

論文紹介

Actor : Action-Guided Kernel Fuzzing
[SEC'23]

山本 航平

概要

ジャンル： カーネルファジング

従来のカーネルファザー：

- ・コードカバレッジの向上のみに注目
- ・特定の順序で発生するバグは単にコードを実行するだけでは不十分

提案手法：

- ・実行されるコードの **Action** (何を実行するか?) を考慮する
- ・ **Action** の順序 を考慮する

結果：

- ・Linux カーネルで 41個の未知のバグを発見 (15個は1日以内に発見)

カーネルファジング

- ・ システムコール (とその引数) の列を入力としたファジング

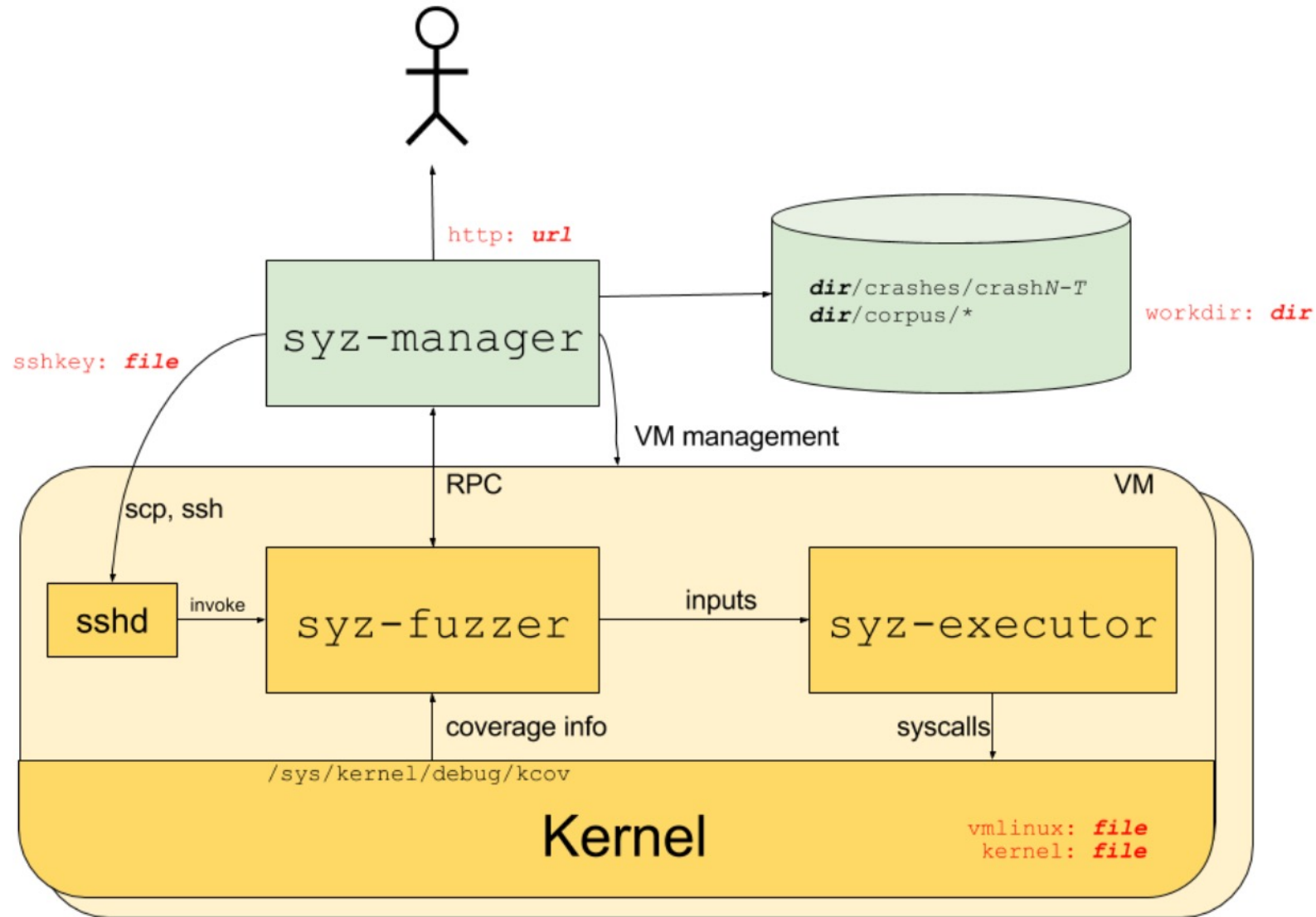
```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

- ・ サニタイザなどを用いてバグを検出
 - ・ KASAN (Address) [2] : 動的なメモリ安全性違反検知器
 - ・ KCSAN (Concurrency) [3] : 動的なレース検知器

[2] <https://www.kernel.org/doc/html/latest/dev-tools/kasan.html>

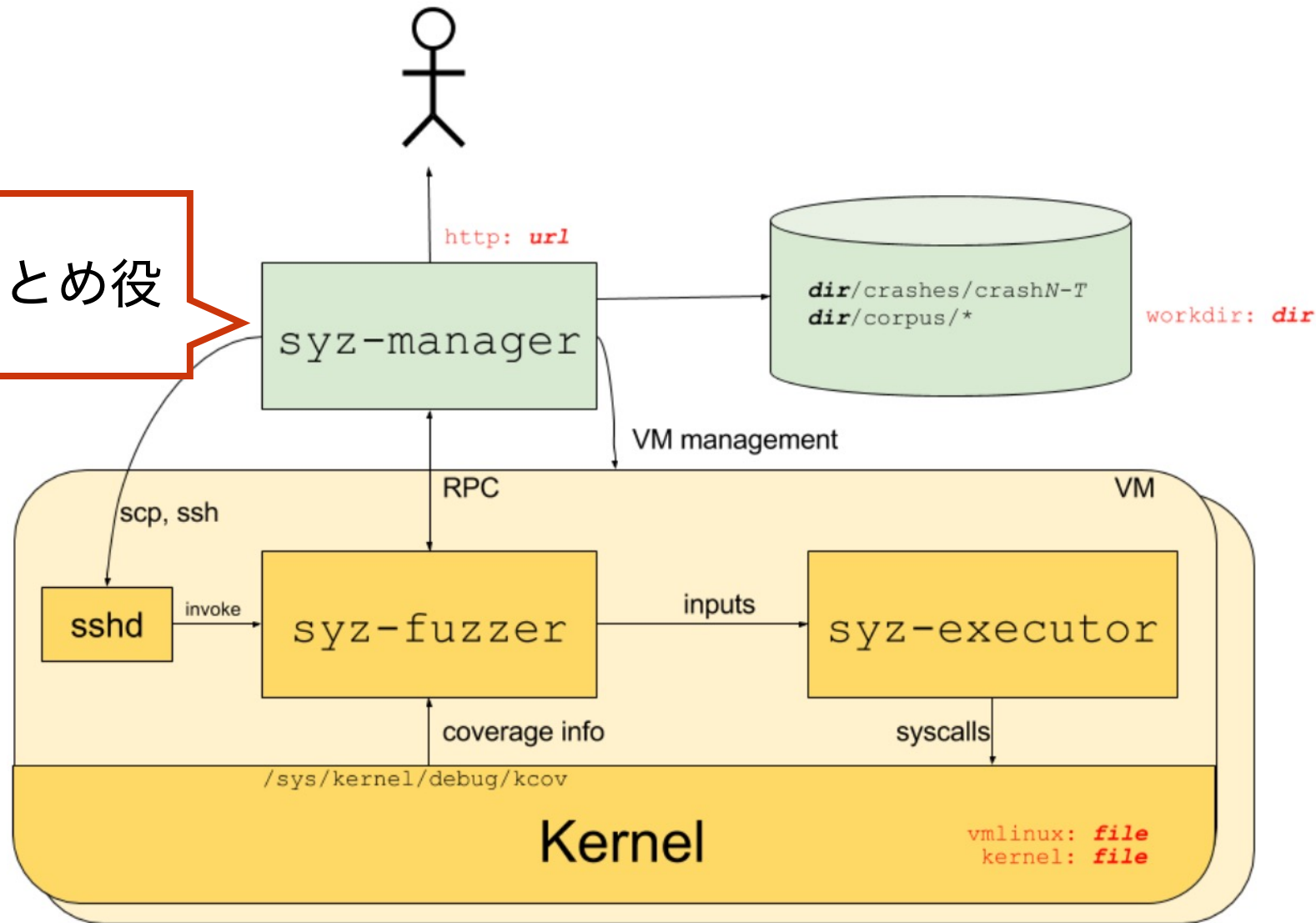
[3] <https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>

Syzkaller [1]

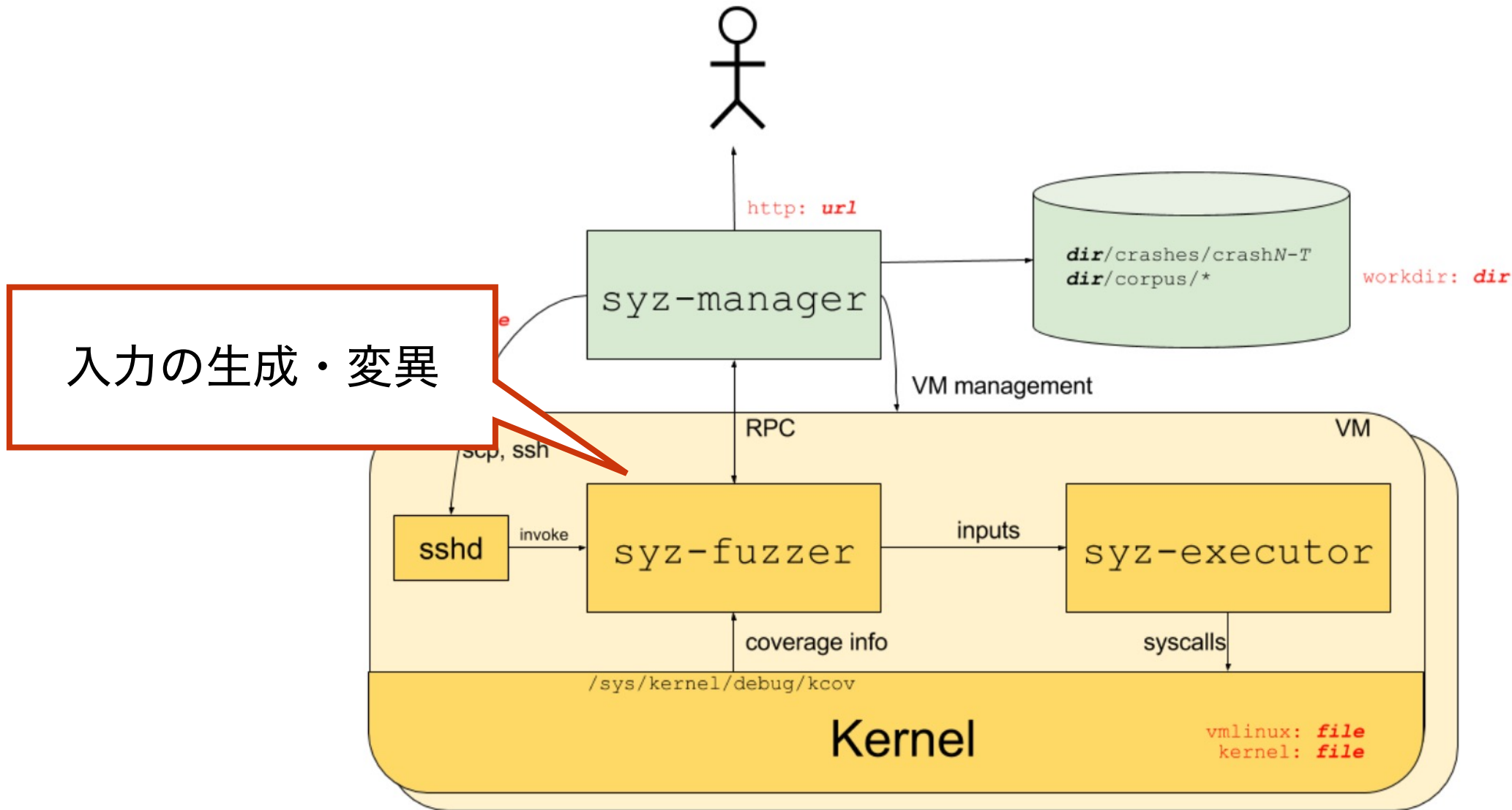


Syzkaller [1]

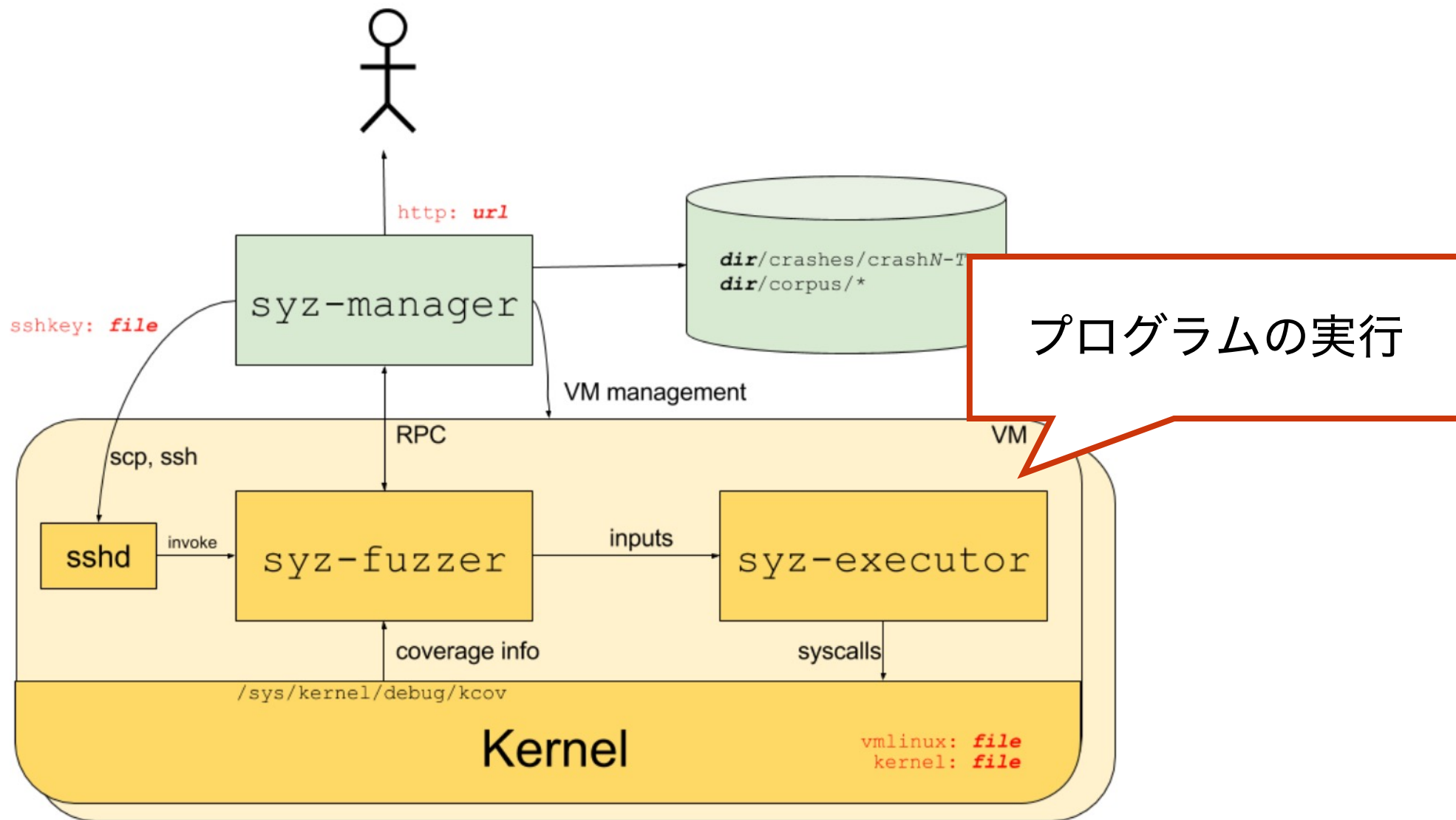
複数 VM のまとめ役



Syzkaller [1]



Syzkaller [1]



従来手法の問題点

- ・カーネルファザーの多くは **コードカバレッジの向上** を目指している
- ・カバレッジだけではバグの発見に非効率な場合がある
 - ・あるバグの発見にはプログラムの意味を考慮した **順序** が必要

ex.) Use-After-Free



Alloc → Use → Free



Alloc → Free → Use

Example

Use-After-Free

