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]

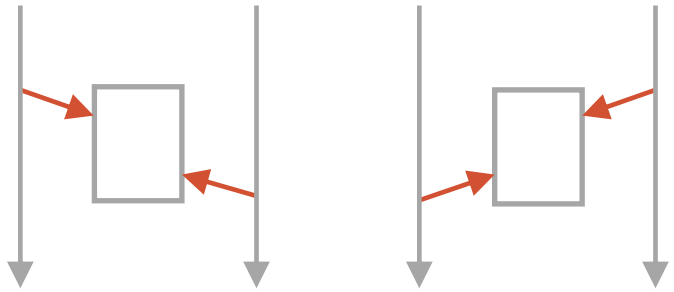
背景知識：並行性バグ

・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？

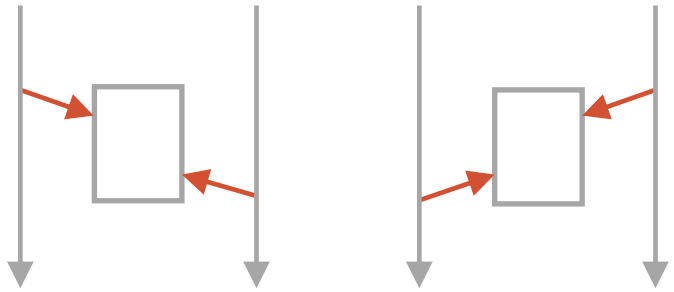
背景知識：並行性バグ

・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

