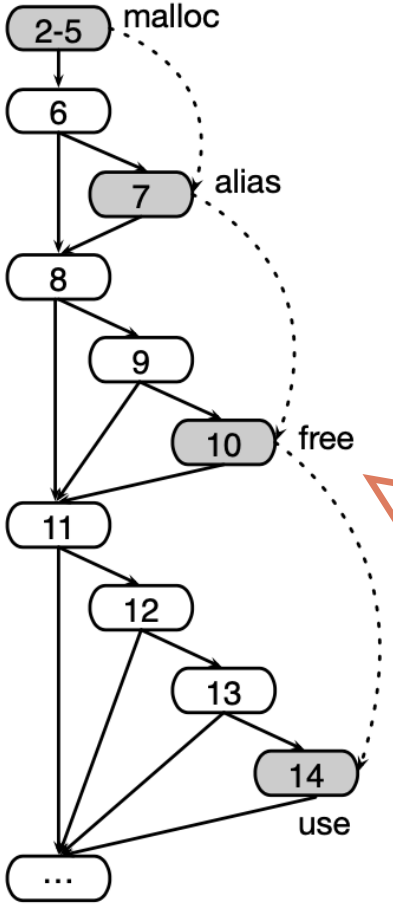
B_7 を通っていないとき（OP_mem[ID_{B7}] != 1）

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

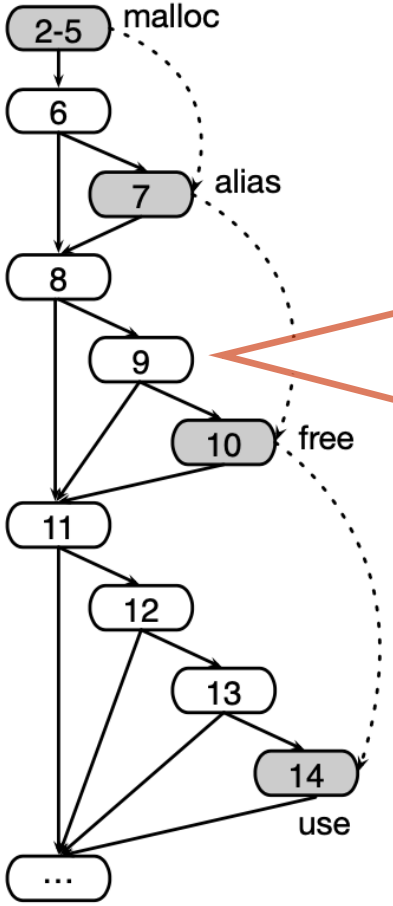


操作シーケンス $B_0 \rightarrow \cdots \rightarrow B_i \rightarrow \cdots \rightarrow B_n$
が与えられたとき, B_i に以下を挿入

```
tID = 0
for j in range(0, i):
  if OP_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 → 10 の操作シーケンス
エッジをカバーするには, 8 → 9 を
カバーする必要あり
（直接 7 → 10 をカバーするのは難しい）

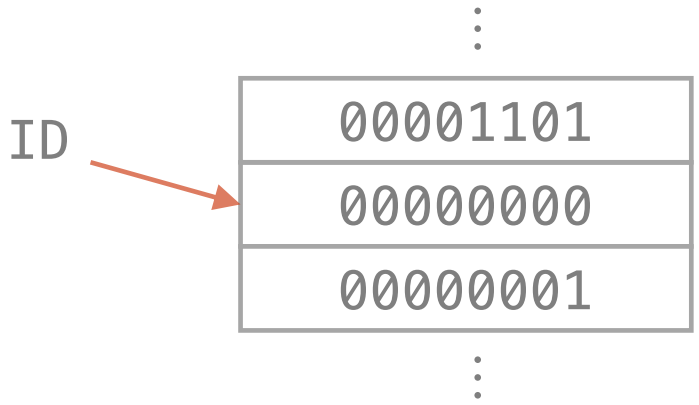
→ B_9 にも同様の計装を行う

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 :

