論文紹介

DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing

[Usenix-Security'23]

DDRace [1] の概要

ジャンル： Linux デバイスドライバファジング

問題提起：
- 従来のスレッド間探索ファザーは主にデータ競合が対象
- Directed ファジングはスレッドのインターリーブを無視する

提案手法： Linux ドライバの並行 UAF を発見するための並行 directed ファザー

1. 事前解析によって, ファジングの探索空間を削減
2. 新しいフィードバックによって並行 UAF 脆弱性と
   スレッドインターリーブを効果的に探索
3. テストケースの再現性を高めるためにスナップショットを活用

結果： Linux ドライバの4つの未知の脆弱性と8つの既知の脆弱性を発見

[1] DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing [Usenix-Security'23]

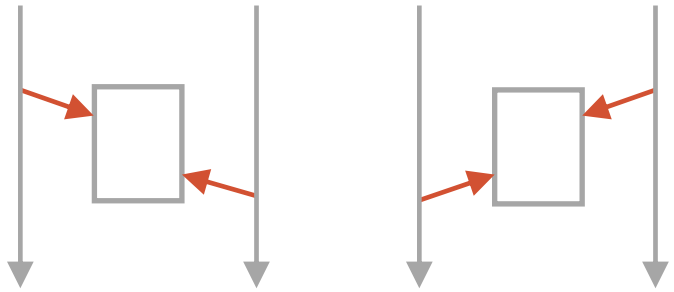
背景知識：並行性バグ

・複数のスレッドやプロセスで並行して実行される際に発生するバグ

　データ競合, デッドロック, アトミック違反 など

データ競合（data race）

1. 複数のスレッド間で共有される変数に対して,
2. 複数スレッドが同時にアクセスし,
3. 一つ以上が書き込み命令のときに発生する



```
1  void thread1() {
2    A = 10;
3    read(A);
4  }
```

```
5  void thread2() {
6    A = 20;
7  }
```

Q. 3行目での A の値は？

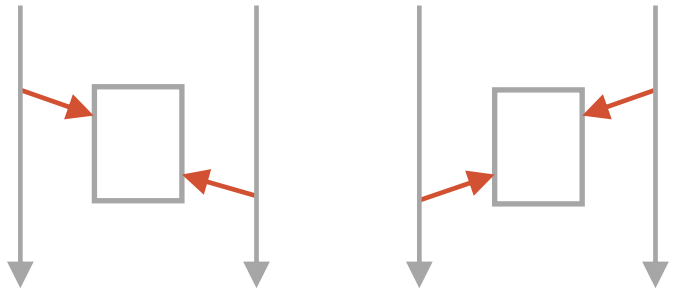
背景知識：並行性バグ

・複数のスレッドやプロセスで並行して実行される際に発生するバグ

　データ競合, デッドロック, アトミック違反 など

データ競合（data race）

1. 複数のスレッド間で共有される変数に対して,
2. 複数スレッドが同時にアクセスし,
3. 一つ以上が書き込み命令のときに発生する



```
1  void thread1() {
2    A = 10;
3    read(A);
4  }
```

```
5  void thread2() {
6    A = 20;
7  }
```

Q. 3行目での A の値は？

A. 10 or 20

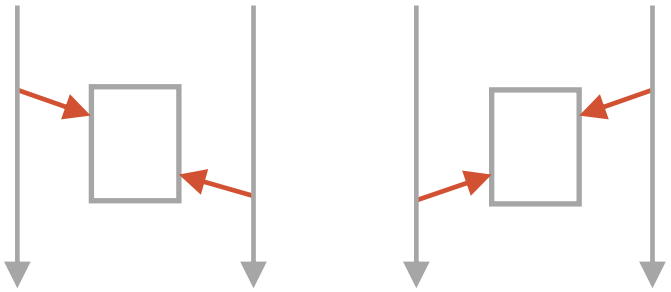
背景知識：並行性バグ

・複数のスレッドやプロセスで並行して実行される際に発生するバグ

　データ競合, デッドロック, アトミック違反 など

<u>データ競合（data race）</u>

1. 複数のスレッド間で共有される変数に対して,
2. 複数スレッドが同時にアクセスし,
3. 一つ以上が書き込み命令のときに発生する



→ スレッドの実行順に依存して,
　プログラムの実行結果が予想のつかないものとなる

背景知識：並行脆弱性

・並行性バグによって脆弱性が導入される可能性がある

例) 2 → 6 → 3 で実行されると UAF

```
1  void thread1() {
2    if (p != NULL)
3      read(*p);
4  }
```

```
5  void thread2() {
6    free(p);
7    p = NULL;
8  }
```

・DDRace の目的は並行処理に関連した UAF を発見すること

背景知識：並行カーネルファジング

・並行性バグの発見を目的としたカーネルファザー

例) Razzer [4], Krace [5]

・スレッドのインターリーブ空間も探索する

・スレッドスケジューリング機構の修正

・スレッドインターリーブフィードバック

・これらのファザーの対象は並行 UAF ではなくデータ競合

・スレッドのインターリーブ空間は非常に広いため,
探索空間を削減することが重要

[4] Razzer: Finding Kernel Race Bugs through Fuzzing [SP'19]
[5] Krace: Data Race Fuzzing for Kernel File Systems [SP'20]

背景知識：Directed ファジング [2]

- ターゲットのプログラム位置に到達する入力を効率的に発見する

  例) 8行目に到達する入力は？

- 使用例
  - パッチテスト
  - クラッシュの再現
  - 静的解析レポートの検証

```
1  char buf[8];
2  input(buf);
3  if (buf[0] == 'h') {
4    if (buf[4] == 'o') {
5      if (buf[2] == 'l') {
6        if (buf[1] == 'e') {
7          if (buf[3] == 'l') {
8            print("hello!");
9          }
10       }
11     }
12   }
13 }
```

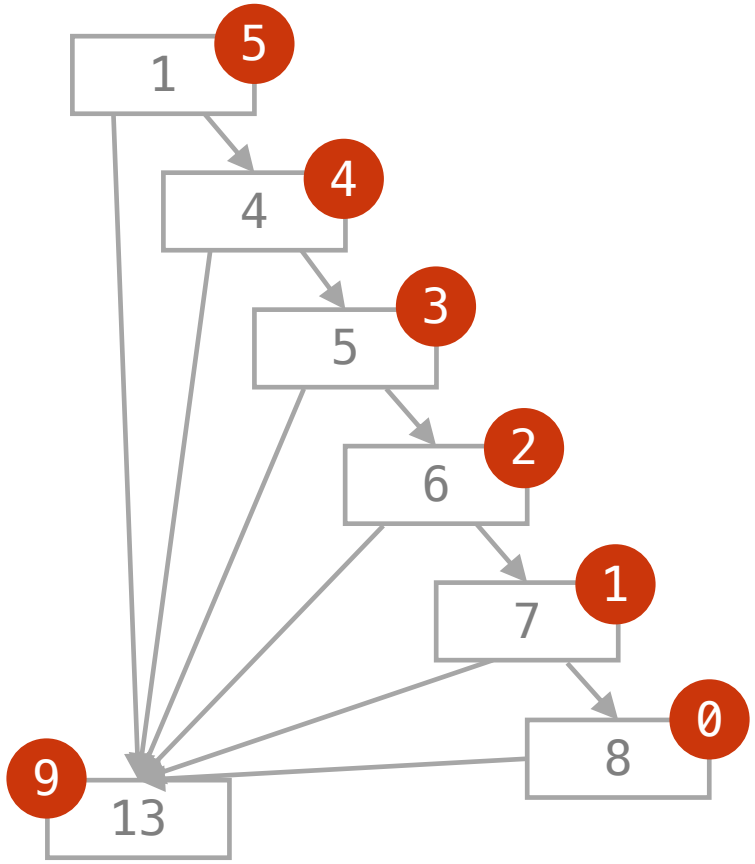
[2] Directed Greybox Fuzzing [CCS'17]

背景知識：Directed ファジング [2]

・Directed ファジングはターゲットからの
距離をフィードバックとして使用する

→ 距離が近いシードを優先する

例) CFG 上の距離




```
1  char buf[8];
2  input(buf);
3  if (buf[0] == 'h') {
4    if (buf[4] == 'o') {
5      if (buf[2] == 'l') {
6        if (buf[1] == 'e') {
7          if (buf[3] == 'l') {
8            print("hello!");
9          }
10       }
11     }
12   }
13 }
```

[2] Directed Greybox Fuzzing [CCS'17]

背景知識：Directed ファジング [2]

・従来の directed ファジングはターゲットサイトまでの
制御フロー距離・データフロー距離のみを考慮する

・スレッドのインターリーブは考慮しない

→ 同じデータでスレッドのインターリーブが異なるテストケースは無視される

[2] Directed Greybox Fuzzing [CCS'17]

背景知識：Linux Drivers

- Linux カーネルはドライバを通じてハードウェアデバイスと相互作用する
- Linux デバイスドライバは特定のドライバインターフェスを実装する必要がある

  特定のシステムコールが呼ばれた時のエントリポイント

- Linux ドライバにおける UAF の大部分は並行性を含む [3]

[3] Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers [Usenix-ATC'19]

Motivating Example

- tty ドライバにおける並行 UAF 脆弱性（CVE-2020-25656）
- 2 → 13 → 15 → 6 の順で実行されると UAF

  → このような並行 UAF 脆弱性を発見するのは難しい

Thread1

```
1  // ioctl$KDGKBSENT
2  p = func_table[i];
3
4  ...
5
6  ... = *p
```

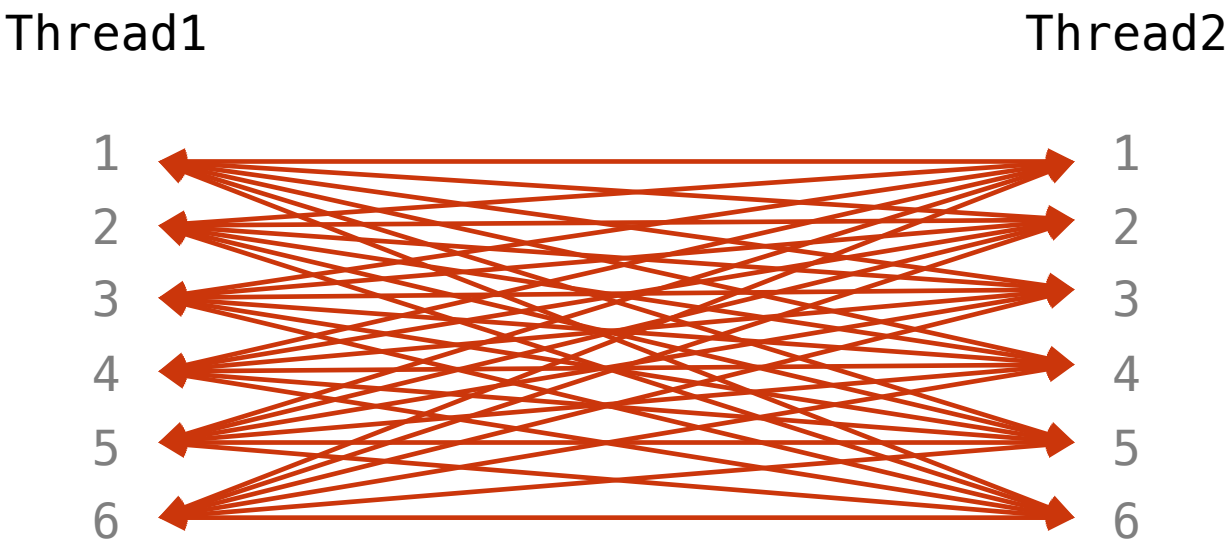
Thread2

```
11 // ioctl$KDSKBSENT
12 // oldptr holds the old func_table[i]
13 func_table[i] = ...;
14 ...
15 kfree(oldptr);
```