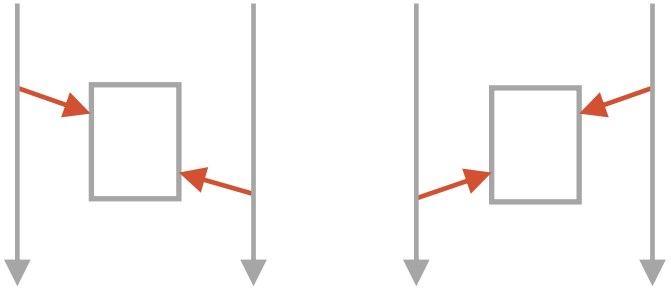
背景知識：並行性バグ

・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 }
```

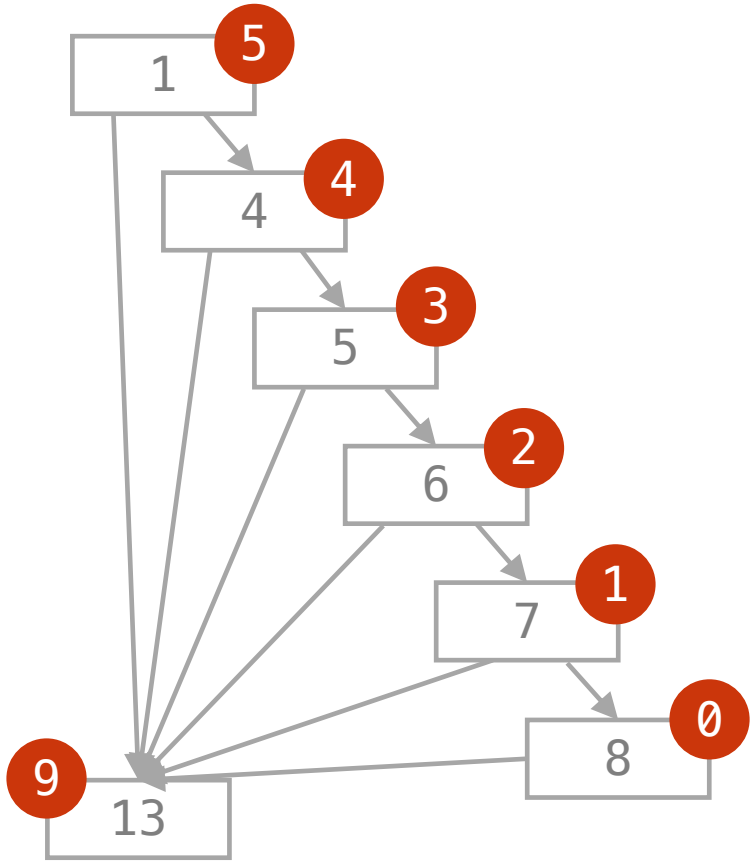
[2] Directed Greybox Fuzzing [CCS'17]

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
```

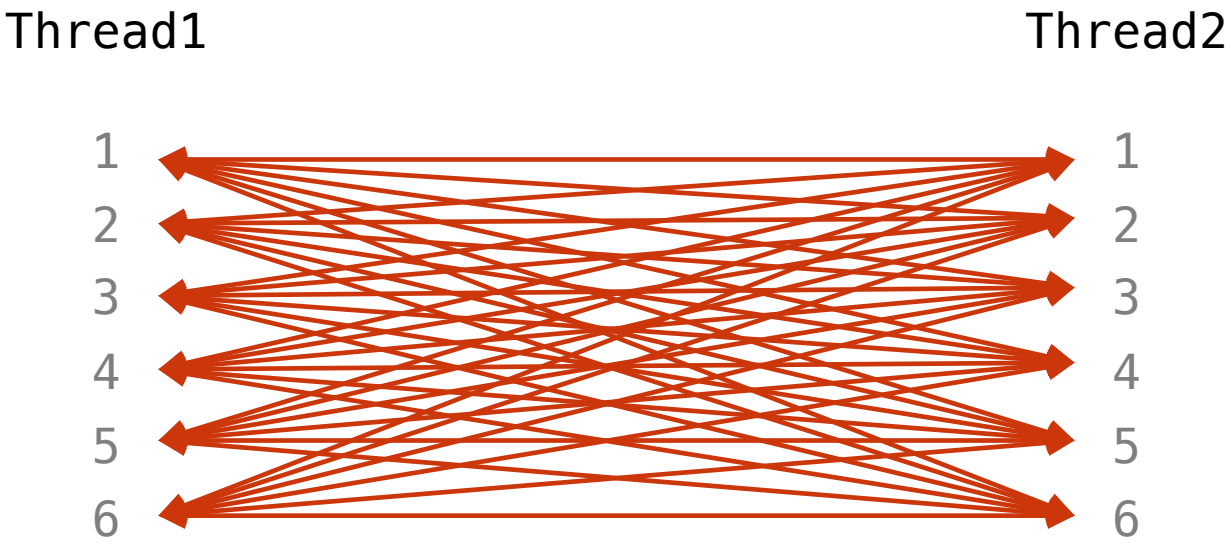