```
1  fd = openat(0, "...", ...)  
2  write(fd, &{CREATE}, ...)  
3  write(fd, &{DESTROY}, ...)  
4  write(fd, &{LISTEN}, ...)
```

```
5  __rdma_create_id(...) {  
6      struct rdma_id_private *id_priv;  
7      ...  
8      id_priv = kzalloc(sizeof *id_priv, ...);  
9      ...  
10 }
```

```
16  cma_listen_on_all(...) {  
17      ...  
18      list_add_tail(&id_prev->list, ...)  
19      ...  
20 }
```

```
11  rdma_destroy_id(...) {  
12      ...  
13      kfree(id_priv);  
14      ...  
15 }
```

Example

Use-After-Free



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{DESTROY}, ...)  
4 write(fd, &{LISTEN}, ...)
```



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

Example

Use-After-Free



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{DESTROY}, ...)  
4 write(fd, &{LISTEN}, ...)
```



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

カバレッジベースのファザー：

- openat → write の順番は考慮

→ openat の返値が write の引数に使われるため、

カバレッジに寄与する可能性大

Example

Use-After-Free



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{DESTROY}, ...)  
4 write(fd, &{LISTEN}, ...)
```



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

カバレッジベースのファザー：

- write の順番は考慮しない

→ カバレッジの増加は期待できないから

Example

Use-After-Free



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{DESTROY}, ...)  
4 write(fd, &{LISTEN}, ...)
```



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```



UAF を見逃すかも

Example

Use-After-Free



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{DESTROY}, ...)  
4 write(fd, &{LISTEN}, ...)
```



```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

- write の順番も考慮して UAF を引き起こすプログラムを生成したい

Motivation

カバレッジガイドファザーより効率的にバグを発見するため

(1) システムコール (とその引数) が **何を実行するか** を考慮する

- ・ `write(fd, &{CREATE}, ...)` : `alloc`
- ・ `write(fd, &{LISTEN}, ...)` : `write`
- ・ `write(fd, &{DEATROY}, ...)` : `dealloc`

(2) バグを誘発する **順序** を考慮する

- ・ Use-after-free: `alloc` → `dealloc` → `read/write`
- ・ Double-free: `alloc` → `dealloc` → `dealloc`

Actor

Actor の2つのフェーズ

(1) システムコール (とその引数) が 何を実行するか を考慮する

→ **Action mining**

Action : 何を実行するか (alloc, read, ...)

Dart : システムコール (とその引数) と action の関係

(2) バグを誘発する 順序 を考慮する

→ **プログラム合成**

Action Mining

(1) システムコール (とその引数) が 何を実行するか を考慮する

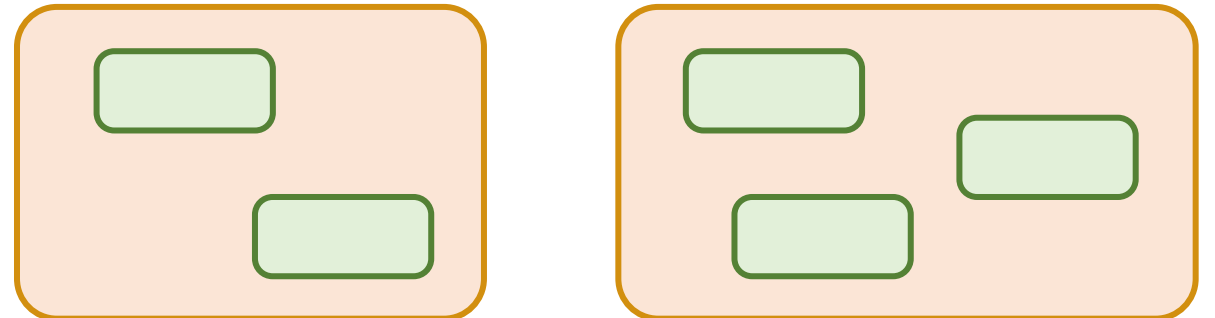
- ・ システムコールと action の関係 (dart) を動的に収集

→ カバレッジガイド
戦略を用いる