並行 UAF 発見の課題 1

・スレッドインターリーブの探索空間は非常に大きい

特に Linux カーネルは大規模で複雑



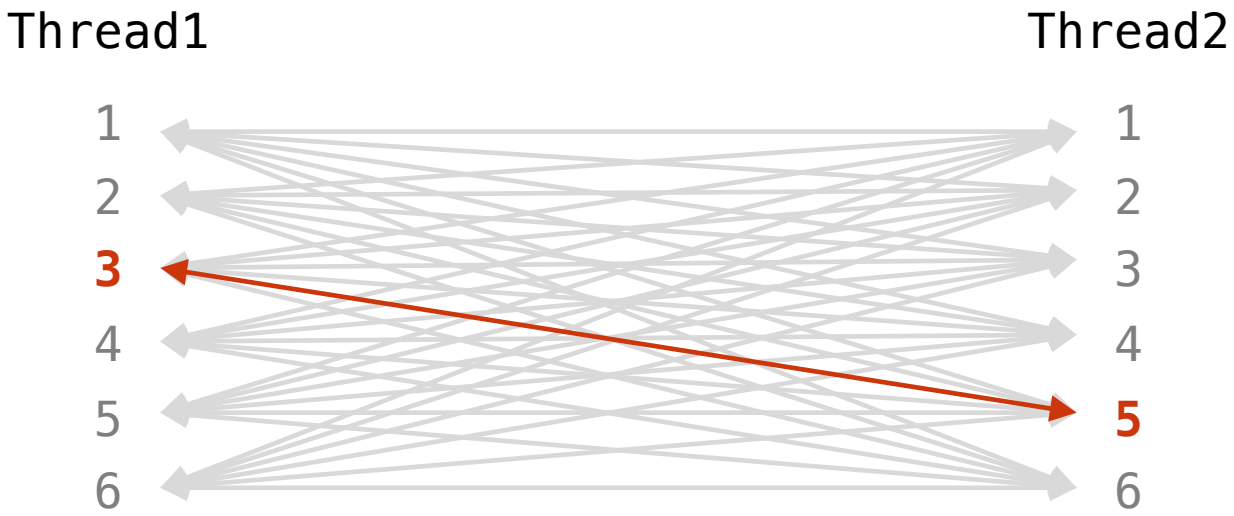
並行 UAF 発見の課題１の解決法

- 事前にカーネルファザーを動かし, 実行トレースから並行 UAF の候補となる `free` と `use` 命令のペアを特定する

  → Directed ファジングを用いて, これらの命令に入力を導く

- 静的解析を用いて, 並行 UAF に関連するレースペアを求める

  → DDRace はこのレースペア間のスレッドインターリーブのみを探索する



並行 UAF 発見の課題２

- Directed ファジングを用いても, 並行 UAF を引き起こすための制約をすべて満たすことは難しい
  - 制御フロー制約 (free と use 命令を実行できるか？)
  - データフロー制約 (free と use 命令は同じメモリを触るか？)
  - スレッドのインターリーブ
- 従来の directed ファザーは CFG 上でのターゲットまでの距離を短縮することを目指す

  → スレッドインターリーブは考慮しない

並行 UAF 発見の課題２の解決法

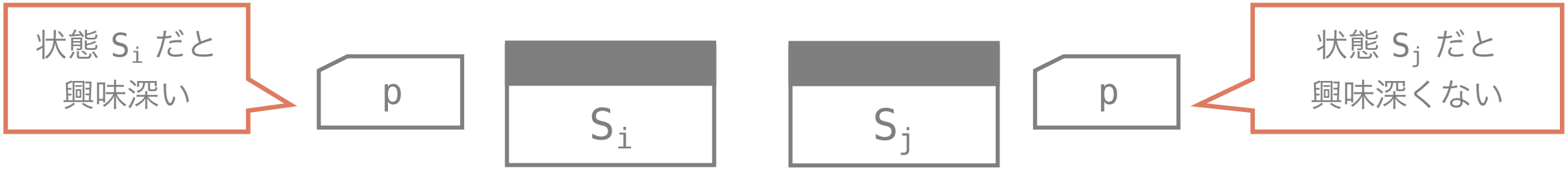
- UAF 脆弱性に焦点を当てた新たな距離を設計する
  - ドミネータ深さ距離（dominator depth distance）

    CFG 上の距離ではなく, 支配木上での距離
  - 脆弱性モデル制約距離（vulnerability model constraint distance）

    UAF が発生する制約を満たした時に距離が縮まる
- 並行実行のフィードバックを導入する
  - 事前に求めたレースペアの read/write 命令の順序を観察し,
    新しいスレッドインターリーブがあるかを確認する

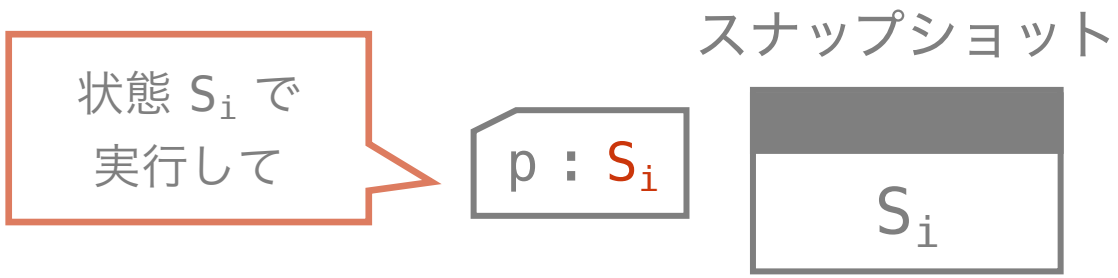
並行 UAF 発見の課題３

- ほとんどのカーネルファザーはカーネルを再起動しないため
  カーネルの状態は常に変化する
- 同じテストケースでも, カーネルの状態が変化すると動作が異なる
  場合がある
- 以前の状態で興味深いテストケースが現在の状態では効果的では
  なくなる可能性がある



並行 UAF 発見の課題３の解決法

- ファジング中に適切なタイミングで, カーネル状態のスナップショットを保存する
- 必要な場合にスナップショットを復元して, テストケースを実行する



Workflow of DDRace

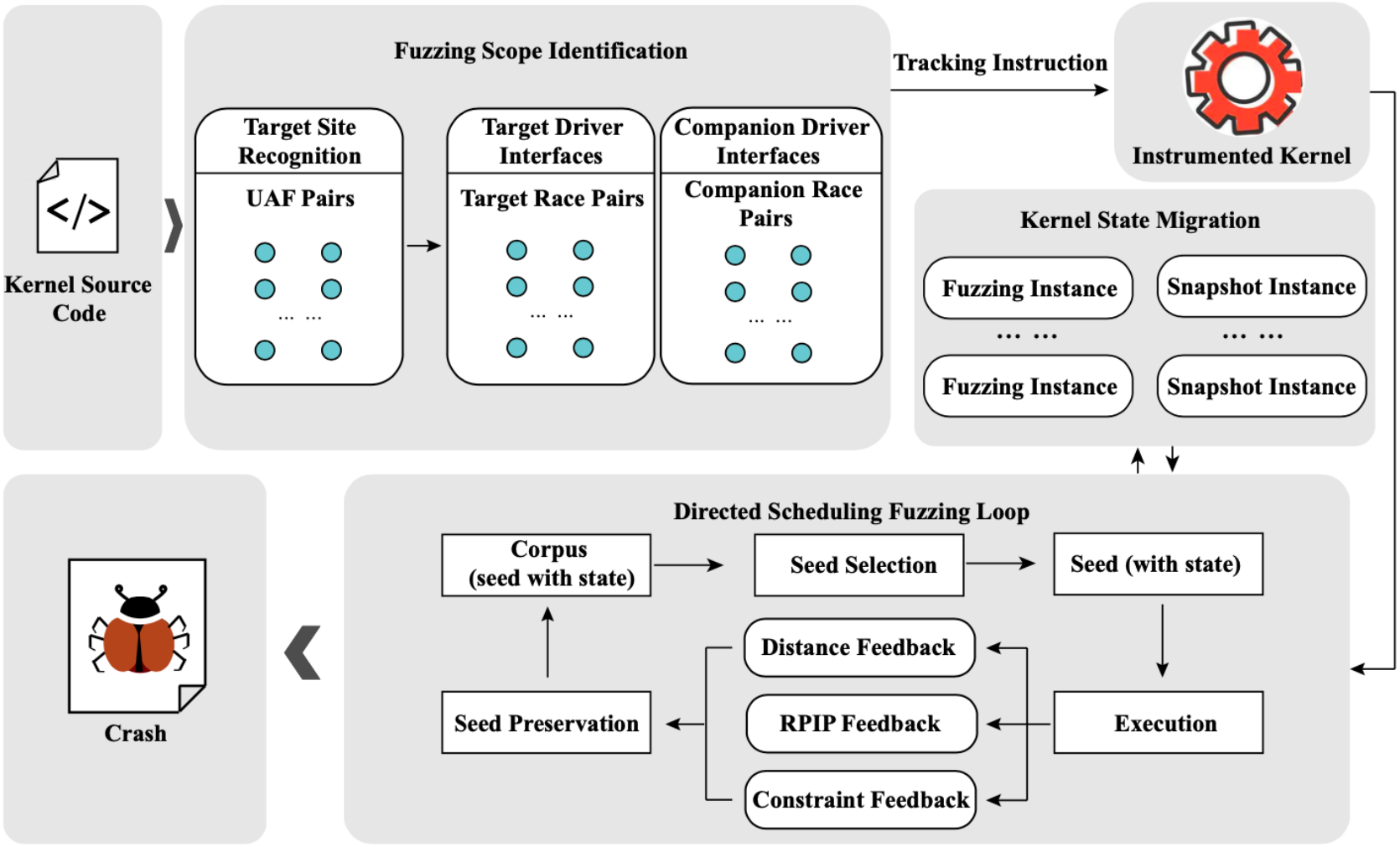


Figure 2: Workflow of DDRace.