Thread2

```
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



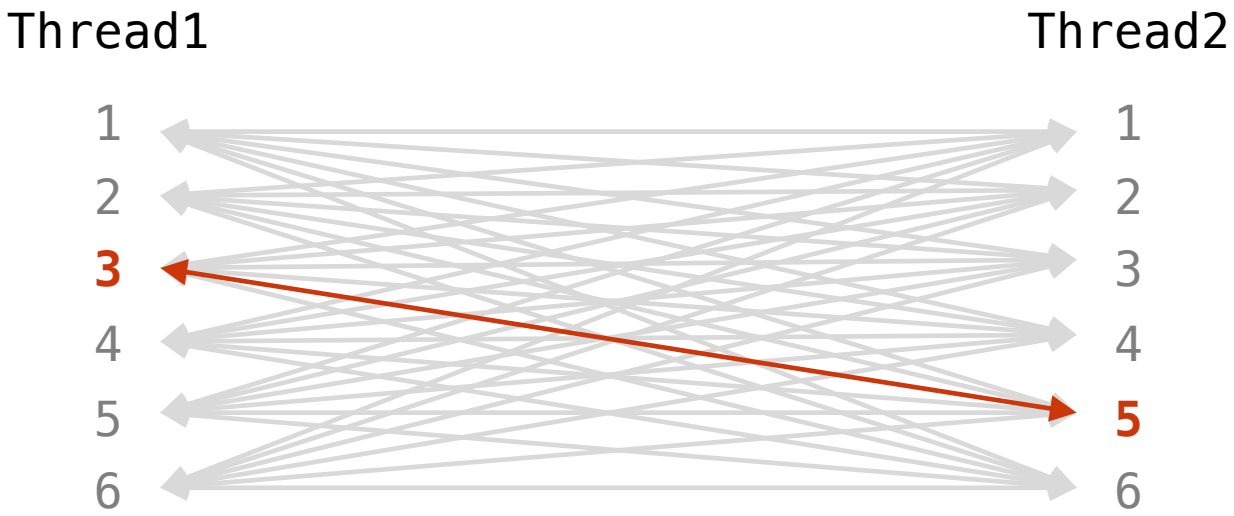
並行 UAF 発見の課題１の解決法

- 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

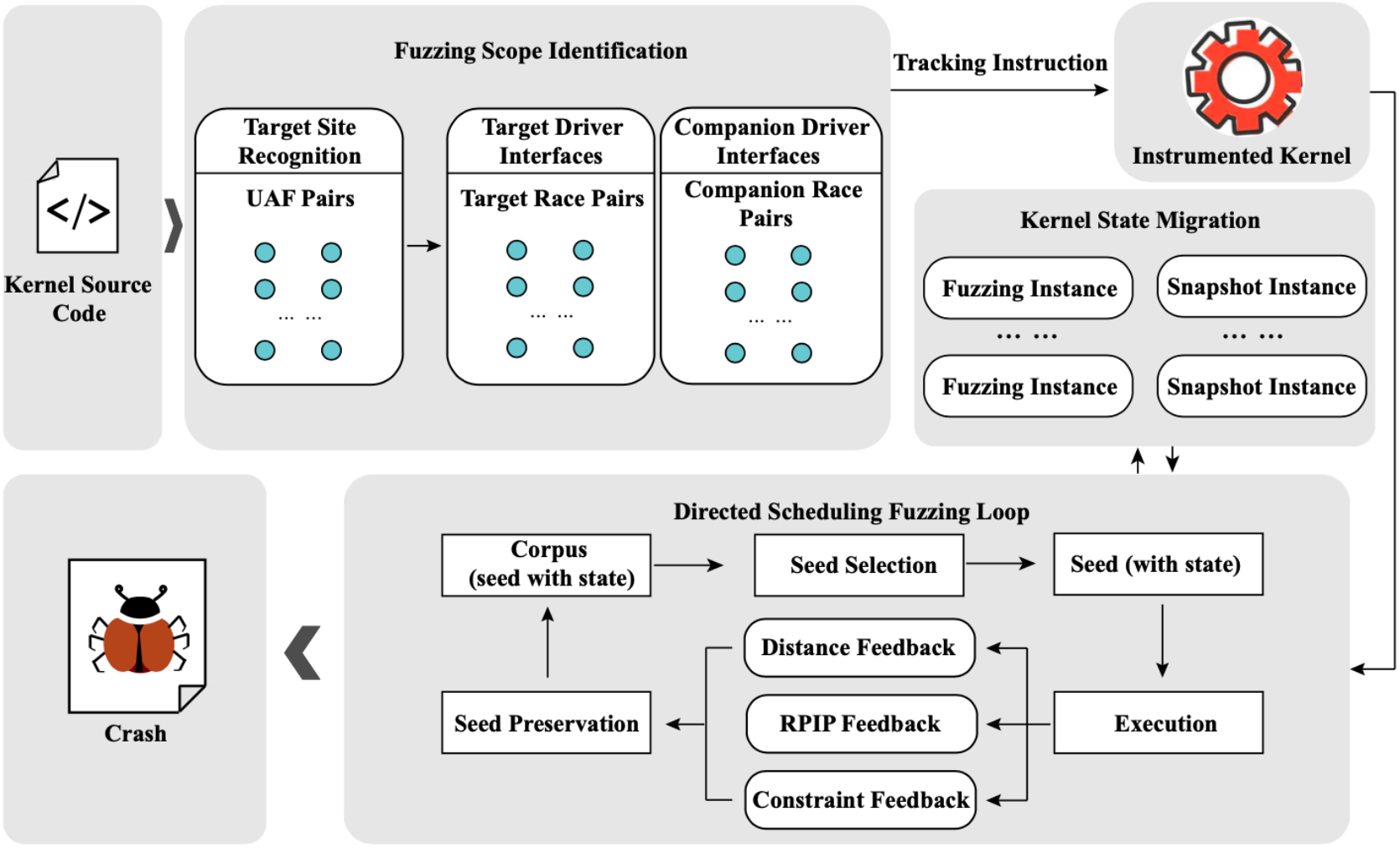


Figure 2: Workflow of DDRace.