```
write(fd, &{CREATE}, ...) : alloc
```

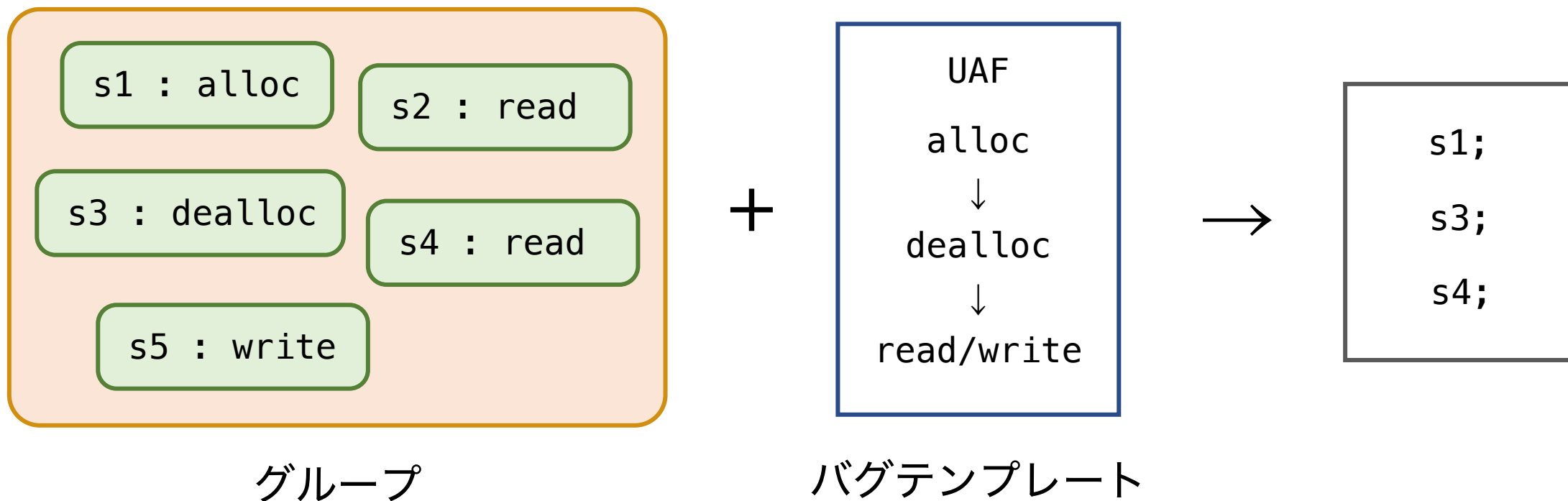
- ・ 同じメモリ領域を操作しそうな dart をグループ化



プログラム合成

(2) バグを誘発する 順序 を考慮する

- ・ (1) で取得したグループとバグテンプレートからプログラムを合成



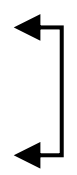
Actor

Actor の2つのフェーズ

(1) Action mining

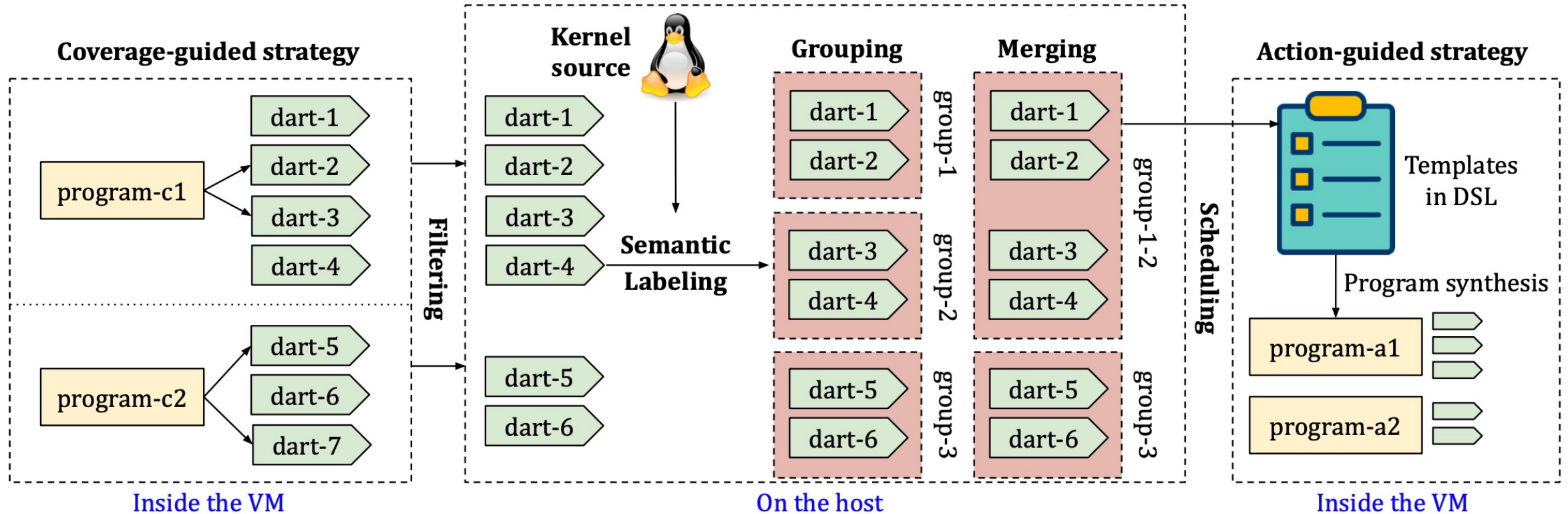
(2) プログラム合成

→ これらはカバレッジガイドファザーの中に組み込まれている

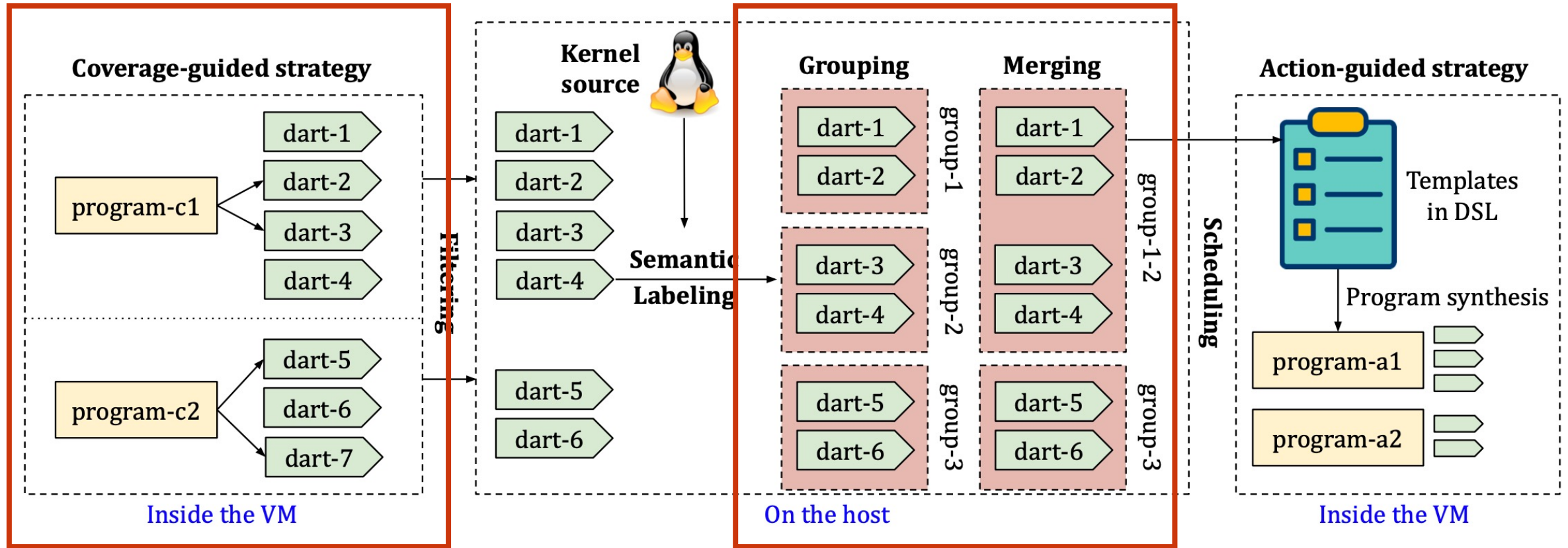
- ・カバレッジガイド的入力生成・変異
 - ・アクションガイド的入力生成・変異
- 
- ランダム

(1) → (2) の順番で明確に区別されるわけではない

Actor Workflow



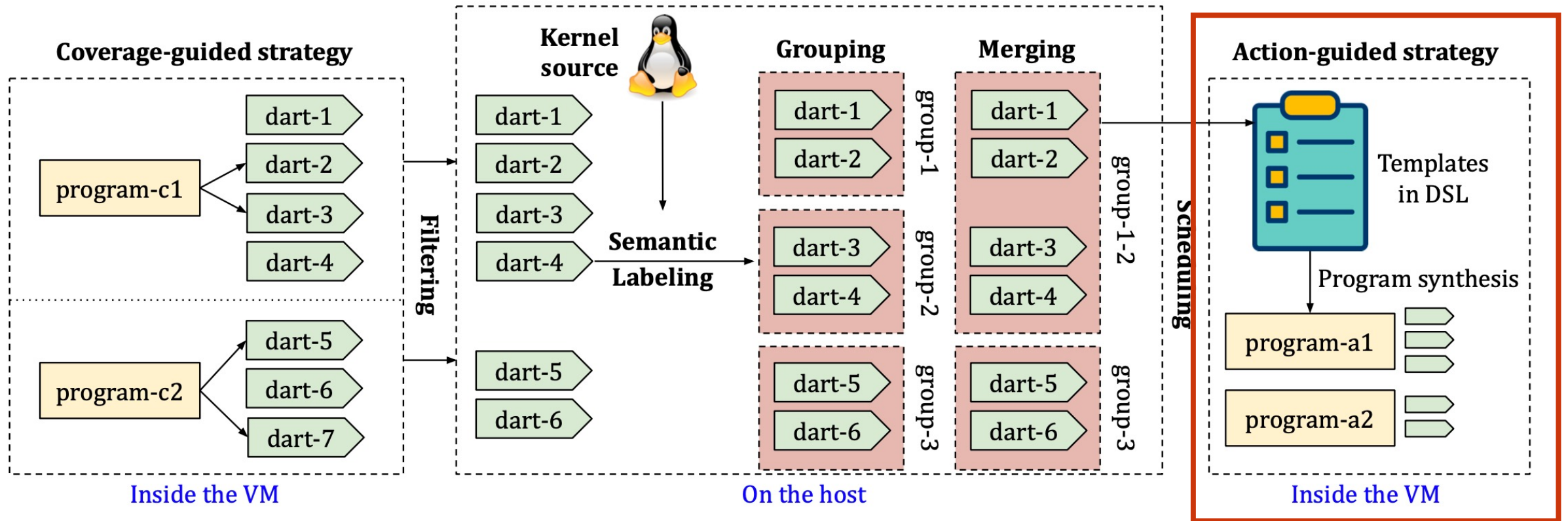
Actor Workflow



dart 収集

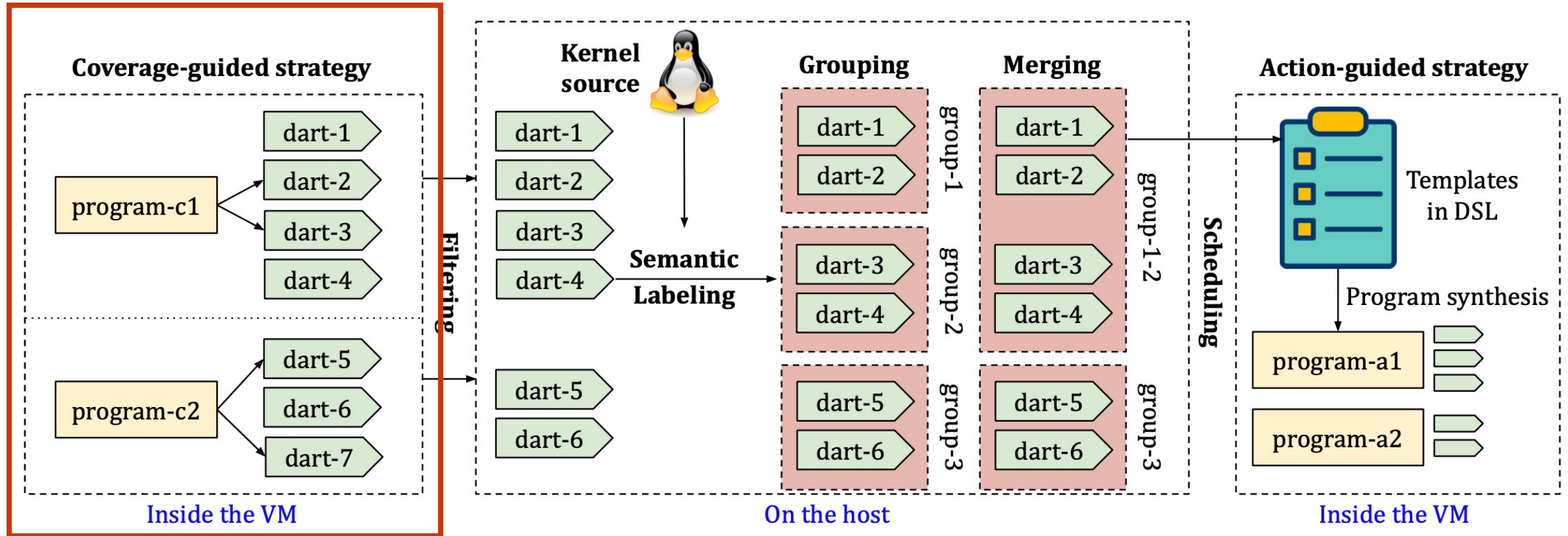
dart グループ化

Actor Workflow



プログラム合成

Actor Workflow



Dart

- ・ システムコールが何を実行するかを表したもの
- ・ Dart が持つ情報
 - ・ システムコール (とその引数)
 - ・ Action Type
 - ・ 操作するメモリ (アドレス, サイズ)