Workflow of DDRace

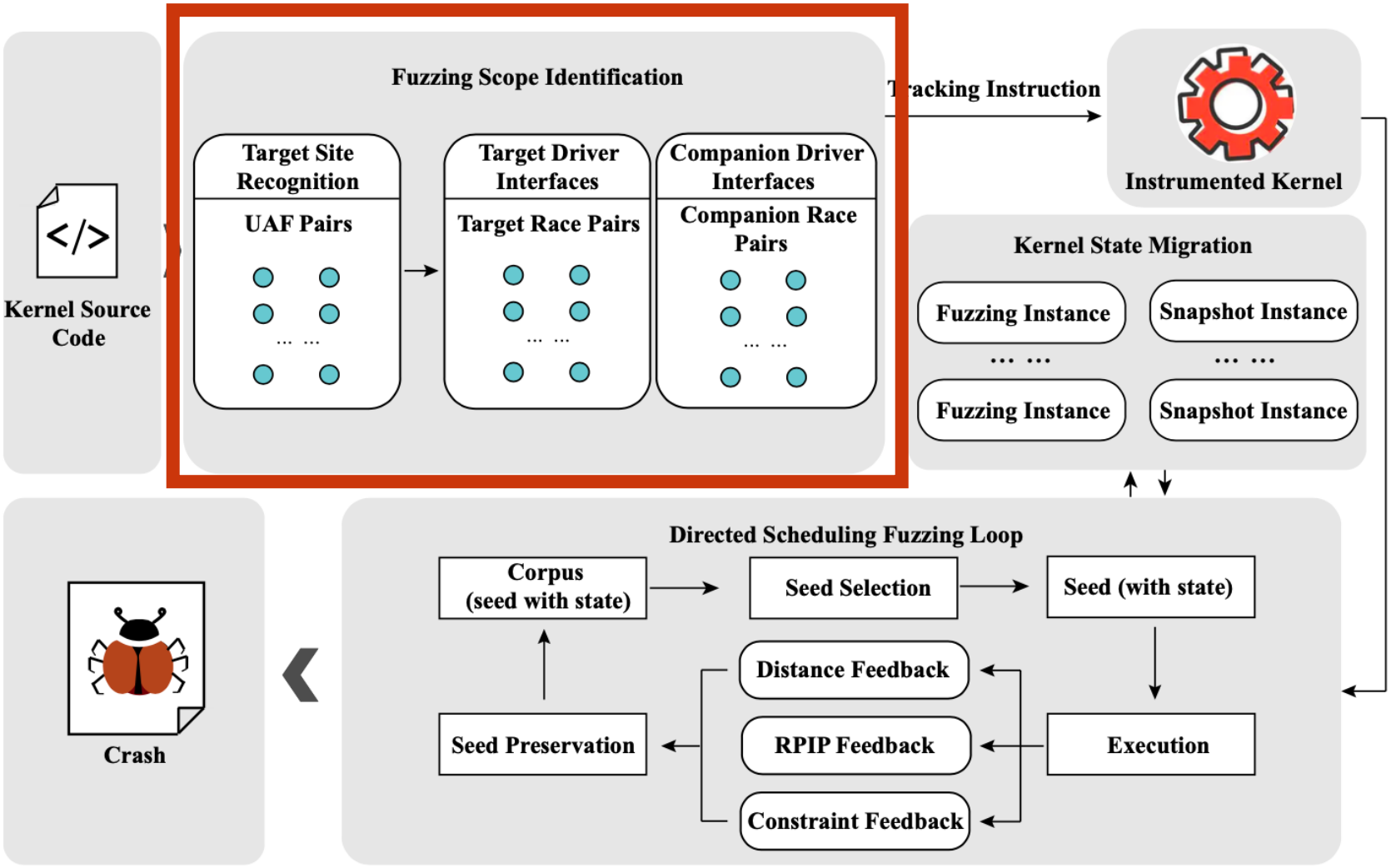


Figure 2: Workflow of DDRace.

ファジングスコープの特定

・Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある free・use 命令のペア（UAF ペア）を特定する

Thread1

```
1  // ioctl$KDGKBSENT
2  p = func_table[i];
3
4  ...
5
6  ... = *p
```

use

Thread2

```
11 // ioctl$KDSKBSENT
12 // oldptr holds the old func_table[i]
13 func_table[i] = ...;
14 ...
15 kfree(oldptr);
```

free

ファジングスコープの特定

- Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある free・use 命令のペア（UAF ペア）を特定する

1. free・use 命令を計装
2. カーネルファザーで一定時間実行
3. 計装から得られた情報を基に, 同じメモリオブジェクトを操作した free・use 命令のペアを報告

ファジングスコープの特定

- Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある `free`・`use` 命令のペア（UAF ペア）を特定する
- UAF ペアに到達可能なドライバインターフェースを特定

Thread1

```
1  // ioctl$KDGKBSENT
2  p = func_table[i];
3
4  ...
5
6  ... = *p
```

Thread2

```
11 // ioctl$KDSKBSENT
12 // oldptr holds the old func_table[i]
13 func_table[i] = ...;
14 ...
15 kfree(oldptr);
```

ファジングスコープの特定

- Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある `free`・`use` 命令のペア（UAF ペア）を特定する
- UAF ペアに到達可能なドライバインターフェースを特定
- ドライバインターフェースからターゲットまでにあるレースペアを特定

Thread1

```
1  // ioctl$KDGKBSENT
2  p = func_table[i];
3
4  ...
5
6  ... = *p
```

Thread2

```
11 // ioctl$KDSKBSENT
12 // oldptr holds the old func_table[i]
13 func_table[i] = ...;
14 ...
15 kfree(oldptr);
```

ファジングスコープの特定

- Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある `free`・`use` 命令のペア（UAF ペア）を特定する
- UAF ペアに到達可能なドライバインターフェースを特定
- ドライバインターフェースからターゲットまでにあるレースペアを特定
  - 静的解析
  - 共有メモリへの `read/write` 命令

ファジングスコープの特定

- Directed ファジングのターゲットとなる, 並行 UAF を引き起こす可能性のある `free`・`use` 命令のペア（UAF ペア）を特定する
- UAF ペアに到達可能なドライバインターフェースを特定
- ドライバインターフェースからターゲットまでにあるレースペアを特定
- point-to 解析によって他のドライバインターフェスから同じメモリにアクセスする可能性のある命令もレースペアとして取得

Workflow of DDRace

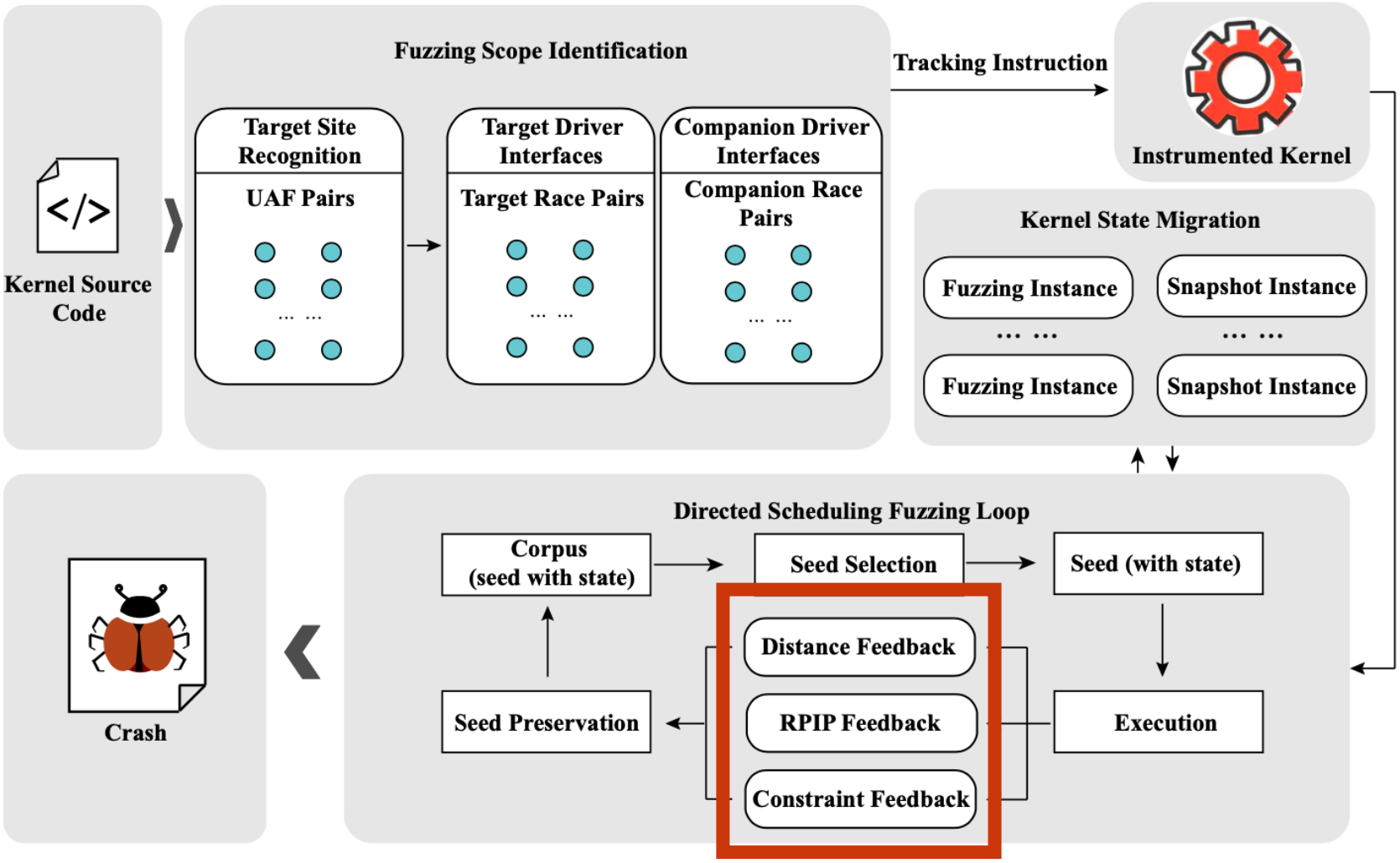


Figure 2: Workflow of DDRace.

Directed Fuzzing : フィードバック

- ここまででターゲット UAF ペアとターゲットレースペアを求めた

  最終的な目標はこの UAF ペアから UAF を引き起こすこと

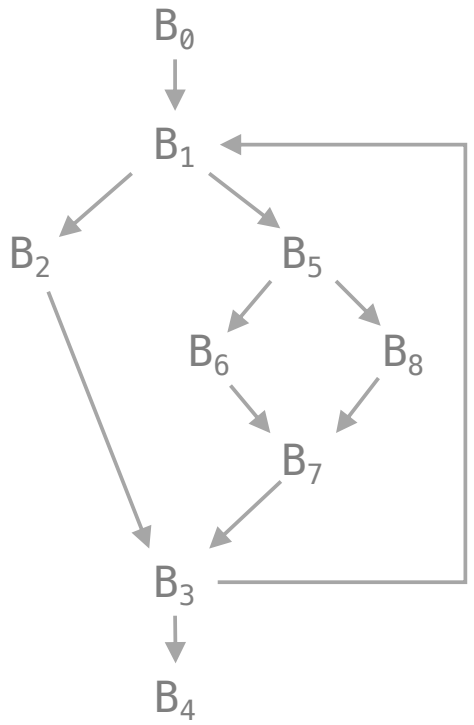
  ファジングによってターゲットレースペア間のスレッド切り替えを探索

- ターゲット UAF ペアに導くために新たな距離メトリクスを導入
  - ドミネータ深さ距離（dominator depth distance）
  - 脆弱性モデル制約距離（vulnerability model constraint distance）
- ターゲットレースペアに導くために並行実行フィードバックを導入