Workflow of DDRace

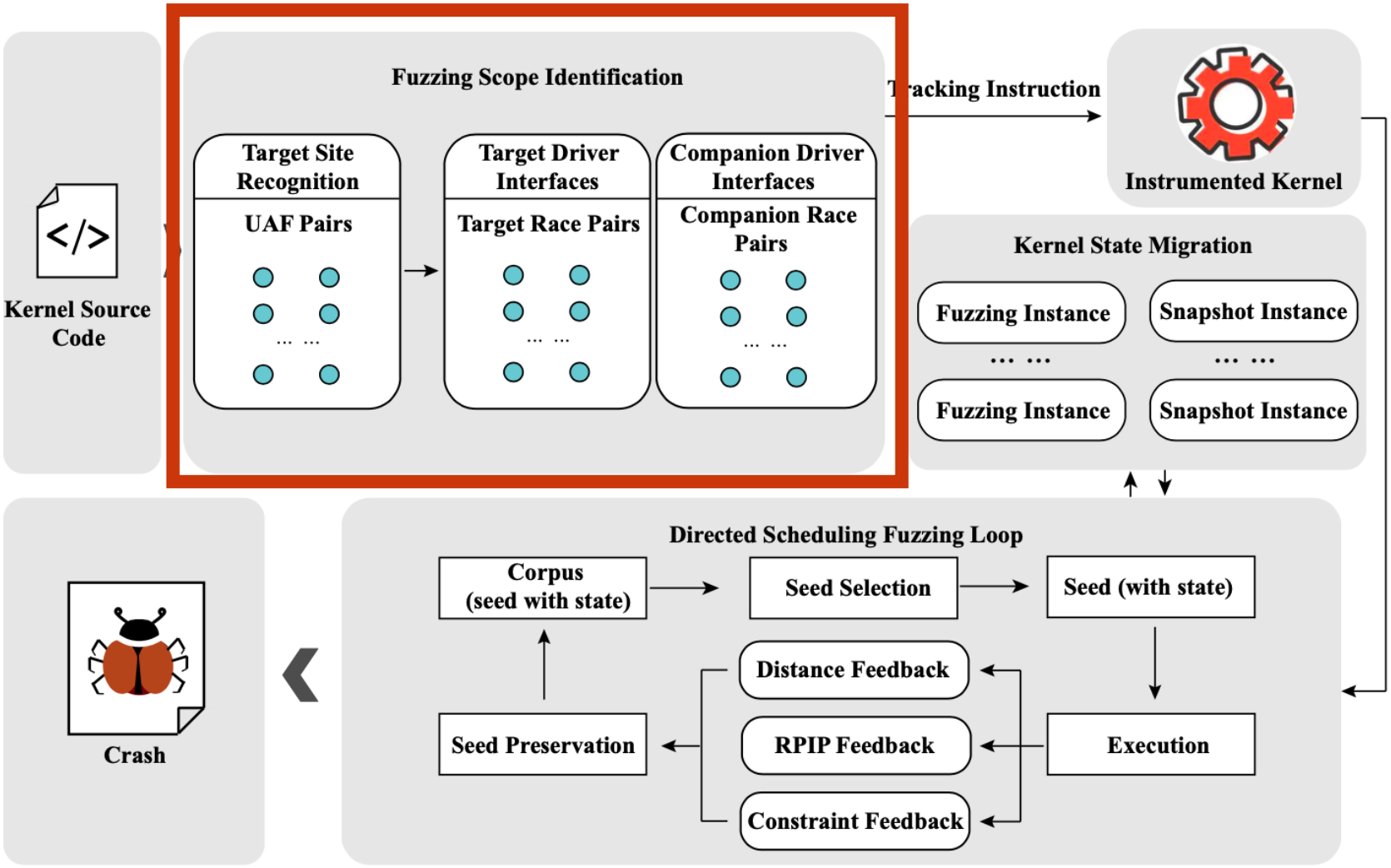


Figure 2: Workflow of DDRace.

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

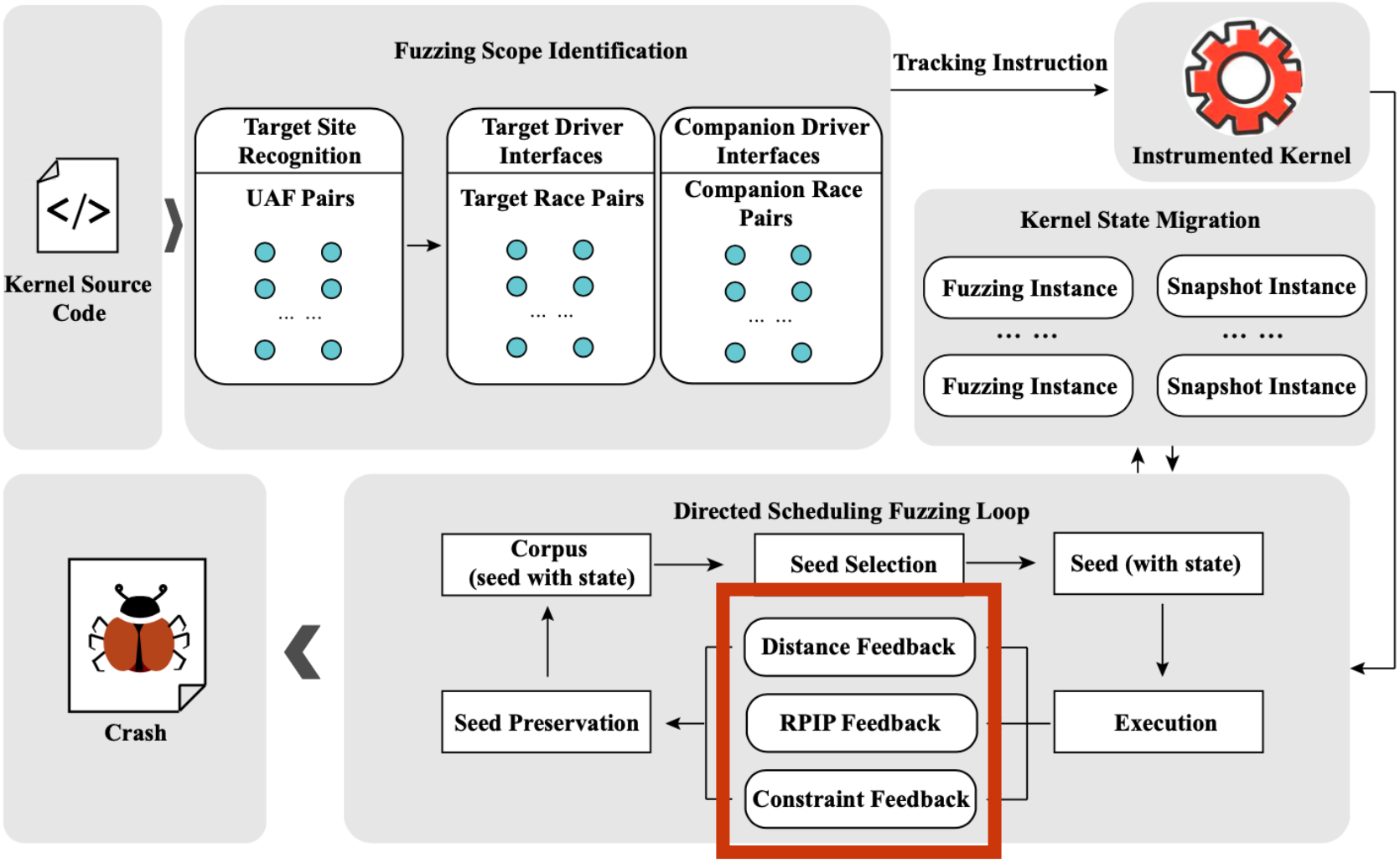