 - ・ (アクションが記録された時の) スタックトレース

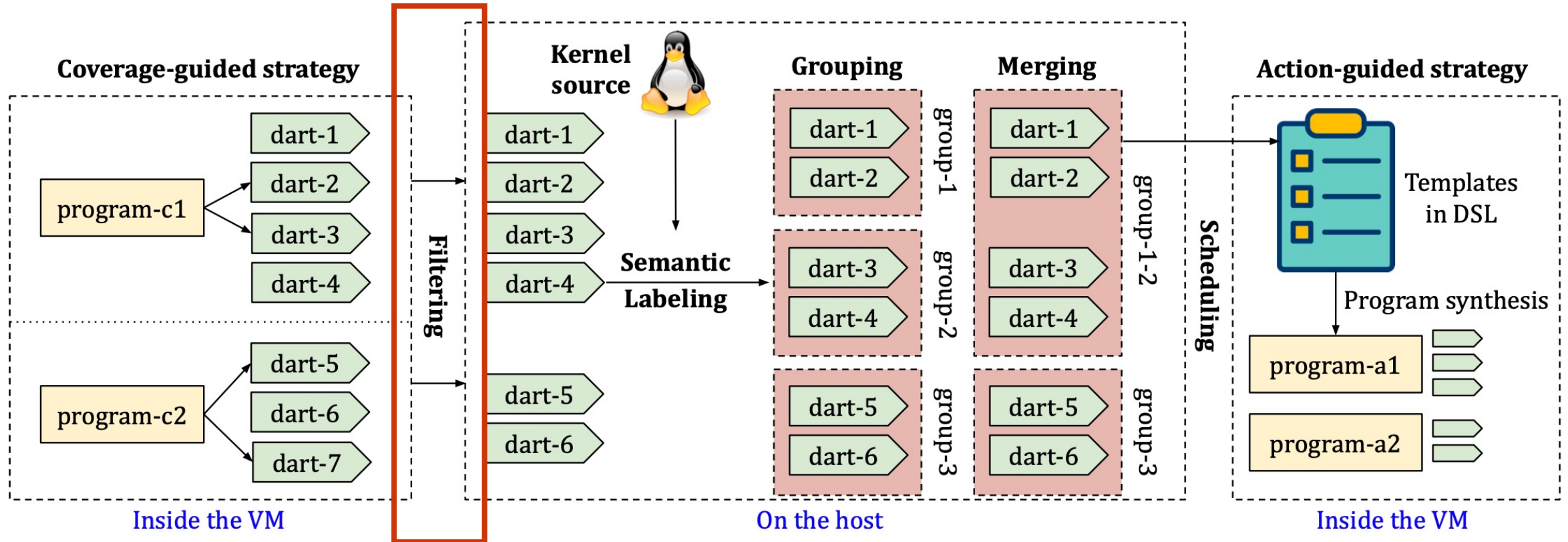
Dart

- ・ システムコールが何を実行するかを表したもの
- ・ Dart が持つ情報

- ・ システムコール (とそれを実行するコード)
- ・ Action Type
- ・ 操作するメモリ (アドレス)
- ・ (アクションが記録されたメモリ)

- ・ allocation ・ deallocation
- ・ value read/write
- ・ pointer read/write
- ・ index read/write

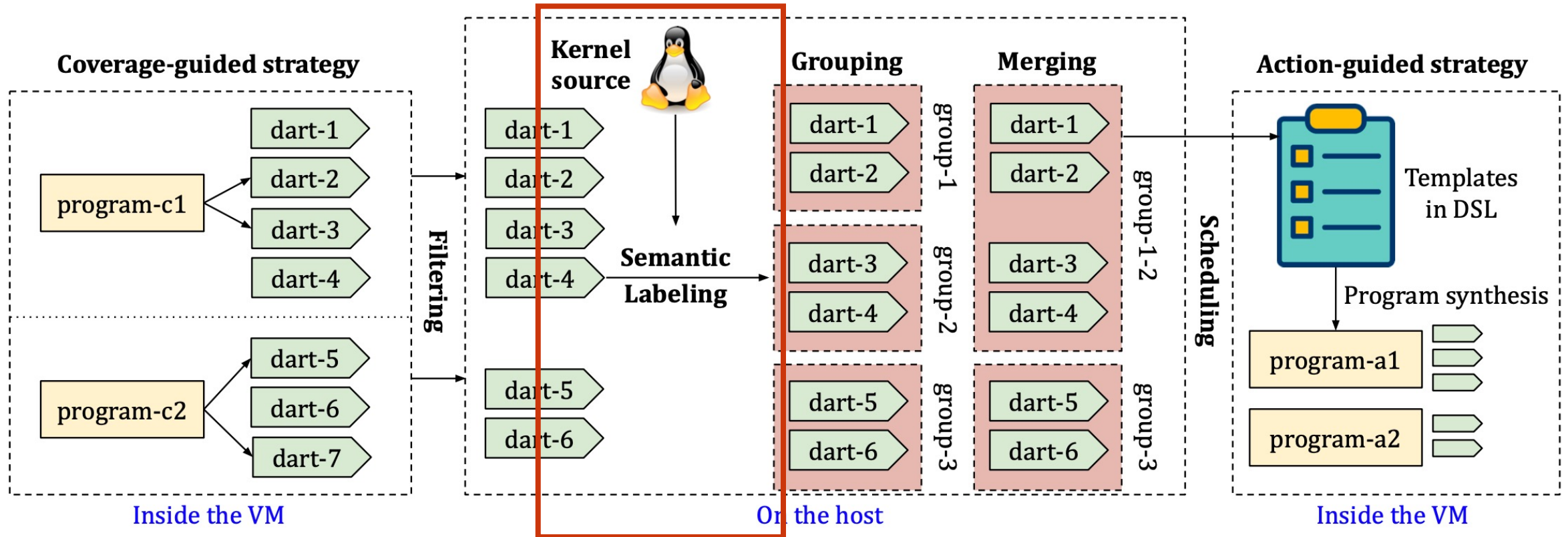
Actor Workflow



Dart Reduction

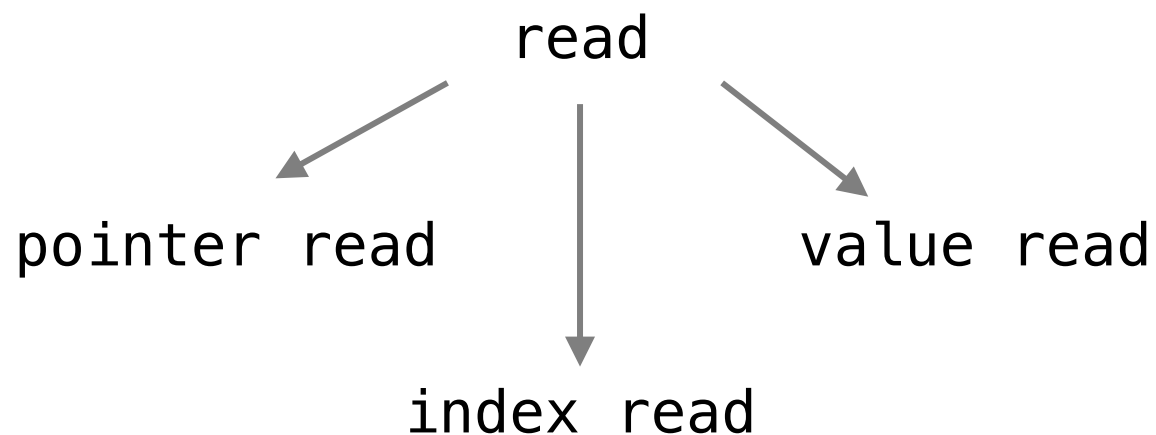
- ・ プログラムが大量の dart を生成する可能性があるため削減したい
- ・ 問題点
 1. VM からホストへの dart の転送オーバーヘッドが増加
 2. 効果的なプログラム合成が難しくなる
- ・ 削減方法
 1. それまでに alloc アクションによって割り当てられたメモリ領域を操作する dart のみを収集
 2. システムコールごと, 同じメモリ領域ごとに最初の read/write のみを記録

Actor Workflow



Dart Labeling

- dart の収集で得られるのはメモリの読み書きのみ
 - ポインタ, index, 値 のどの読み書きなのかは分からない
- ソースコードを静的解析して dart のアクションタイプを洗練させる



index read