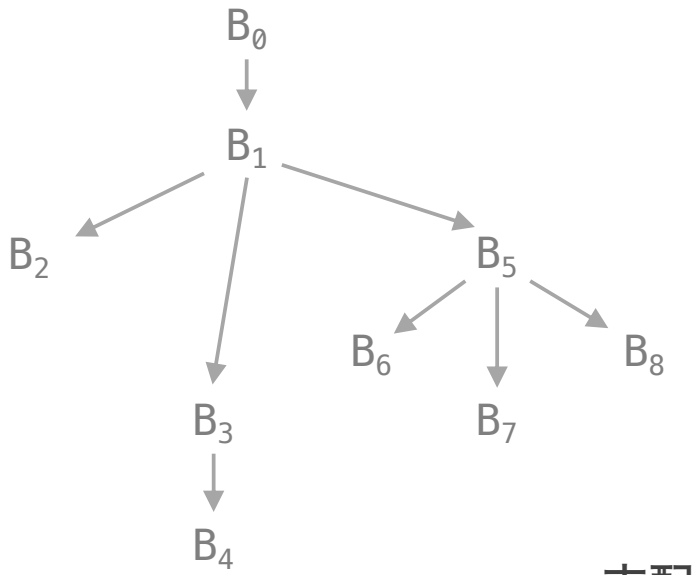
1 : Dominator Depth Distance

・CFG 上での距離ではなく, 支配木上の距離を用いる

→ CFG 上で距離が短いテストケースは, ターゲットサイトに到達するための制御フローの制約を満たすことができない場合もある



CFG



支配木

2 : Vulnerability Model Constraint Distance

・UAF 制約を満たす確率を測定するデータフロー距離

・UAF モデルは以下の3つの制約

1. free 操作と use 操作がある
2. この2つの操作は同じメモリを操作する
3. use 操作は free 操作の後に実行される

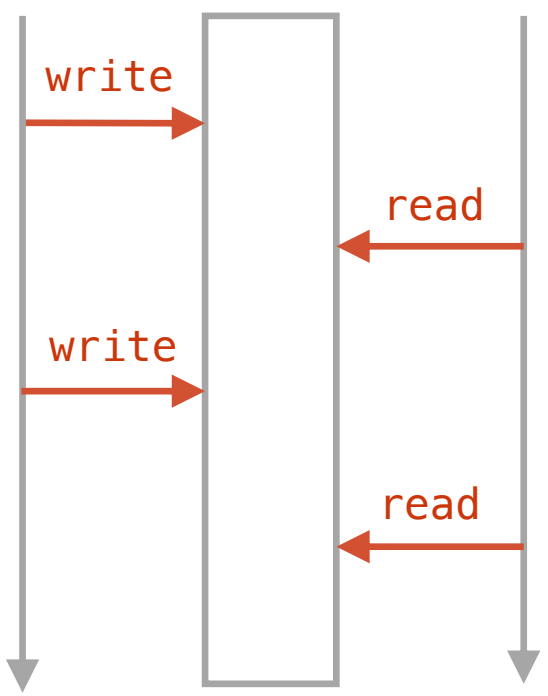
・1つ制約が満たされるとそれに応じて距離が縮まる

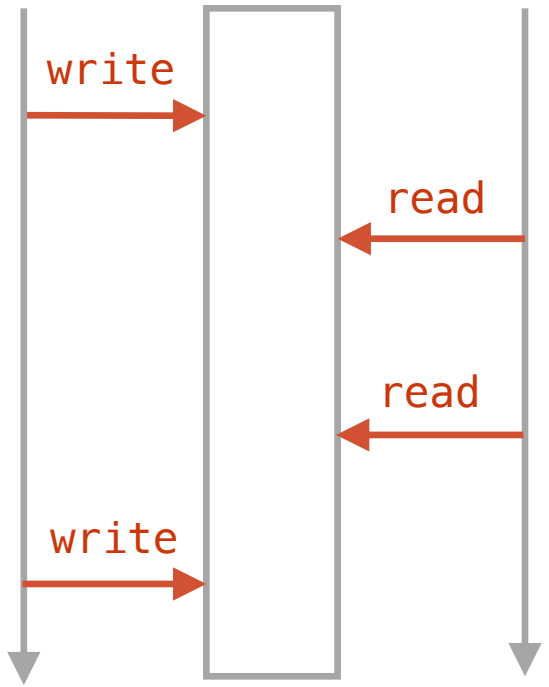
3 : Race Pair Interleaving Path Feedback

Race Pair Interleaving Path Feedback（RPIP）:

共有変数のメモリアクセスパターンをフィードバックとして活用する

→ 異なるメモリアクセスパターンを興味深いものとする





3 : Race Pair Interleaving Path Feedback

RPIP のアルゴリズム :

レースペアの各共有変数について最後の read/write の情報を保存する

タプル RT, WT を記録する

タプル $\langle ID_{instruction}, ID_{thread}, Value_{shared\ variable} \rangle$

- $ID_{instruction}$: 命令の ID
- ID_{thread} : スレッドの ID
- $Value_{shared\ variable}$: 共有変数の値

3 : Race Pair Interleaving Path Feedback

レースペアの各共有変数について最後の read/write の情報を保存する

タプル RT, WT を記録する

write → read

```
1  # T: test case
2  # t: thread
3  # v: value of the sheared variable
4  # InterEdge : thread-interleaving edge
5  for i in instructions(T):
6    if isReadAccess(i):
7      RT = <i,t,v>
8      i_w,t_w,v_w = WT
9      if t != t_w:
10       InterEdge.insert(<i_w,i>)
```

異なるスレッドのとき
write → read を記録

3 : Race Pair Interleaving Path Feedback

レースペアの各共有変数について最後の read/write の情報を保存する

タプル RT, WT を記録する

read → write

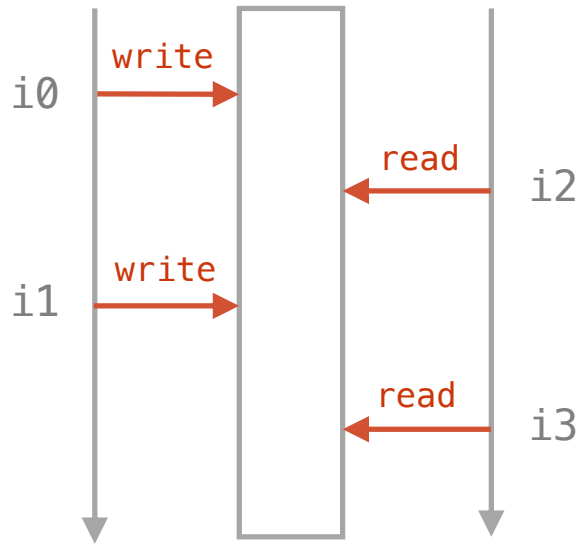
```
1  # T: test case
2  # t: thread
3  # v: value of the sheared variable
4  # InterEdge : thread-interleaving edge
5  for i in instructions(T):
6    if isWriteAccess(i):
7      oldValue = GetOperandValue(i)
8      if oldValue != v:
9        WT = <i,t,v>
10       i_r,t_r,v_r = RT
11       if t != t_r and v != v_r:
12         InterEdge.insert(<i_r,i>)
```

異なるスレッド,
異なる値を書き込むとき
read → write を記録

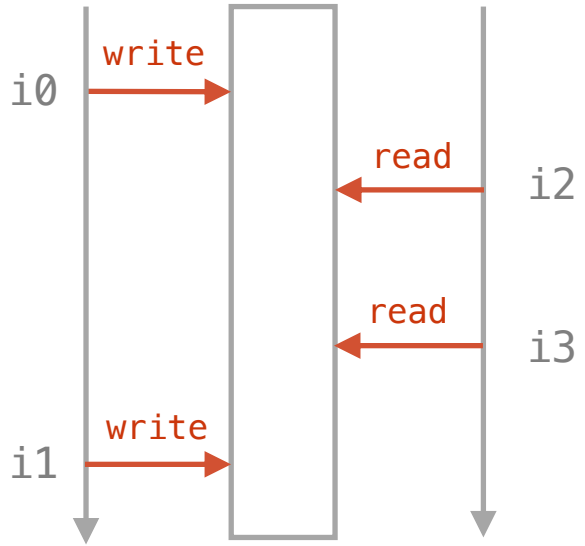
3 : Race Pair Interleaving Path Feedback

Race Pair Interleaving Path Feedback（RPIP）:

最終的に得られた `InterEdge [<i0,i1>, <i2,i3>, …]` を
ハッシュ化したもの




```
InterEdge :
  [<i0,i2>, <i2,i1>, <i1,i3>]
```




```
InterEdge :
  [<i0,i2>, <i3,i1>]
```

Workflow of DDRace

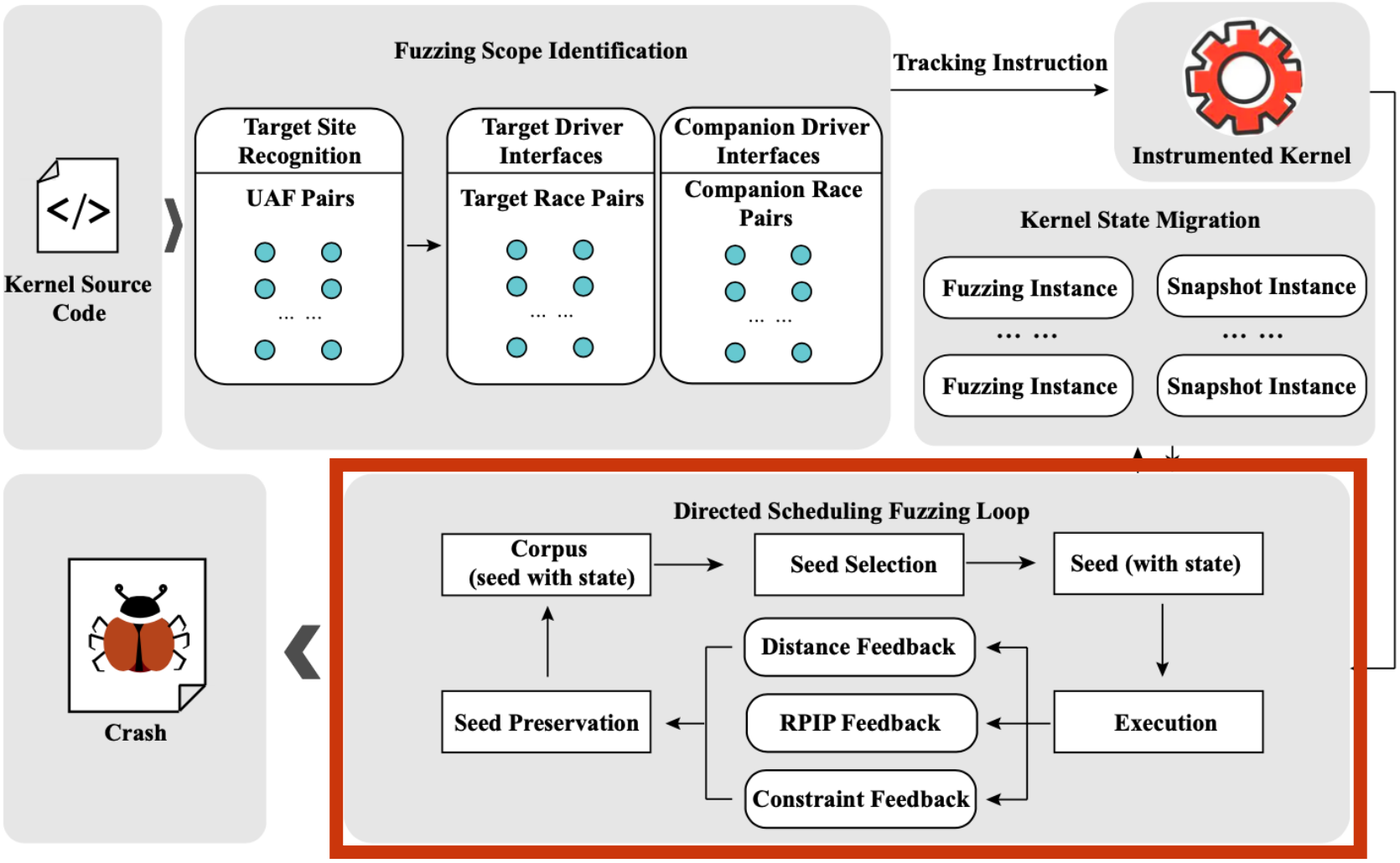


Figure 2: Workflow of DDRace.