Figure 2: Workflow of DDRace.

Directed ファジング

・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）

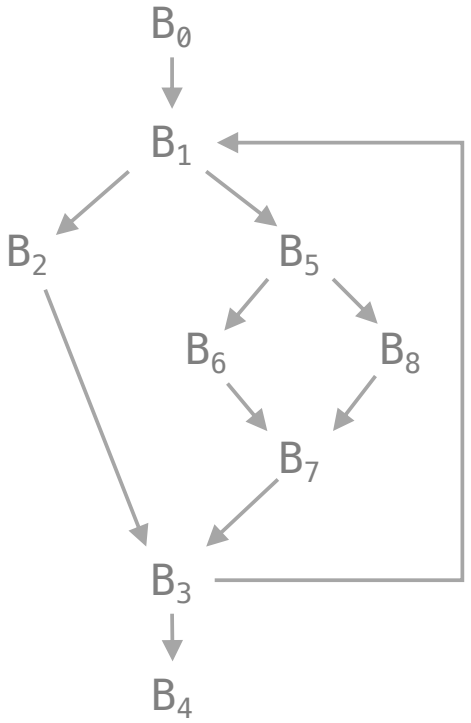
・脆弱性モデル制約距離（vulnerability model constraint distance）

・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

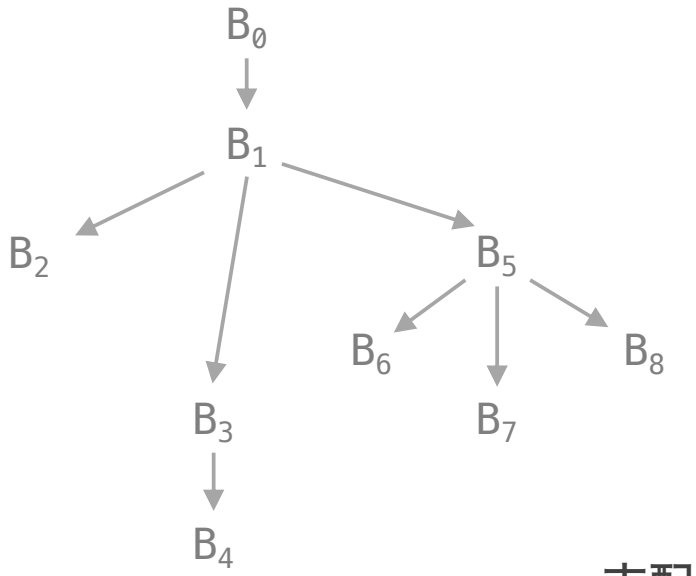
Dominator Depth Distance

・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

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