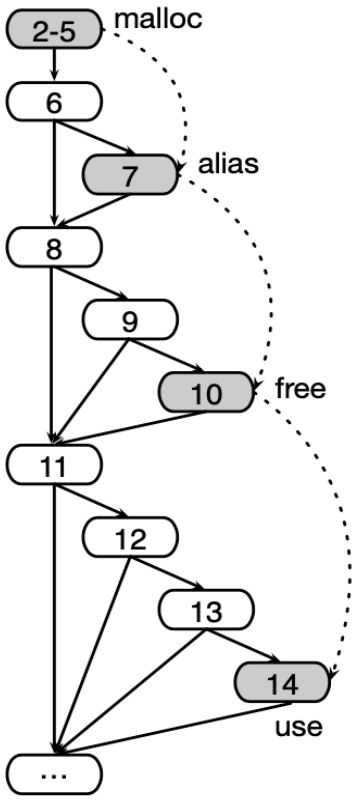
ファジングテスト全体のカバレッジ



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



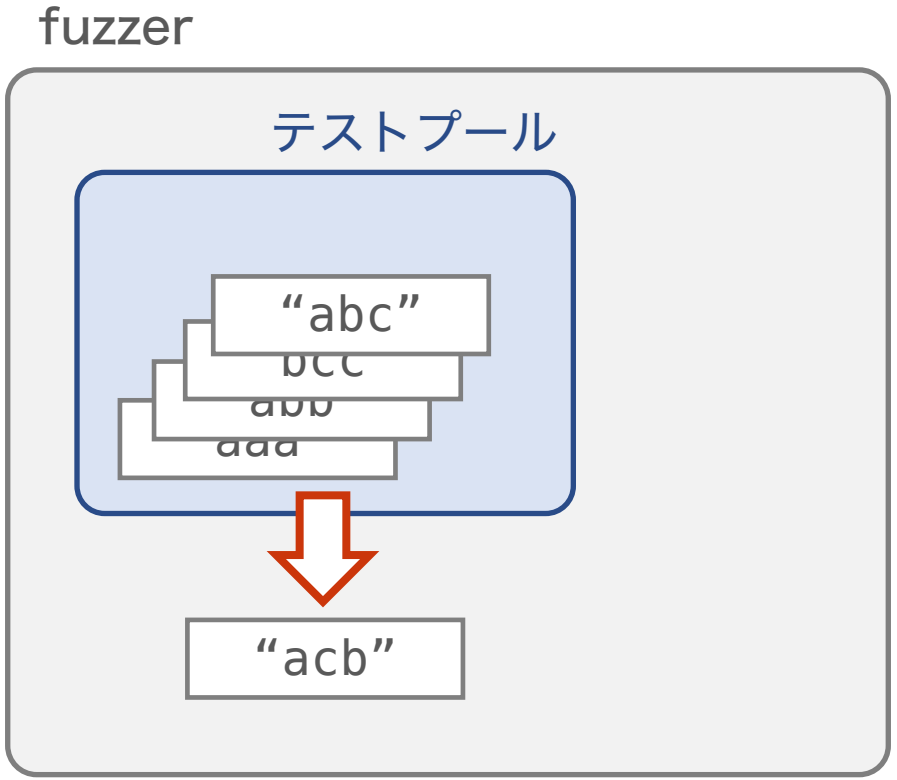
操作シーケンス：4 → 7 → 10 → 14

新しく

- 4 → 7 → 9
- 4 → 7 → 10

をカバーしたテストケースは保存

⇨ 4 → 7 → 10
のテストケースを優先



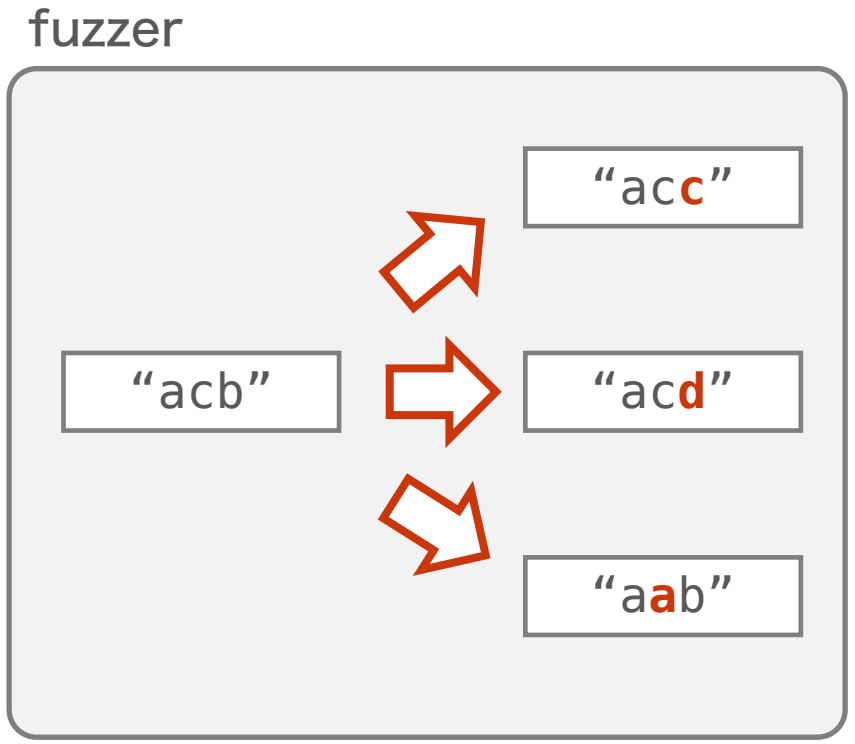
UAFL：シードの変異戦略

・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



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 → 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 → 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 }
```