Directed Fuzzing : 保存戦略

2つのシードの保存戦略 :

1: <u>ターゲットまでの距離</u>

距離 = ドミネータ深さ距離 + 脆弱性モデル制約距離

・ドミネータ深さ距離 : 支配木上でのターゲットまでの距離

・脆弱性モデル距離 : UAF 制約を満たすまでの距離

・最短距離でなくとも, 新しい距離の値が観測されたらテストケースを保存

→ スレッドのスケジューリングで距離を縮められるかも

Directed Fuzzing : 保存戦略

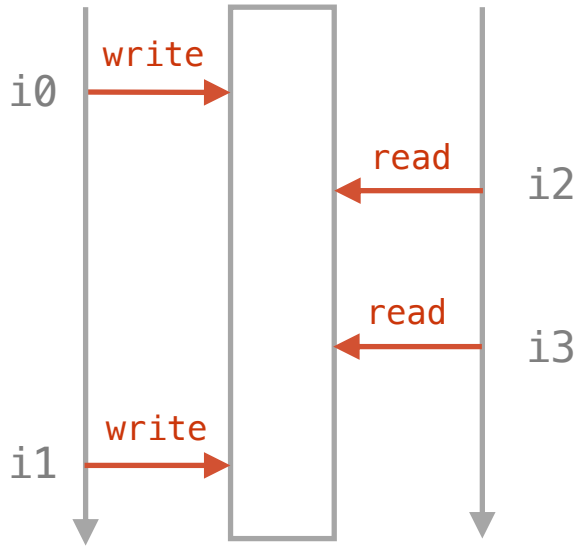
2つのシードの保存戦略 :

2: RPIP

・新しい RPIP が観測されたらそのテストケースを保存

・未発見の近傍スレッド間エッジ（neighber thread-interleaving edges）の数が平均より多いテストケースを保存

→ レースペアを活用？



RPIP :

[<i0,i2>, <i3,i1>]

Directed Fuzzing : 選択戦略

2つのシードの選択戦略 :

1: 距離の短いシードを優先

2: 稀なスレッドインターリーブを持つシードを優先

→ 稀かどうかはファジング中に判定？

Directed Fuzzing : 変異戦略

2段階のシードの変異戦略 :

1: **ターゲットサイトに到達するまで**

Syzkaller と同じ変異戦略

システムコールの追加・削除, パラメータの変異など

2: **ターゲットサイトに到達した後**

スレッドスケジューリング（`setdelay`）の変異に重点を置く

- 計装の段階でレースペア命令の前に計装
- `setdelay(inst, delay)` システムコールを作成
  → レースペア命令（の一つ）`inst` を `delay` 時間遅延させる

Workflow of DDRace

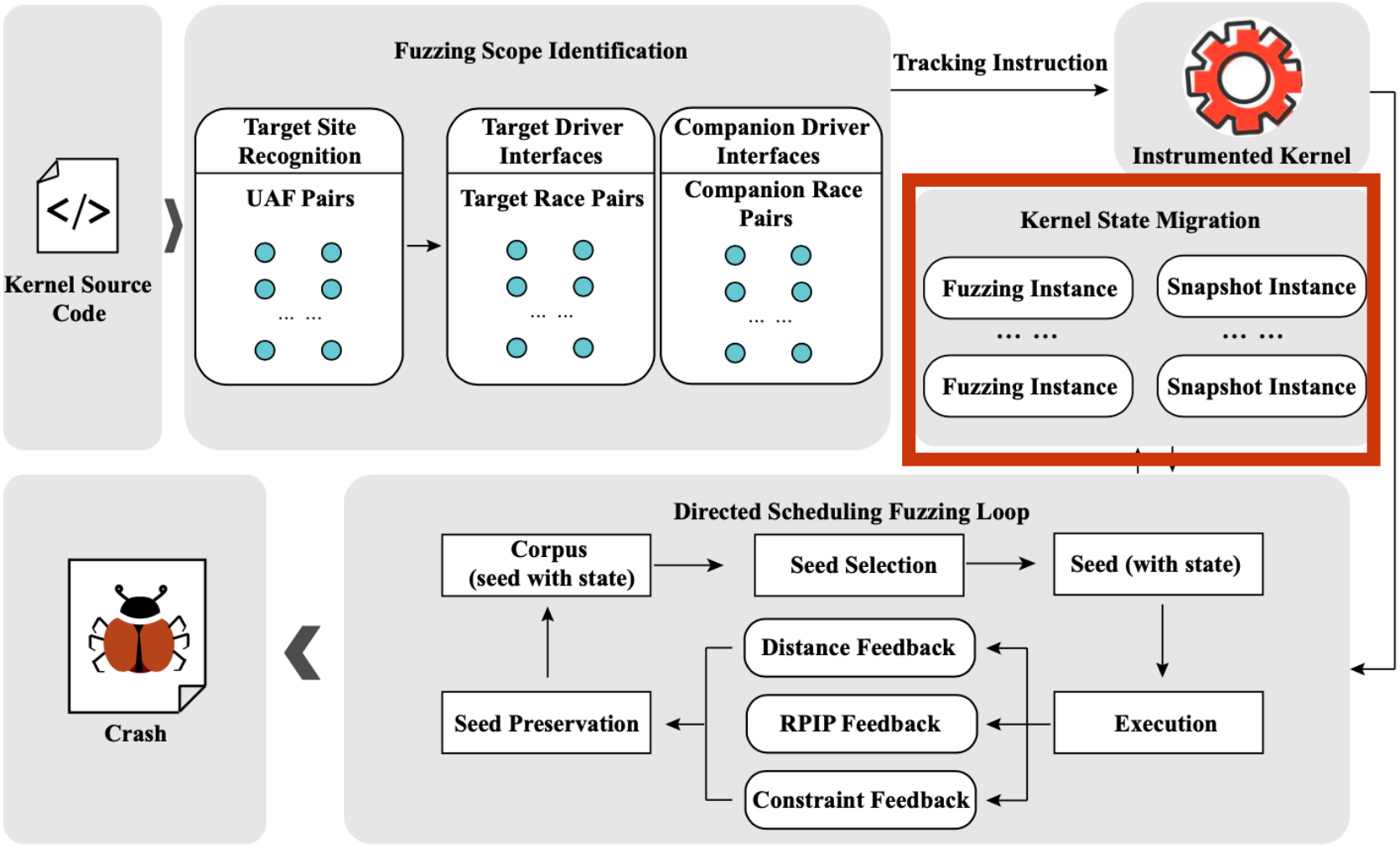


Figure 2: Workflow of DDRace.

Adaptive State Migration

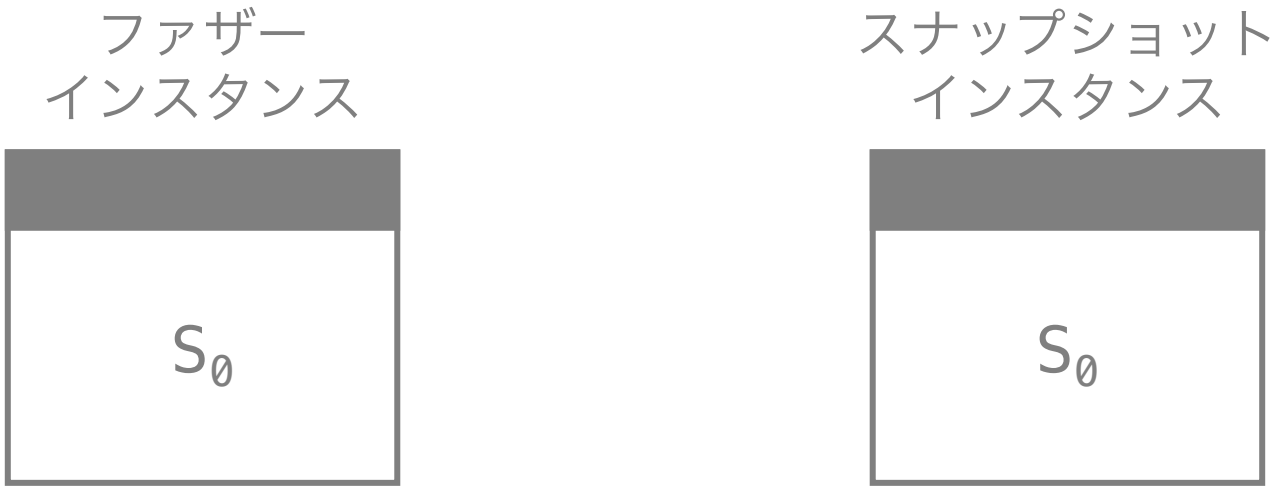
・ほとんどのカーネルファザーはカーネルを再起動しないため
カーネルの状態は常に変化する