```
1 i = S.f;  
2 arr[i];
```

Dart Labeling

Index Access かどうか？

- Heap の値が index に使われる時, 構造体のフィールドの場合が多い
(primitive 型を heap 上に確保するのは稀)
- Step1 : index としてアクセスされる構造体のフィールドを特定
→ `S.f`

```
1  S.f = 5;  
2  i = S.f;  
3  arr[i];
```

Dart Labeling

Index Access かどうか？

- Heap の値が index に使われる時, 構造体のフィールドの場合が多い
(primitive 型を heap 上に確保するのは稀)
- Step2 : その構造体のフィールドにアクセスする命令を特定

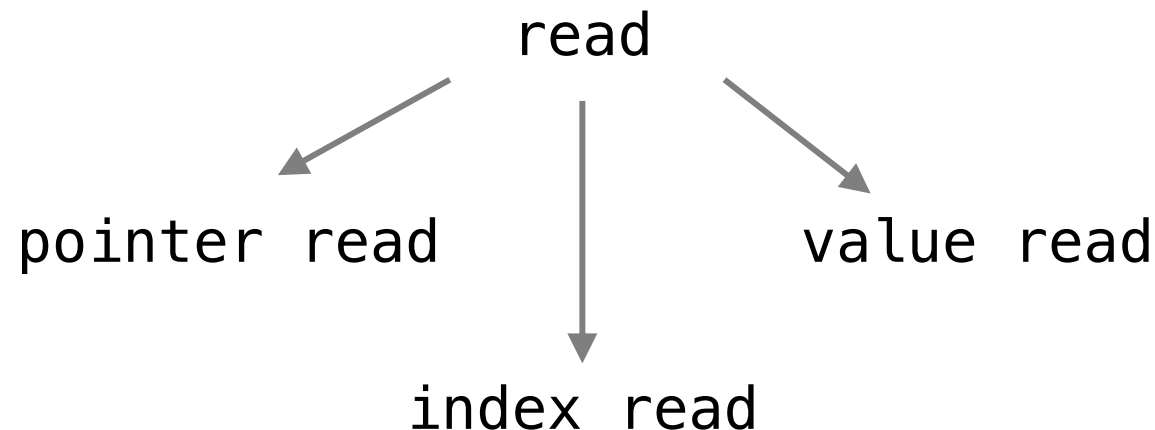
index write