[5] Measuring the strength of information flows in programs [TOSEM '09]

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

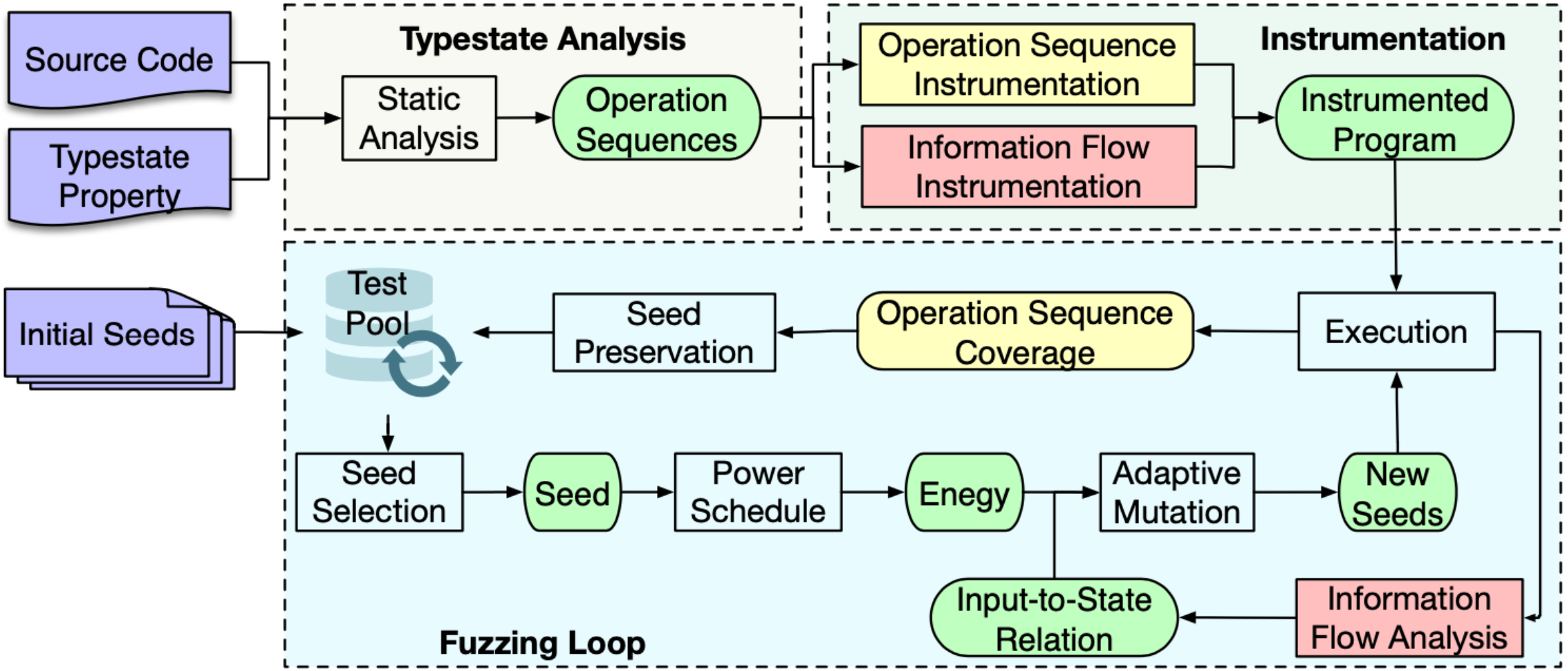
・3バイト目：3.3

・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


Source Code
Typestate Property
Typestate Analysis
Static Analysis
Operation Sequences
Operation Sequence Instrumentation
Information Flow Instrumentation
Instrumentation
Instrumented Program
Initial Seeds
Test Pool
Seed Preservation
Operation Sequence Coverage
Execution
Seed Selection
Seed
Power Schedule
Enegy
Adaptive Mutation
New Seeds
Input-to-State Relation
Information Flow Analysis
Fuzzing Loop

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]