→ 以前の状態で興味深いテストケースが現在の状態では効果的では
なくなる可能性がある

・必要に応じてカーネル状態のスナップショットを作成・復元する

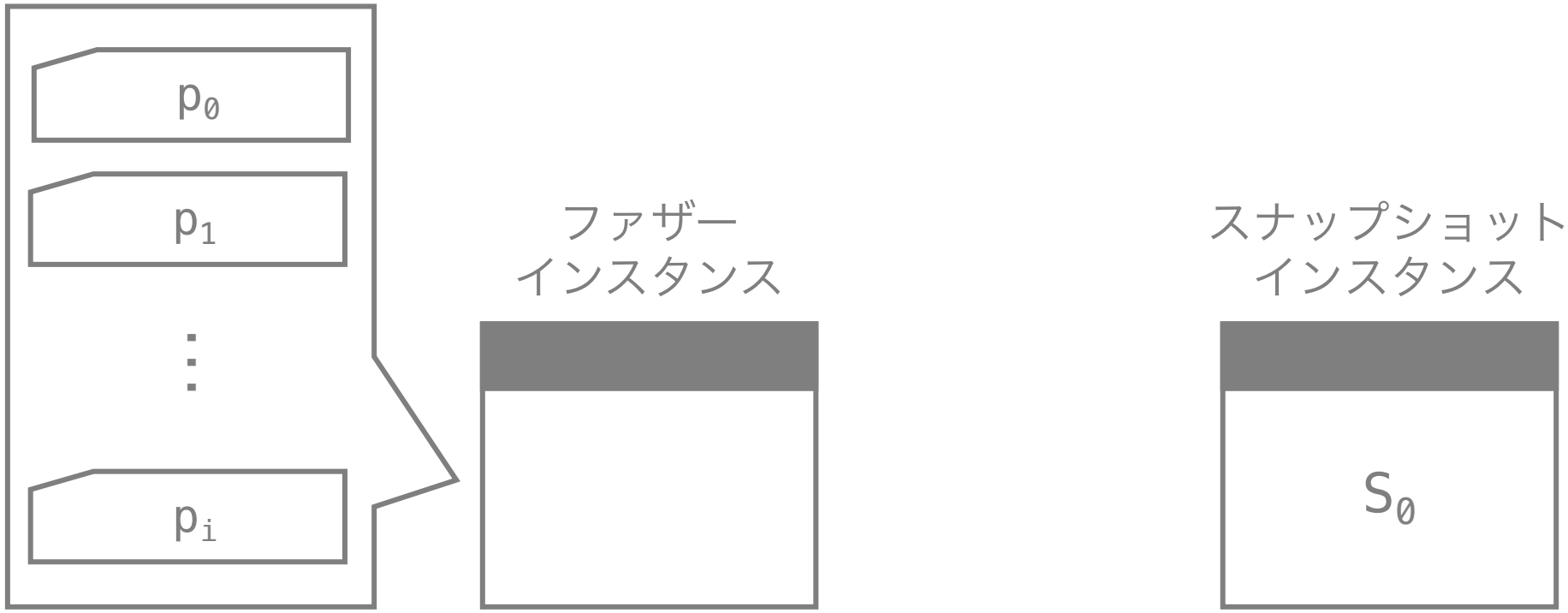
Adaptive State Migration

・カーネルの初期状態 S_0



Adaptive State Migration

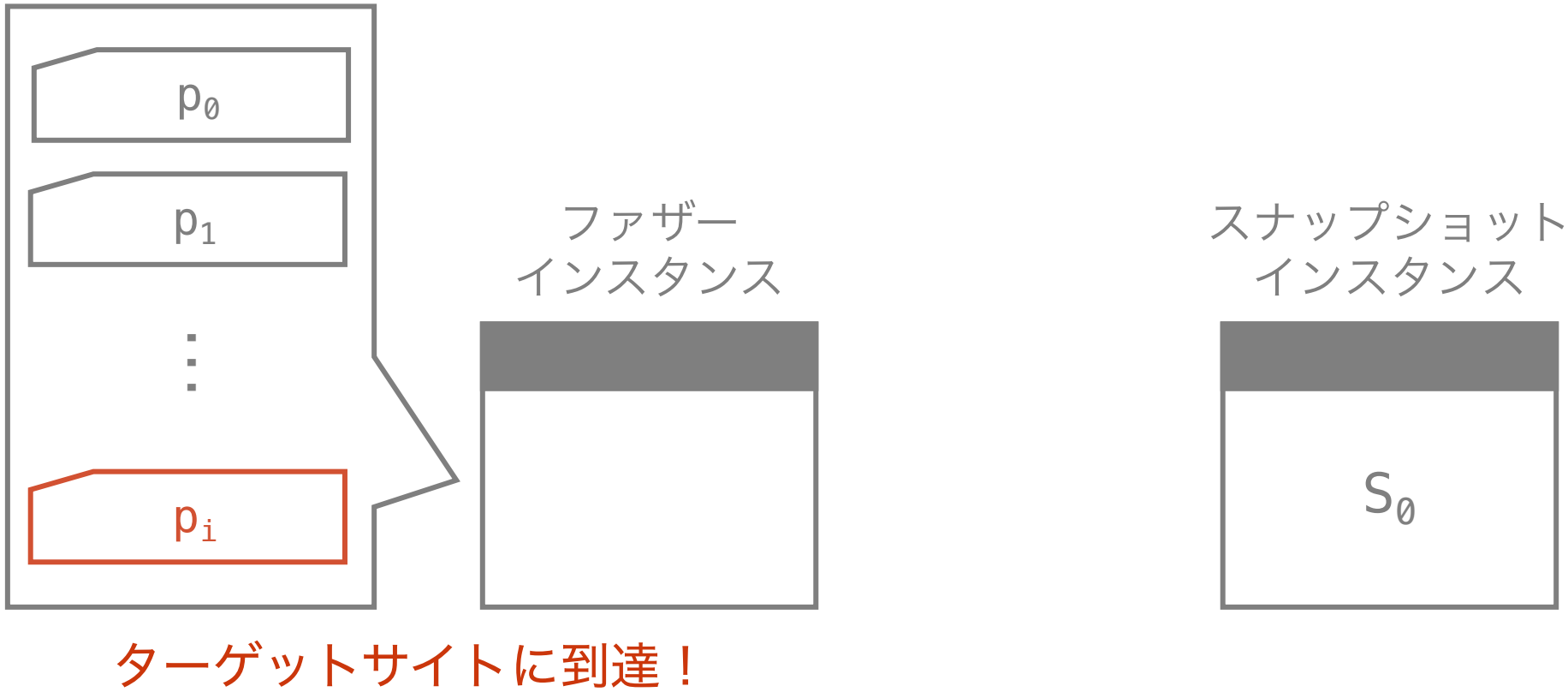
・ファザーインスタンスでファジングを行う



Adaptive State Migration

・価値のあるテストケース p_i に対してスナップショットを作成するか判定

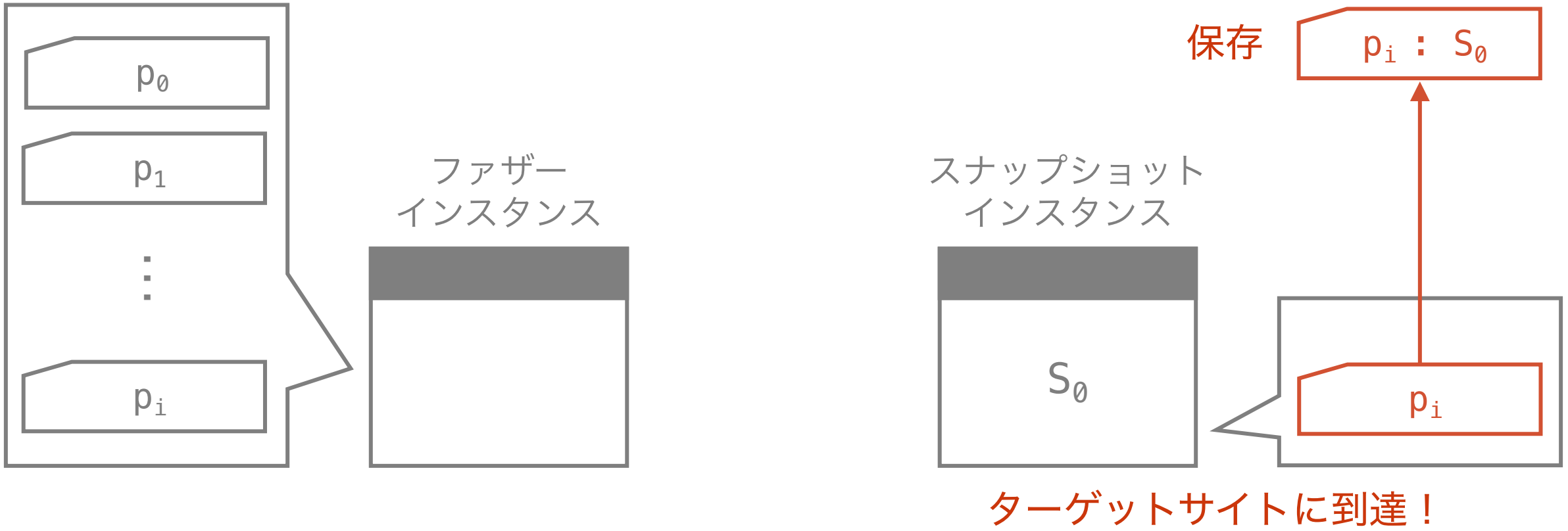
価値のあるテストケース： ターゲットサイトに到達 or 最短距離を得た



Adaptive State Migration

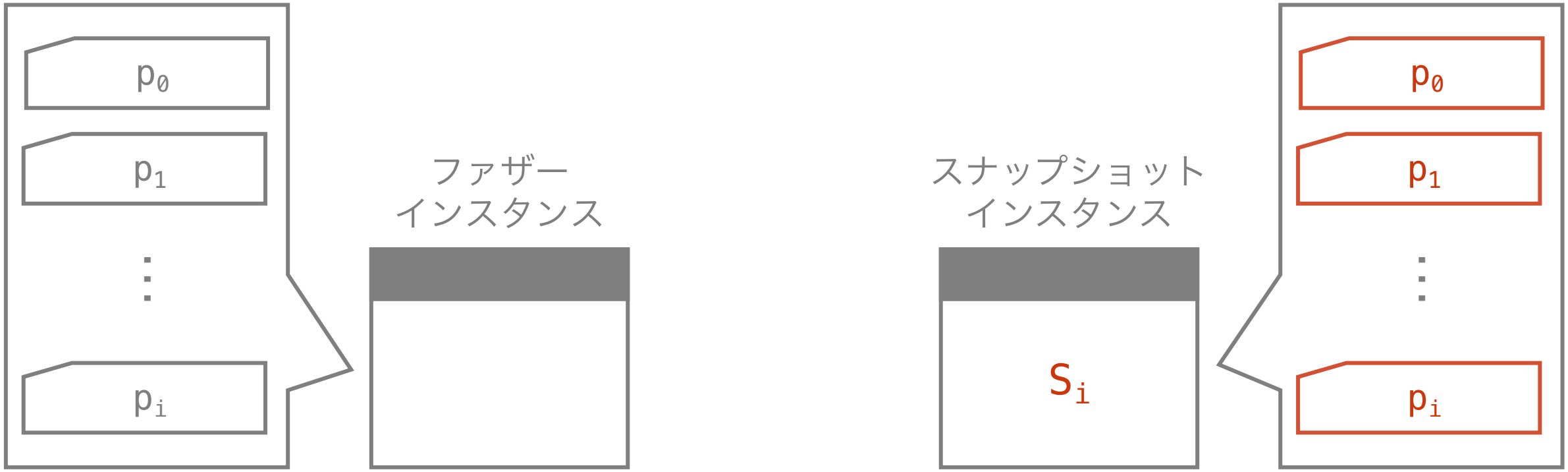
・カーネルの初期状態 S_0 で p_i を実行

→ 同じ結果（ターゲットサイトに到達・同じ距離）が得られたら
p_i に S_0 でアノテーションし保存



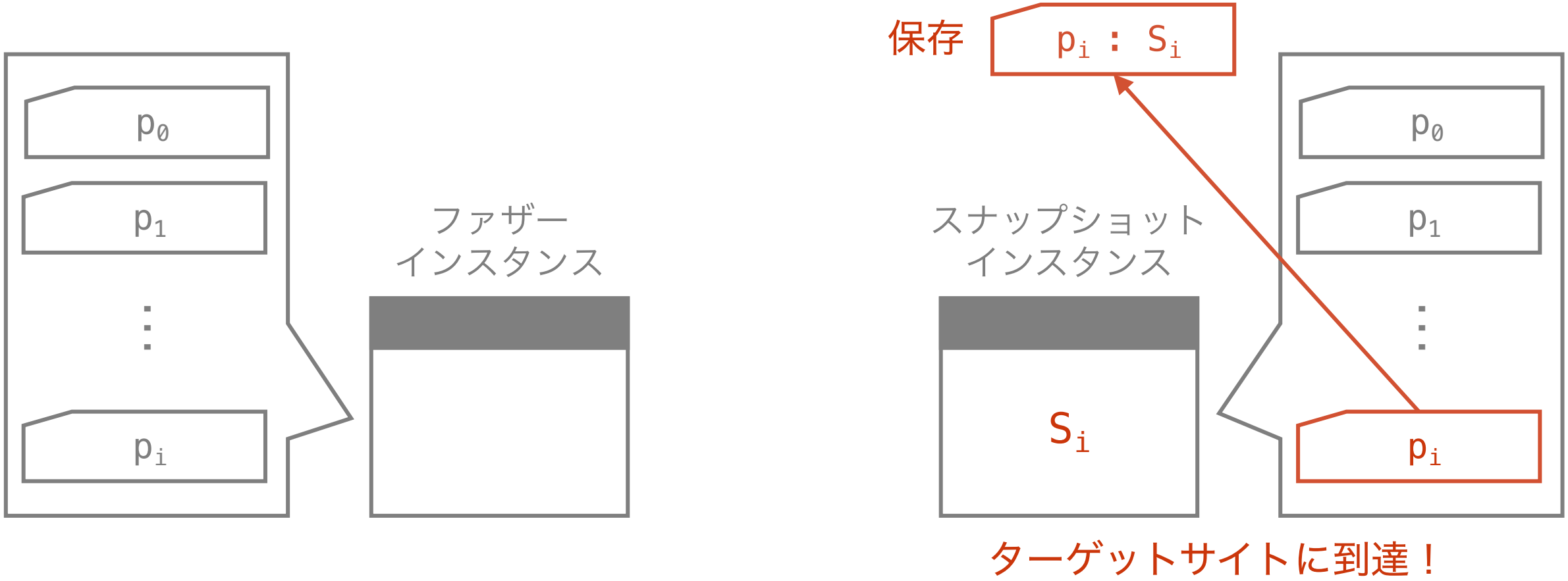
Adaptive State Migration

・違う結果の場合, 初期状態 S_0 から p_i までに実行したテストケースを
すべてもう一度実行してみる



Adaptive State Migration

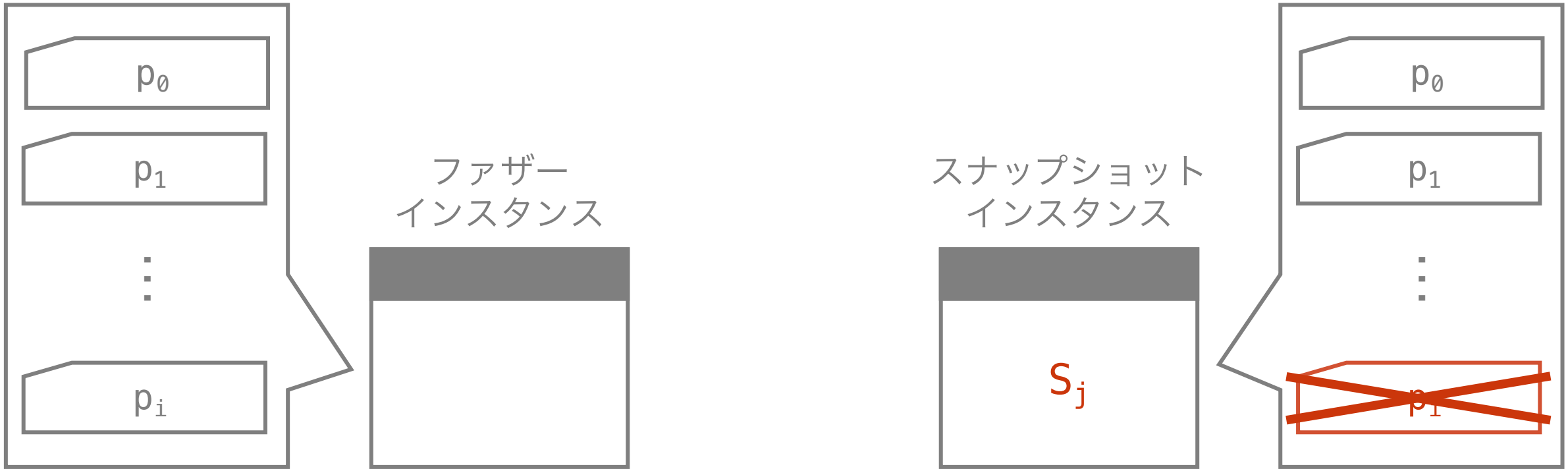
・同じ結果が得られたら, 現在のカーネル状態 S_i のスナップショットを作成, p_i に S_i でアノテーションし保存



Adaptive State Migration

・違う結果の場合, p_i を破棄

→ 割り込みなど別の要因で状態が変化した可能性



Adaptive State Migration

- 価値のあるテストケース p_i に対してスナップショットを作成するか判定

  価値のあるテストケース： ターゲットサイトに到達 or 最短距離を得た

- カーネルの初期状態 S_0 で p_i を実行

  → 同じ結果（ターゲットサイトに到達・同じ距離）が得られたら

  p_i に S_0 でアノテーションし保存

- 違う結果の場合, 初期状態 S_0 から p_i までに実行したテストケースを
  すべてもう一度実行してみる

  → 同じ結果が得られたら, 現在のカーネル状態 S_i のスナップショットを作成,

  p_i に S_i でアノテーションし保存

  → 違う結果の場合, p_i を破棄

山口 vs 鹿児島

Implementation

- CRIX : 制御フロー・コールフローグラフの作成に使用