```
1 S.f = 5;  
2 i = S.f;  
3 arr[i];
```

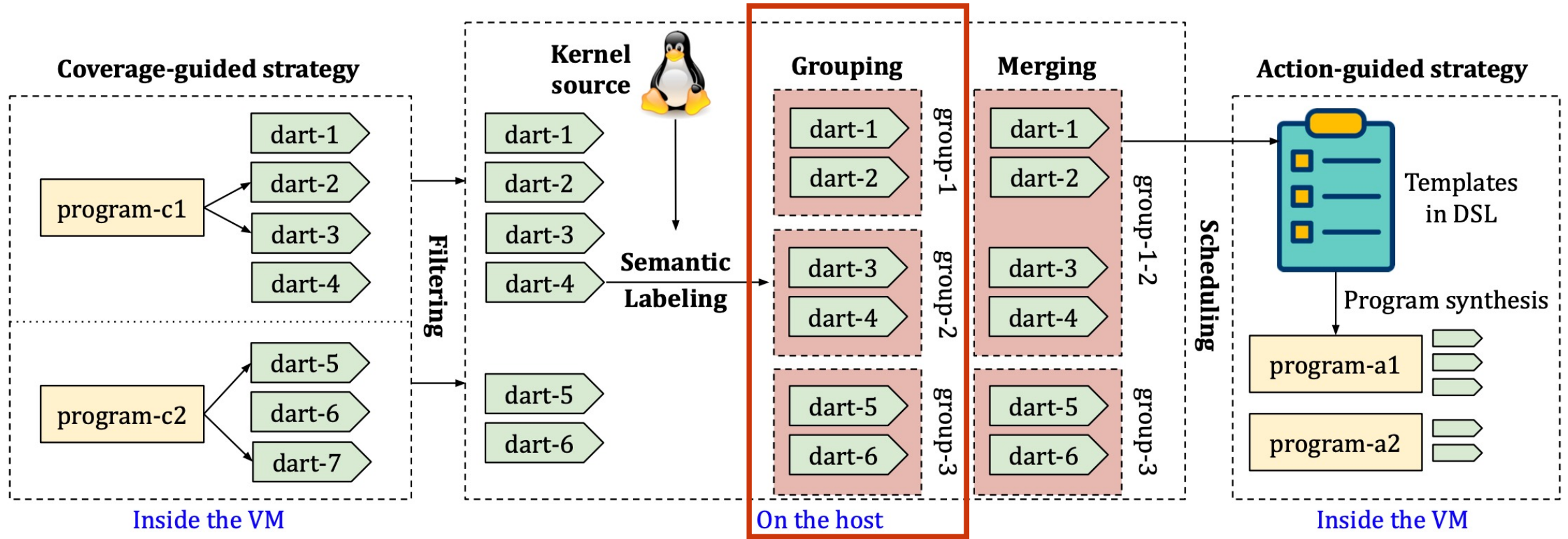
index read

Dart Labeling

- ・ 静的解析で各命令が index read/write か pointer read/write か分かった
- ・ dart のアクションタイプの洗練方法
 - ・ デバッグ情報を用いて dart のスタックトレースから命令を復元
 - ・ その命令が index/pointer read/write なら, そのように更新
 - ・ どちらでもないなら value read/write に更新
- ・ この静的解析は事前に一度だけ行えばよい

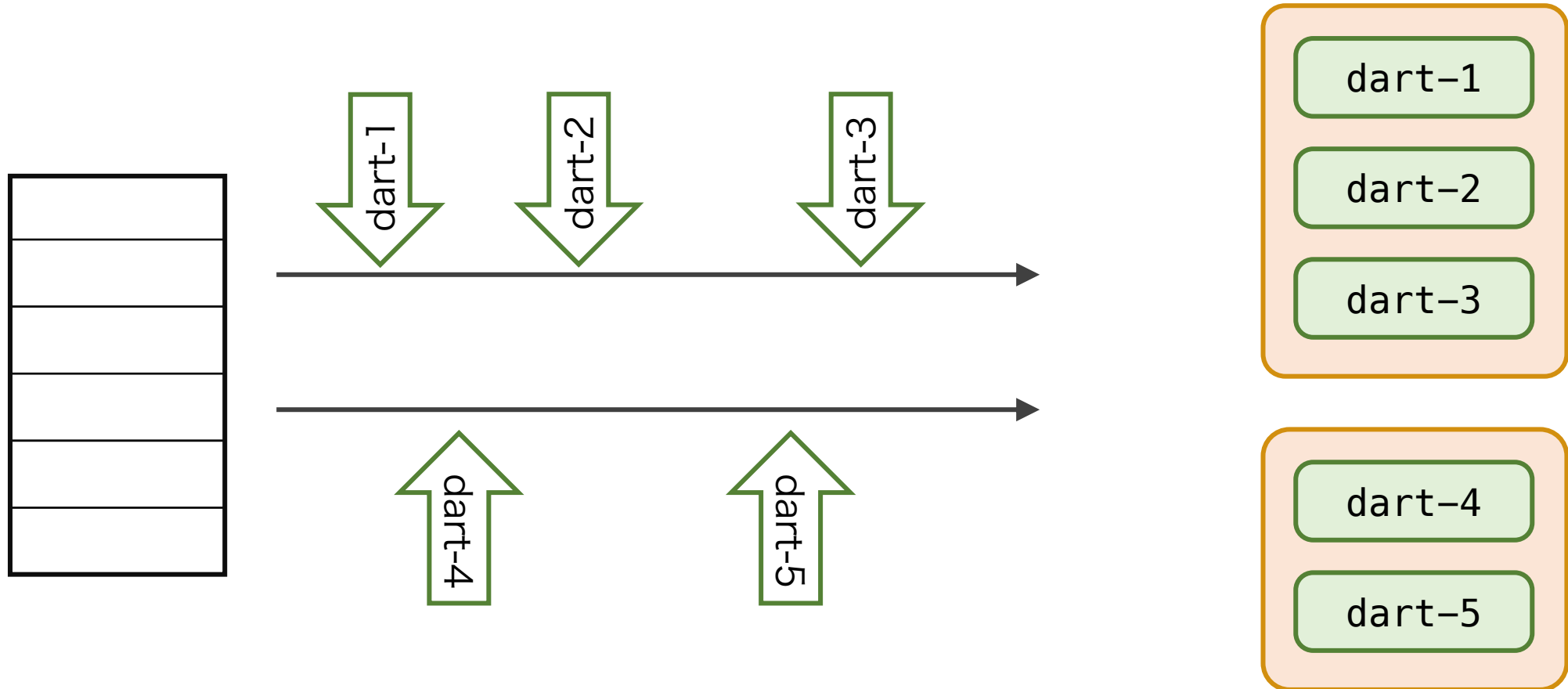


Actor Workflow

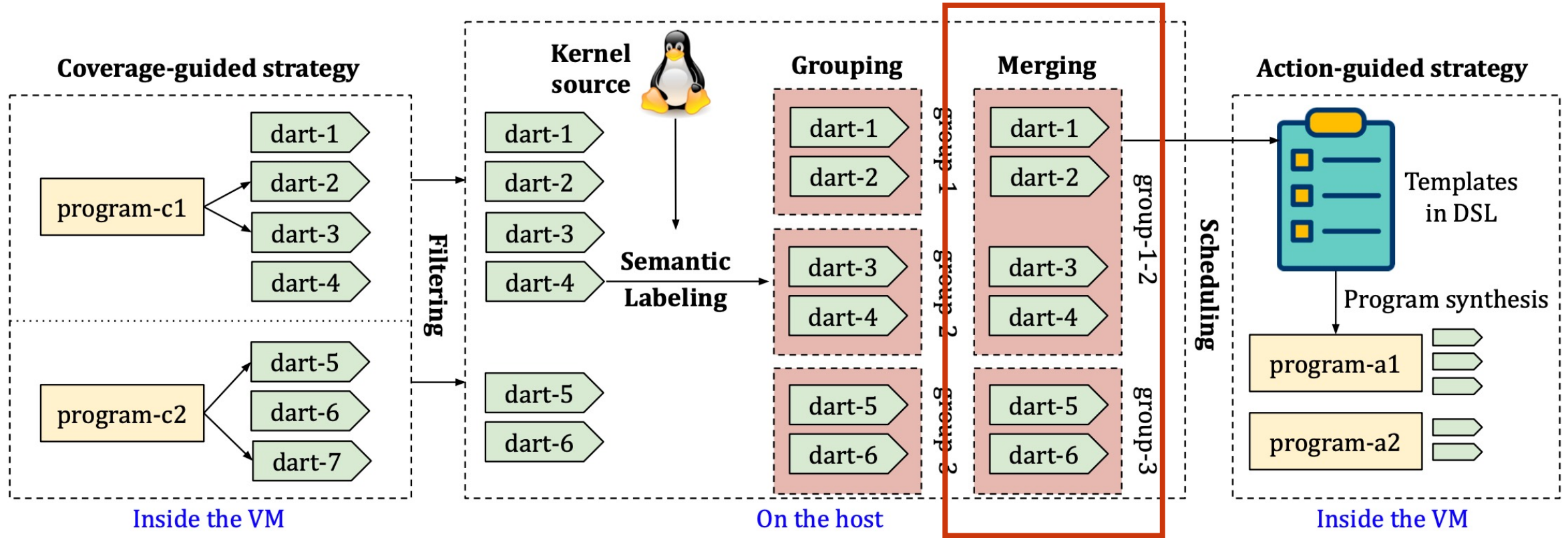


Dart Grouping

- 1つのプログラムについて同じメモリ領域を操作する dart をグループ化
(dart は操作するメモリの情報を持つ)

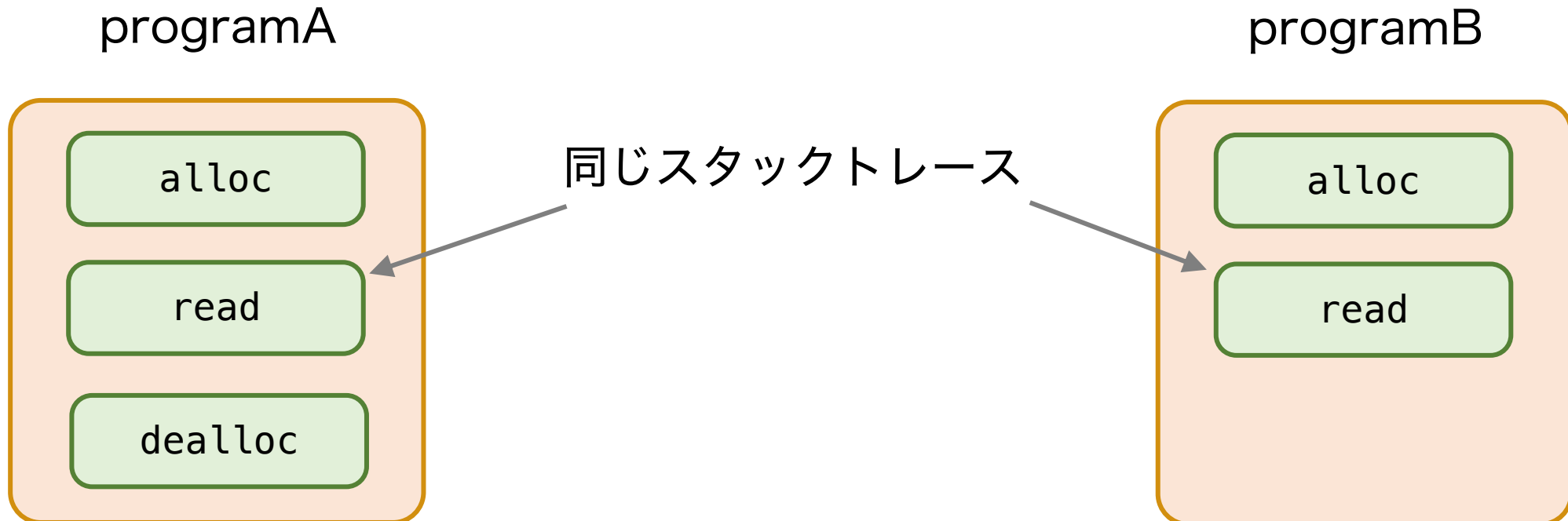


Actor Workflow



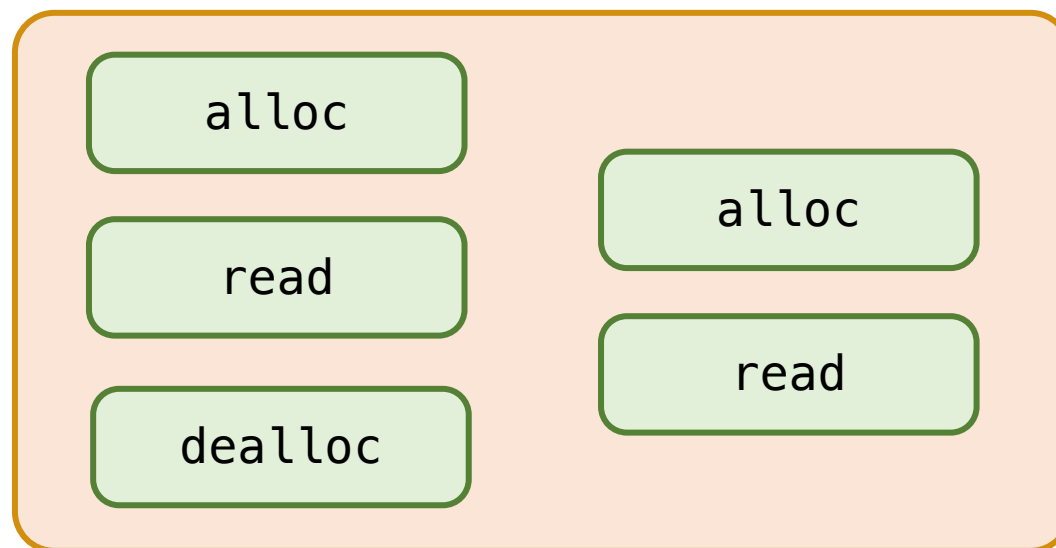
Dart Merging

- ・ 複数のプログラムから得られる dart グループを結合
- ・ 2つのグループで同じスタックトレース + アクションタイプを持つ
dart が存在するなら2つのグループをマージ



Dart Merging

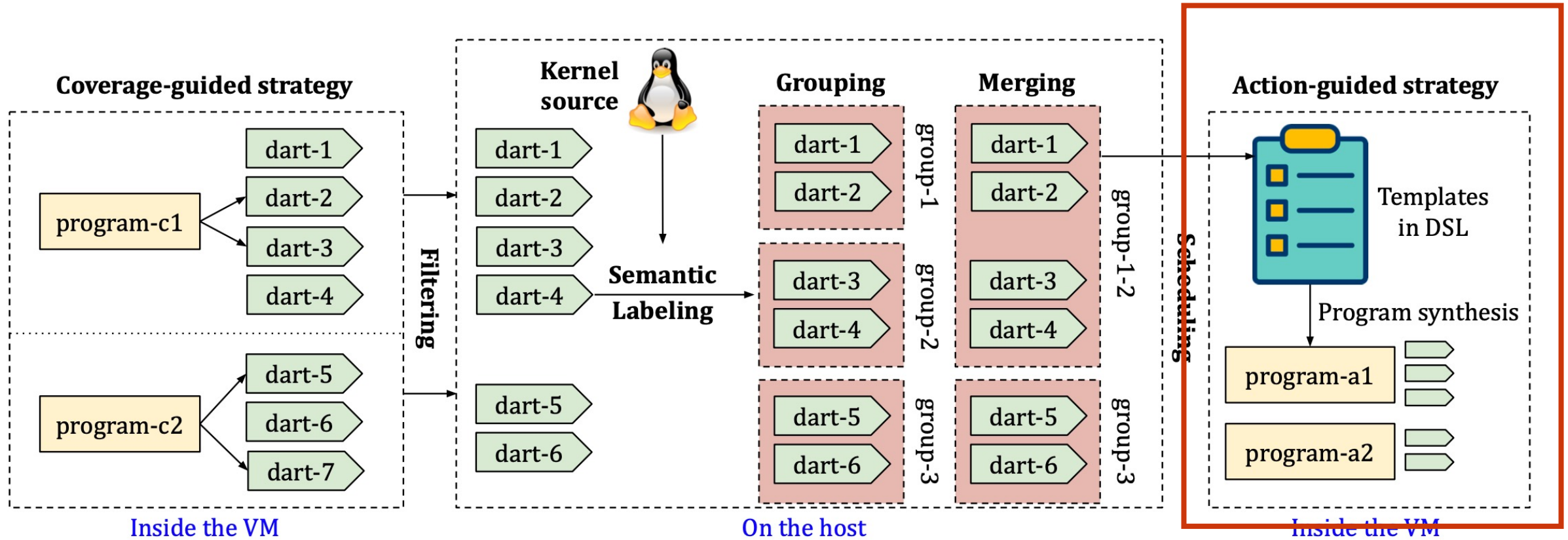
- ・ 複数のプログラムから得られる dart グループを結合
- ・ 2つのグループで同じスタックトレース + アクションタイプを持つ
dart が存在するなら2つのグループをマージ



Dart Merging

- ・ 複数のプログラムから得られる dart グループを結合
- ・ 2つのグループで同じスタックトレース + アクションタイプを持つ
dart が存在するなら2つのグループをマージ
- ・ 意味的な類似性 (semantic similarity) があるとき
2つのシステムコールは関連があると考え

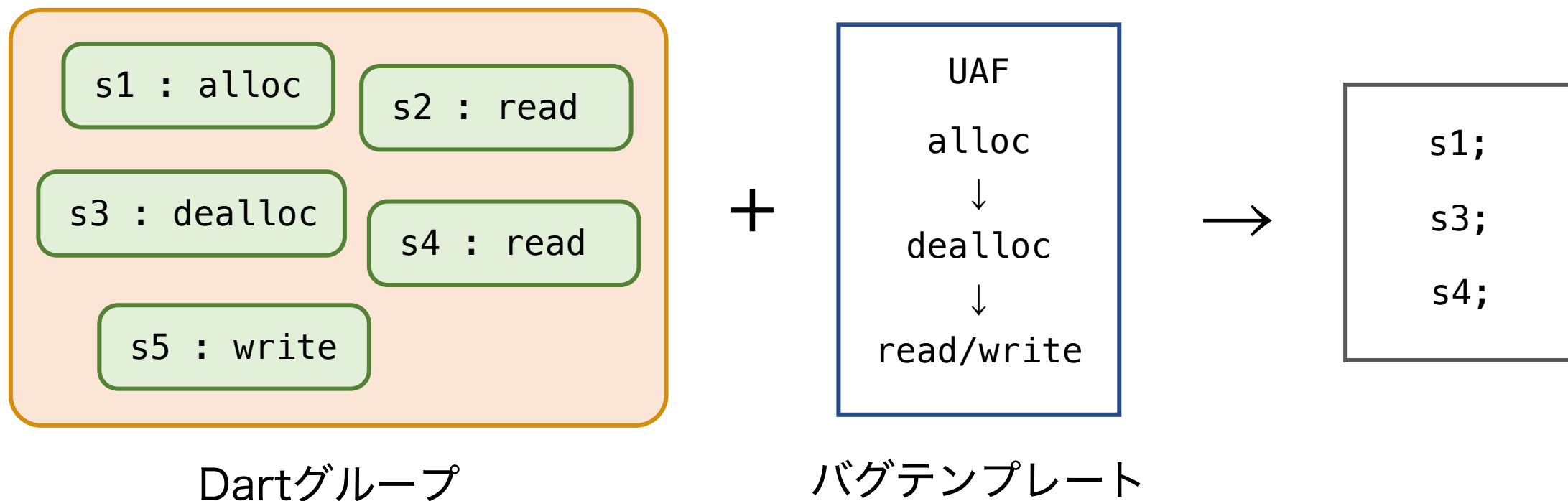
Actor Workflow



プログラム合成

(2) バグを誘発する 順序 を考慮する

- Dart グループとバグテンプレートからプログラムを合成



バグテンプレート

Bug class	Template	Bug class	Template
Use After Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow [\mathcal{A}_r \mathcal{A}_w]$	Null Ptr Deref	$\mathcal{A}_a^x \rightarrow \mathcal{A}_d^x$
Double Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow \mathcal{A}_d$	Invalid Free	\mathcal{A}_d
Out of Bounds (1)	$\mathcal{A}_a \rightarrow \mathcal{A}_{iw}^* \rightarrow \mathcal{A}_{ir}$	Memory Leak (1)	\mathcal{A}_a^*
Out of Bounds (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw}^* \rightarrow \mathcal{A}_{pr}$	Memory Leak (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw} \rightarrow \mathcal{A}_d$
Uninitialized Read	$\mathcal{A}_a \rightarrow \mathcal{A}_r$	-	-

Table 1: The bug templates defined by ACTOR

バグテンプレート

Out of Bounds (1)

`alloc` → `index write*` → `index read`

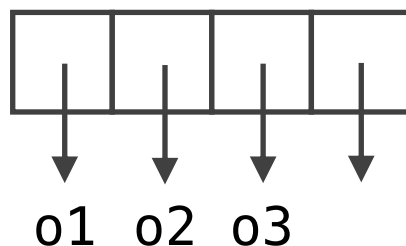
- `index` はループの中でインクリメントされることが多い
- `index write` を複数回繰り返すことで `array` の長さを超えることを期待

バグテンプレート

Null Ptr Deref

`alloc * x → dealloc * x`

- kernel 内の配列はオブジェクトを指すポインタとそのサイズを表す変数 `c` を保持することが多い
- `alloc` で失敗して `c` が誤って更新されることを期待



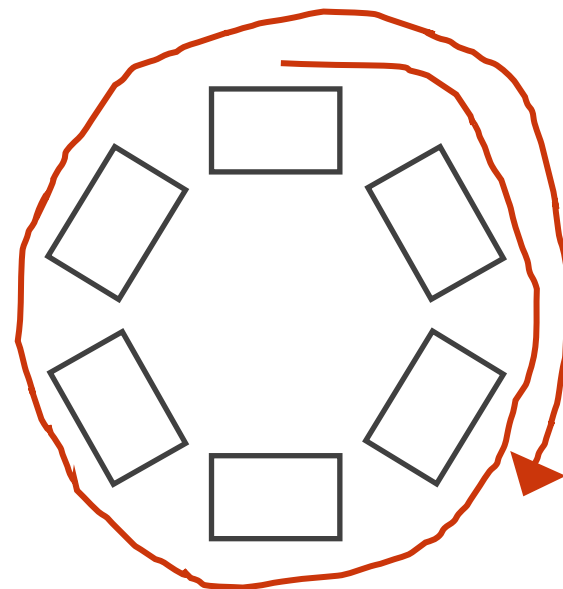
`c = 4`

バグテンプレート

Memory Leak (1)

alloc*

- ・ リングバッファなど固定サイズのバッファで
ポインタが上書きされることを期待

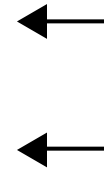


Domain-specific Language

- ・ バグテンプレートは DSL を用いて指定される
- ・ 解析者が追加のバグテンプレートを簡単に指定できるようにする

プログラム合成

- ・カバレッジガイド的入力生成・変異
- ・アクションガイド的入力生成・変異



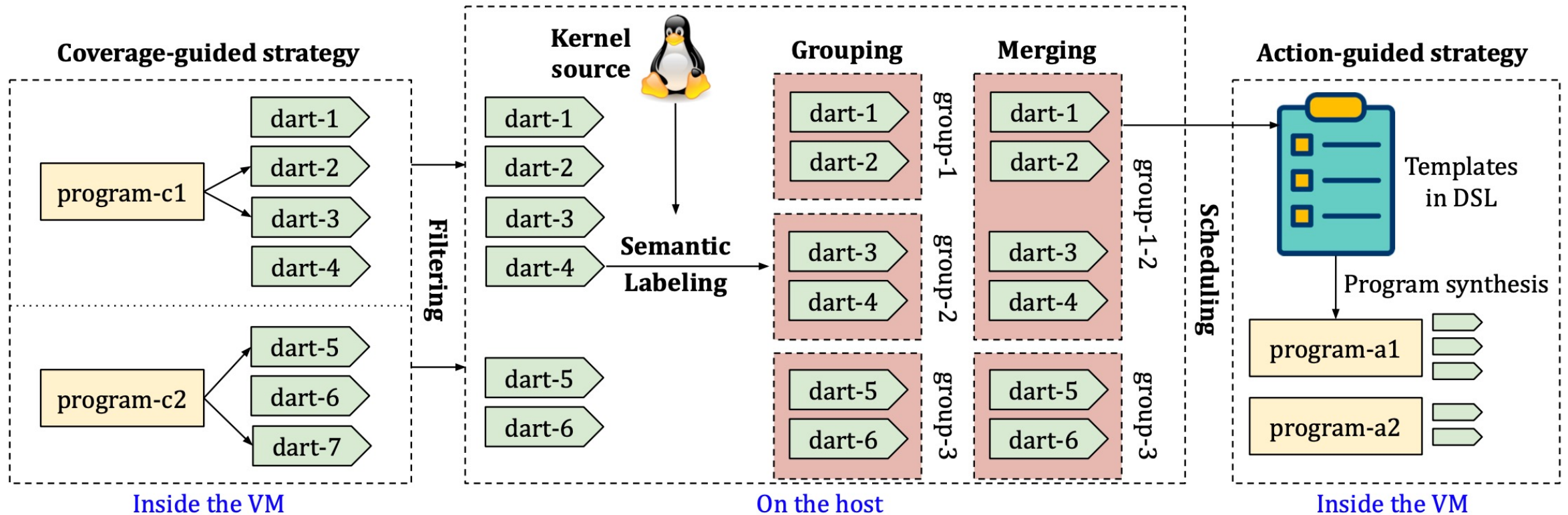
ランダム (確率 : 0.5)

- ・どのバグテンプレートを使用するか
- ・ Dart グループの選び方
- ・ どの Dart を選択するか



→ ランダム

Actor Workflow





中山競馬場

実装

- ・ Syzkaller 上に実装
- ・ Dart labeling の静的解析は LLVM のパス