- SVF : point-to 解析に使用
  → Linux カーネルは高度にモジュール化されているため,
  各サブモジュールに対して point-to 解析を適用
- Syzkaller は各システムコールのスレッドをランダムに選択
  → DDRace では各システムコールに
  スレッドID を付与して保存

Table 1: Implementation details of DDRace.

Component	Base Tool	LoC
Race Pair Extraction	SVF, CRIX, LLVM	1,500 (C++)
Fuzzing Loop	Syzkaller	2,500 (Go)
Instrumentation	LLVM SanitizerCoverage	600 (C++)
Glue scripts	-	1,000 (Python)
Total	-	5,600

Evaluation

- RQ1: DDRace はターゲットに関連したレースペアの発見にどの程度効果的か？
- RQ2: DDRace は並行 UAF を発見できるか？
- RQ3: DDRace は既存の手法と比較してどうか？
- RQ4: DDRace の各構成要素はどの程度有効か？

Experimental Setup

Dataset

- Linux カーネル 4.19.100 の6つのドライバ

  tty, drm, sequencer, midi, vivid, floppy

Environment and Configuration

- LLVM-10.0 で syzbot [6] の設定で Linux カーネルをコンパイル
  - コードカバレッジ計装のために KCOV を有効に
  - UAF 検出のために KASAN を有効に
- Intel Xeon（E5-2695 v4 2.10GHz）と 384GB RAM を搭載した Ubuntu 16.04 LTS 上で実験

[6] https://syzkaller.appspot.com

RQ1: Race Pair Extraction

RQ1: ターゲットに関連したレースペア
発見の有効性

・Syzkaller を24時間実行し
227 組の UAF ペアを発見

・UAF ペアに関連するレースペアは
ほとんどが 20 個以下（図3）

→ Razzer [4] や Krace [5] と比較しても少ない

・関連するドライバインターフェースも
少ない（図4）

→ 少数に集中して計算資源を割り当てられる

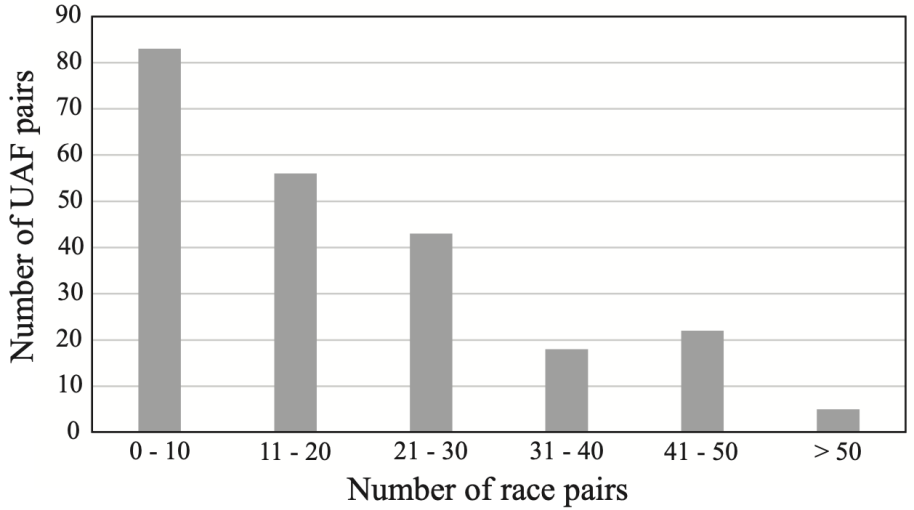


Figure 3: Race pairs statistics.

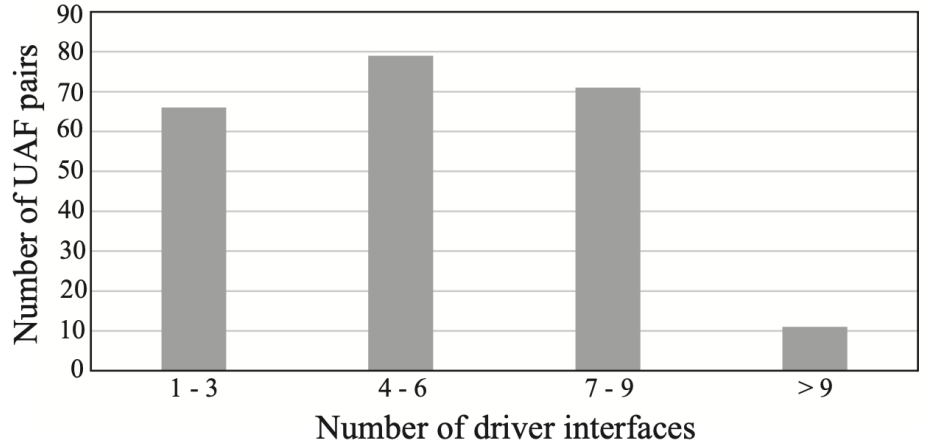


Figure 4: Driver interfaces statistics.

[4] Razzer: Finding Kernel Race Bugs through Fuzzing [SP'19]
[5] Krace: Data Race Fuzzing for Kernel File Systems [SP'20]

RQ2: Vulnerability Discovery

RQ2: DDRace は並行 UAF を発見できるか？

・各 UAF ペアに対して 12時間のファジングを行う

12時間 × 227 = 2724 時間 = 113.5日！

・12個の並行 UAF 脆弱性を発見（内 4個は未知の脆弱性）

・227個の UAF ペアから発見できた脆弱性はわずか12個

→ ターゲットサイトに到達するだけでは脆弱性を見つけるのに十分でない

→ DDRace がクラッシュを検出しても, 並行脆弱性を診断・再現するのは多くの手作業が必要

[4] Razzer: Finding Kernel Race Bugs through Fuzzing [SP'19]
[5] Krace: Data Race Fuzzing for Kernel File Systems [SP'20]

RQ3: Comparison

RQ3: DDRace は既存の手法と比較してどうか？

<u>比較ファザー</u>

- Syzkaller [7]　コードカバレッジフィードバックを持つベースファザー
  スレッドインターリーブ探索は未サポート
- Razzer [4]　ファジング中にランダムにレースペアを選択し,
  スケジューリングにハードウェアブレイクポイントを使用する
- Krace [5]　並行カバレッジを導入したファザー
  スケジューリングにランダムな遅延を挿入する

- それぞれ 12時間 × 5回 実行する

[7] https://github.com/google/syzkaller
[4] Razzer: Finding Kernel Race Bugs through Fuzzing [SP'19]
[5] Krace: Data Race Fuzzing for Kernel File Systems [SP'20]

RQ3: Comparison

RQ3: DDRace は既存の手法と比較してどうか？

- DDRace は他の手法と比較してより多く，
  より早く並行 UAF 脆弱性を発見する

- Krace が syzkaller と比較して脆弱性の
  発見が遅いのは，ランダムなスケジューリング
  によって多くの無駄が発生するため

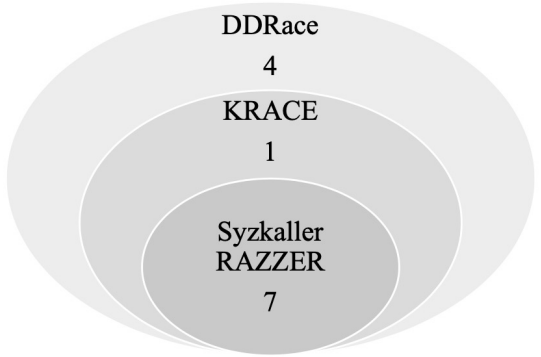


Figure 5: The Venn diagram of vulnerability discovery results on the Linux kernel 4.19.100.

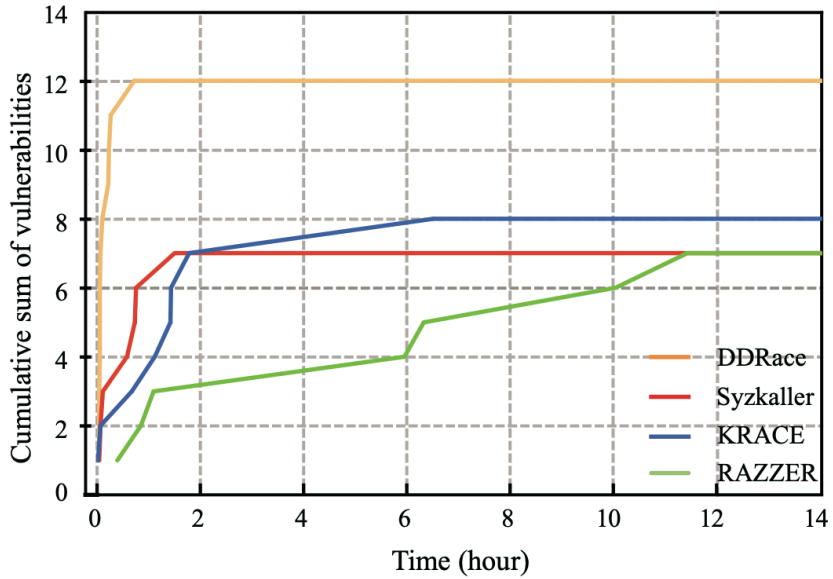


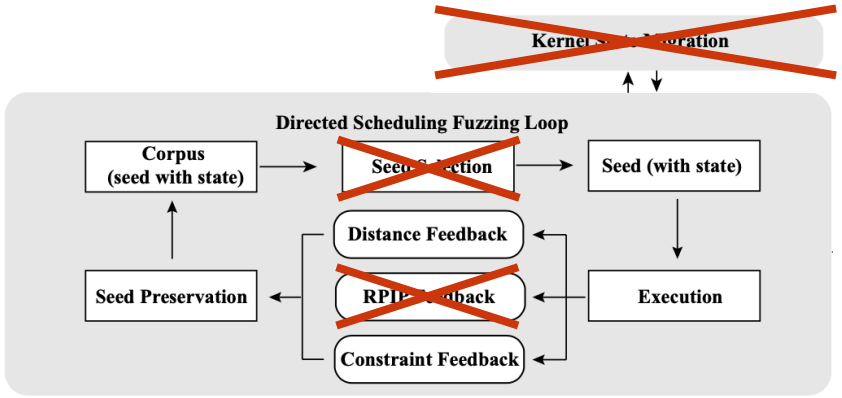
Figure 6: Comparison of vulnerability-finding time for 12 concurrency UAF vulnerabilities.

RQ4: Ablation Study

RQ4: DDRace の各構成要素はどの程度有効か？

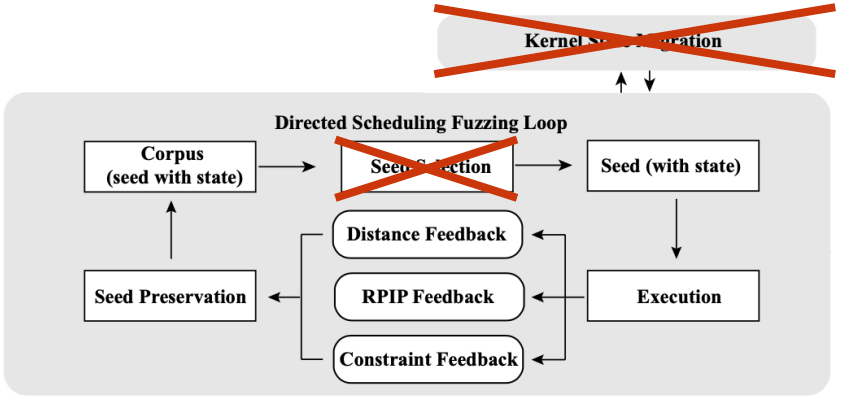
・DDRace_D

距離フィードバックのみを
有効にする



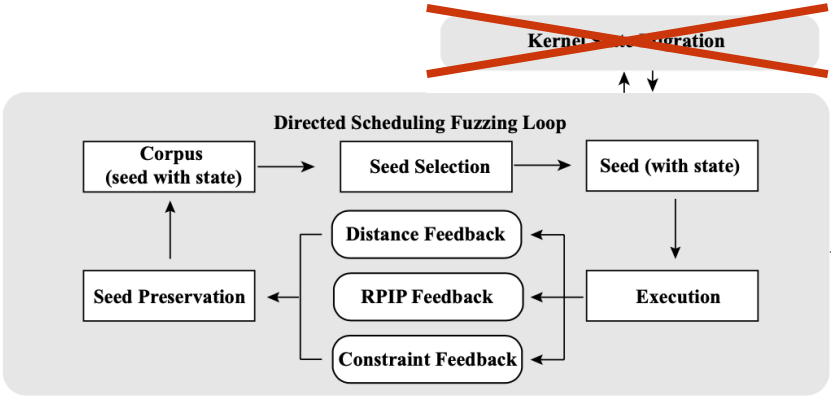
・DDRace_R

DDRace_D +
RPIP のフィードバック



・DDRace_P

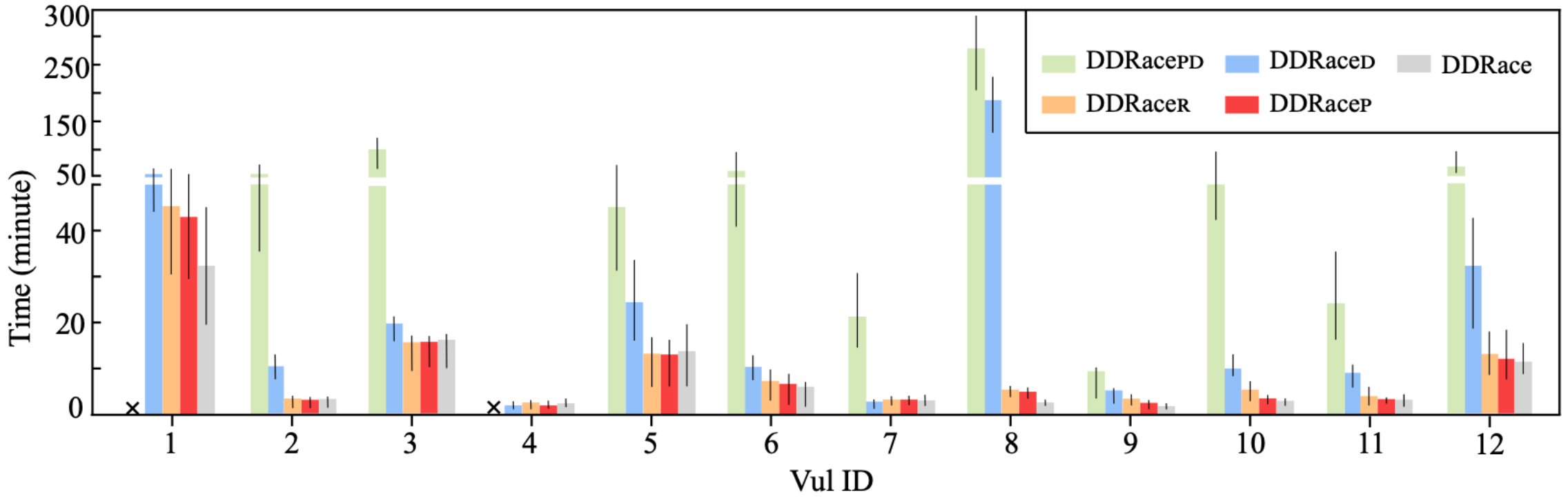
DDRace_R +
並行シードの選択戦略



・DDRace_{PD} : UAF ペアに向けた素朴な距離フィードバックファザー

RQ4: Ablation Study

RQ4: DDRace の各構成要素はどの程度有効か？



Discussion 1

- 最近のカーネルファザーの研究はシステムコール間の依存関係の抽出を目指す

  例) `open` してから `write`

- これらの手法は DDRace と直交するため適用可能

  例) `open` と `write` の両方に含まれるレースペアは同時に実行できない

Discussion 2

・静的解析では false-positive が発生する

→ ファジングで候補が正しいか確認する

・静的解析ではドライバサブモジュール内のみを解析するため
[　　]が発生する

→ 異なるカーネルモジュール間のレースペアの見逃しが発生する

Q.　true-positive
　　true-negative
　　false-positive
　　false-negative

Discussion 2

- 静的解析では false-positive が発生する

  → ファジングで候補が正しいか確認する

- 静的解析ではドライバサブモジュール内のみを解析するため
  false-negative が発生する

  → 異なるカーネルモジュール間のレースペアの見逃しが発生する

  → 一般的に異なるカーネルモジュール間が頻繁に相互作用することは少ない

Discussion 3

- レースペア間のスケジューリングに加えて, ロックやセマフォなども
  プログラムに影響を与える

  → DDRace ではこのような同期を考慮に入れていない

Discussion 4

- DDRace は他のモジュールに対しても適用可能

  → Linux ドライバは高度にモジュール化されているため
  ターゲットの絞り込みが容易

DDRace [1] のまとめ

ジャンル： Linux デバイスドライバファジング

問題提起：
- 従来のスレッド間探索ファザーは主にデータ競合が対象
- Directed ファジングはスレッドのインターリーブを無視する

提案手法： Linux ドライバの並行 UAF を発見するための並行 directed ファザー

1. 事前解析によって, ファジングの探索空間を削減
2. 新しいフィードバックによって並行 UAF 脆弱性と
   スレッドインターリーブを効果的に探索
3. テストケースの再現性を高めるためにスナップショットを活用

結果： Linux ドライバの4つの未知の脆弱性と8つの既知の脆弱性を発見

[1] DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing [Usenix-Security'23]