- ・ Action 記録のために Linux カーネルモジュールの開発,
KASAN の修正

Action の記録

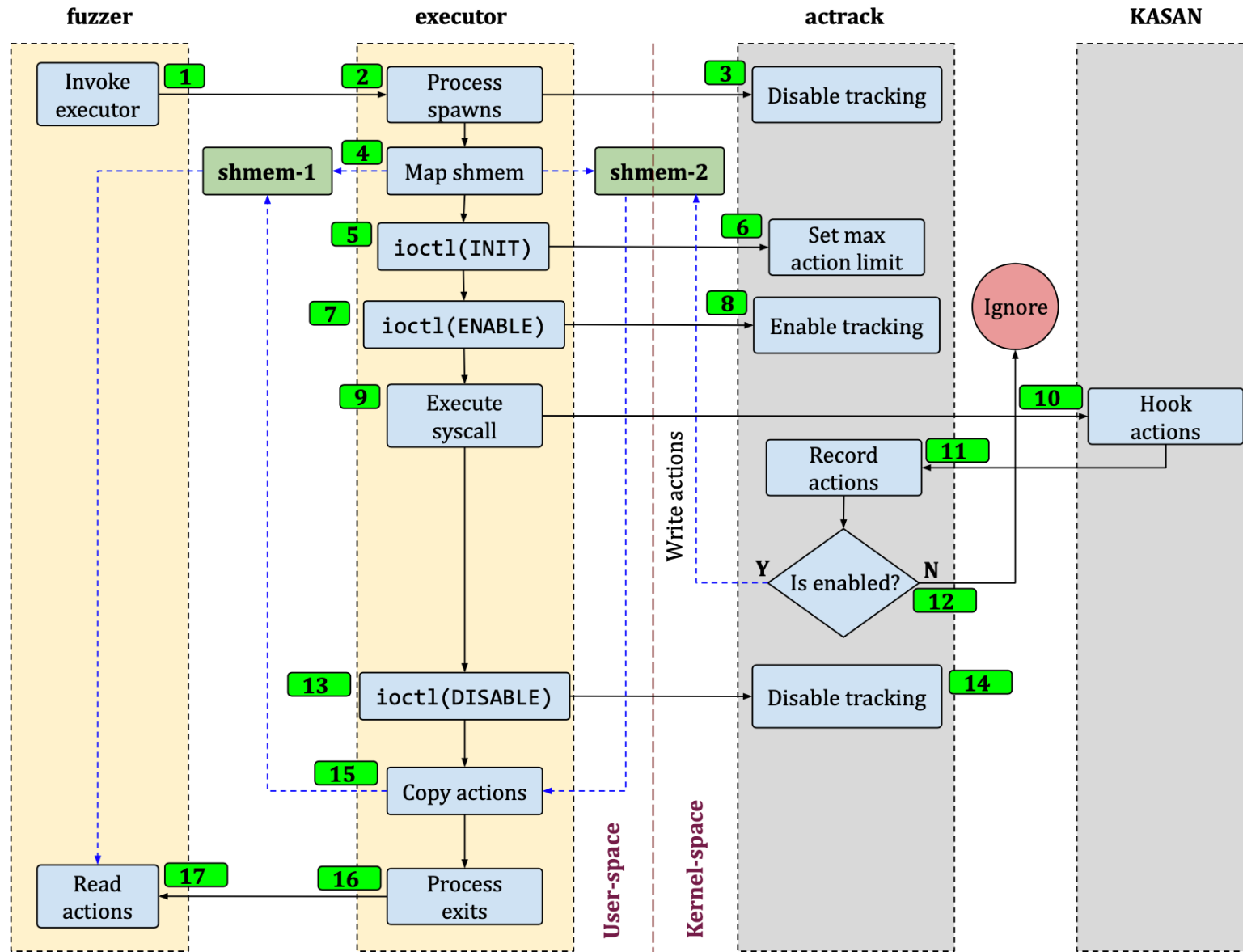
- ・ KASAN の2種類のフック
 1. カーネルメモリアロケータ API を計装
 2. メモリアクセスの前に計装
 - ・ フックを用いて自作のカーネルモジュール `actrack` を呼び出す
 - ・ アクセスのアドレス
 - ・ アクセスタイプ (`alloc/free/read/write`)
- などを `actrack` に渡す

Action の記録

カーネルモジュール actrack

- ・ システムコール入力に関連するアクションを収集
 - ・ 割り込み, スケジューラなどの非決定的アクションは収集しない

Action の記録



評価

RQ1 : Actor は新しいバグを発見できたか？

RQ2 : Dart は異なるカーネル状態でも同じ action を実行するか？

RQ3 : Actor は多くの共有アクセスをトリガーできるか？

RQ4 : Actor はバグを誘発するパターンのプログラムを
多く生成できるか？

RQ5 : Actor は Syzkaller の発見できない
システムコール間の関係を発見できるか？

RQ6 : Actor と既存のカーネルファザーとの比較

Syzkaller との比較

実験環境

- CPU : Intel(R) Xeon(R) E5-2690 v2 @ 3.00GHz
- メモリ: 256 GiB
- ホストOS : Ubuntu 20.04.4 LTS 64bit

実験環境 (RQ1)

- ・ 対象 Linux バージョン
 - ・ LTS (5.4.206, 5.10.131)
 - ・ stable release (5.19)
 - ・ latest release (6.2-rc5)
- ・ それぞれ 12 日間ファジング
- ・ 4GiB RAM, 2つの CPUコアをもつ 4つの VM を使用
- ・ KCOV, KASAN, KMSAN, KMEMLEAK を有効に

RQ1: New Bug Discovery

RQ1 : Actor は新しいバグを発見できたか？

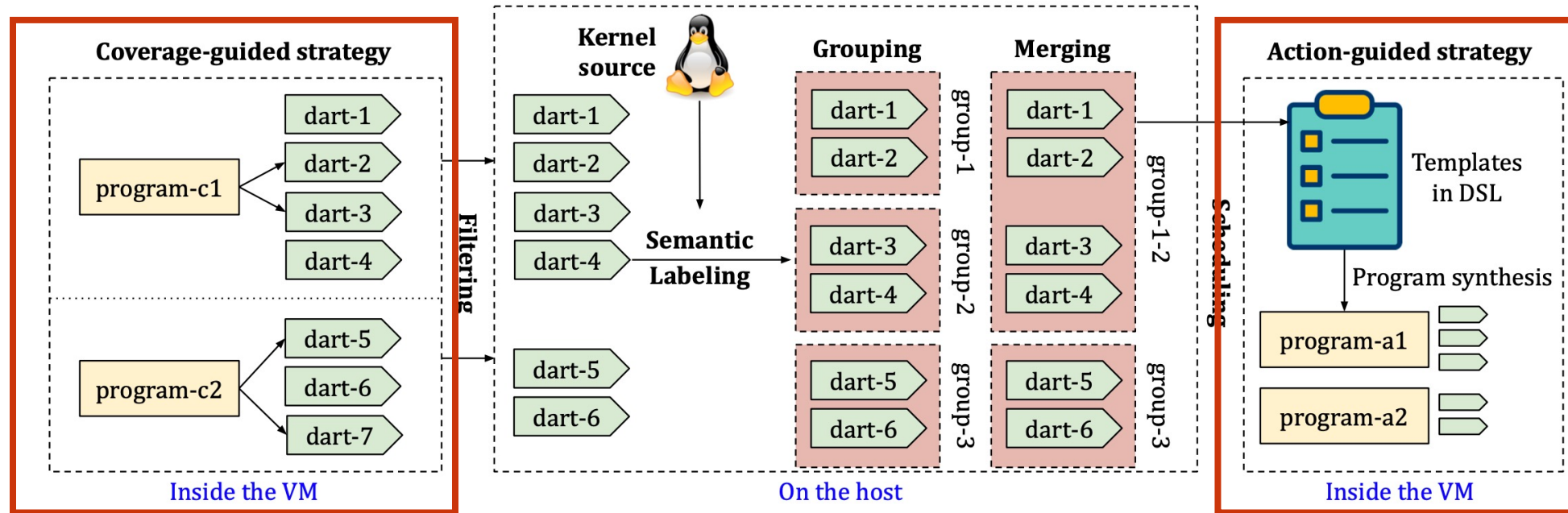
- ・ 41個の未知のバグを発見
- ・ 15個 (36.59%) は1日以内に発見
 - ・ 63.41% はメモリ破壊のバグ
 - ・ 26.83% は assertion などの論理的なバグ

実験環境 (RQ2-RQ6)

- ・ 対象 Linux バージョン
 - ・ 開発時に使用したカーネル (5.17)
- ・ 各実験は 24時間実行し 5回繰り返した平均値

RQ2 : Re-execution Success

RQ2 : Dart は異なるカーネル状態でも同じ action を実行するか？



Action mining 時点とプログラム生成時点でカーネルの状態が異なる

→ 同じシステムコールが別の action を実行するかも

RQ2 : Re-execution Success

RQ2 : Dart は異なるカーネル状態でも同じ action を実行するか？

Action	Re-ex. success	Action	Re-ex. success
Alloc (\mathcal{A}_a)	68.07%	Dealloc (\mathcal{A}_d)	42.92%
Val Read (\mathcal{A}_{vr})	38.91%	Val Write (\mathcal{A}_{vw})	32.91%
Ptr Read (\mathcal{A}_{pr})	38.18%	Ptr Write (\mathcal{A}_{pw})	56.27%
Idx Read (\mathcal{A}_{ir})	29.37%	Idx Write (\mathcal{A}_{iw})	18.49%
Overall		54.68%	

それぞれの action type ごとの再実行成功率

RQ2 : Re-execution Success

RQ2 : Dart は異なるカーネル状態でも同じ action を実行するか？

Action	Re-ex. success	Action	Re-ex. success
Alloc (\mathcal{A}_a)	68.07%	Dealloc (\mathcal{A}_d)	42.92%
Val Read (\mathcal{A}_{vr})	38.91%	Val Write (\mathcal{A}_{vw})	32.91%
Ptr Read (\mathcal{A}_{pr})	38.18%	Ptr Write (\mathcal{A}_{pw})	56.27%
Idx Read (\mathcal{A}_{ir})	29.37%	Idx Write (\mathcal{A}_{iw})	18.49%
Overall		54.68%	

それぞれの action type ごとの再実行成功率

- alloc 直後に割り当て領域がポインタ変数に代入 ptr write
→ ptr write の高い成功率

RQ2 : Re-execution Success

RQ2 : Dart は異なるカーネル状態でも同じ action を実行するか？

Action	Re-ex. success	Action	Re-ex. success
Alloc (\mathcal{A}_a)	68.07%	Dealloc (\mathcal{A}_d)	42.92%
Val Read (\mathcal{A}_{vr})	38.91%	Val Write (\mathcal{A}_{vw})	32.91%
Ptr Read (\mathcal{A}_{pr})	38.18%	Ptr Write (\mathcal{A}_{pw})	56.27%
Idx Read (\mathcal{A}_{ir})	29.37%	Idx Write (\mathcal{A}_{iw})	18.49%
Overall		54.68%	

- ・ アロケーション, システムコールごとに最初の書き込みだけを記録
 - ほとんどの場合初期化書き込み
 - 2回目以降の index write は失敗する可能性が高い

RQ3 : Shared Accesses

RQ3 : Actor は多くの共有アクセスをトリガーできるか？

- ・ dart グループのマージが効果的なら, 共通のメモリを操作する
→ 共有メモリアクセスが増加するはず

If ACTOR's *merging* strategy is effective, it should generate groups with related darts that operate on a common memory buffer.

Then, during *program synthesis*, since ACTOR chooses darts from these groups, the darts should result in actions that generate shared memory accesses.

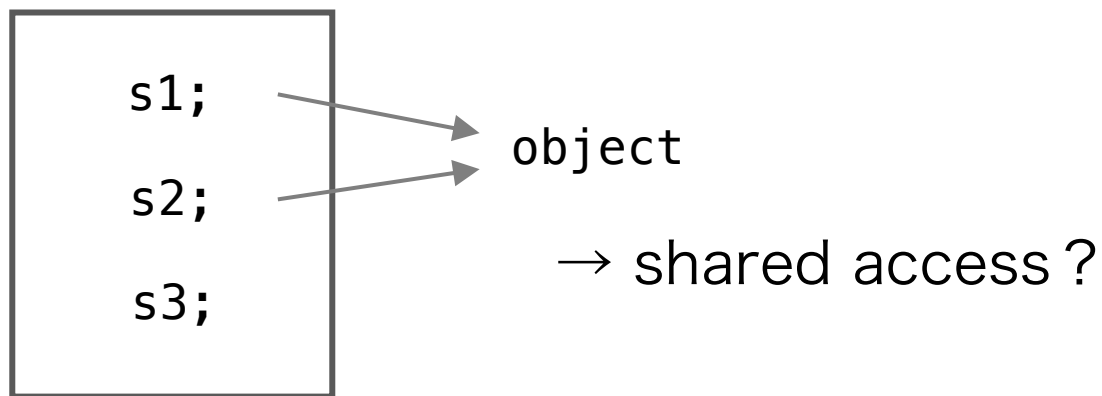
RQ3 : Shared Accesses

RQ3 : Actor は多くの共有アクセスをトリガーできるか？

- ・ dart グループのマージが効果的なら, 共通のメモリを操作する

→ 共有メモリアクセスが増加するはず

システムコールの列



RQ3 : Shared Accesses

RQ3 : Actor は多くの共有アクセスをトリガーできるか？

Subsystem	ACTOR	SYZKALLER+	Improvement
arch/	389,978	321,947	21.13%
block/	158,012	168,602	−6.28%
certs/	0	0	0.00%
crypto/	29,405	41,628	−29.36%
drivers/	1,068,698	775,113	37.88%
fs/	4,222,386	4,733,901	−10.81%
ipc/	132,340	141,906	−6.74%
kernel/	3,684,928	2,734,836	34.74%
lib/	1,408,382	659,223	113.64%
mm/	132,374	117,369	12.78%
net/	6,178,633	3,409,346	81.23%
security/	3,020,899	2,925,410	3.26%
sound/	261,289	224,960	16.15%
total	20,687,324	16,254,239	27.27%

Table 4: Shared accesses of ACTOR and SYZKALLER+ per subsystem

RQ3 : Shared Accesses

RQ3 : Actor は多くの共有アクセスをトリガーできるか？

Subsystem	ACTOR	SYZKALLER+	Improvement
arch/	389,978	321,947	21.13%
block/	158,012	168,602	-6.28%
certs/	0	0	0.00%
crypto/	29,405	41,628	-29.36%
drivers/	1,068,698	775,113	37.88%
fs/	4,222,386	4,733,901	-10.81%
ipc/	132,340	141,906	-6.74%
kernel/	3,684,928	2,734,836	34.74%
lib/	1,408,382	659,223	113.64%
mm/	132,374	117,369	12.78%
net/	6,178,633	3,409,346	81.23%
security/	3,020,899	2,925,410	3.26%
sound/	261,289	224,960	16.15%
total	20,687,324	16,254,239	27.27%

サブシステムが小さい
 → バグテンプレートに必要な action を持つ
 dart が集まらない
 → プログラム合成が
 実行できない

Table 4: Shared accesses of ACTOR and SYZKALLER+ per subsystem

RQ4 : Bug-including Program Generation

RQ4 : Actor はバグを誘発するパターンのプログラムを多く生成できるか？

- Syzkaller+ :

dart のグループを生成するが, Syzkaller のプログラム合成を使用

- バグテンプレートの順に action type を並べられる dart グループが存在するプログラムの数を計測

プログラム

{s1, s2, s3}

(s1, alloc)

(s2, dealloc)

(s3, read)

RQ4 : Bug-including Program Generation

RQ4 : Actor はバグを誘発するパターンのプログラムを多く生成できるか？

Strategy	ACTOR	SYZKALLER+	Improvement
Use After Free (UAF)	2,085,492	92,844	22.46
Double Free (DF)	2,266,692	79,440	28.53
Out of Bounds (OOB-1)	517,092	24,702	20.93
Out of Bounds (OOB-2)	422,298	11,160	37.84
Uninitialized Read (UR)	5,906,130	1,960,104	3.01
Null Ptr Deref (NPD)	7,468,788	652,764	11.44
Invalid Free (IF)	26,296,746	22,583,958	1.16
Memory Leak (ML-1)	215,840,400	180,381,540	1.20
Memory Leak (ML-2)	986,856	45,486	21.70
Total	261,790,494	205,831,998	1.27

複雑なバグテンプレート
には効果的

Bug class	Template	Bug class	Template
Use After Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow [\mathcal{A}_r \mathcal{A}_w]$	Null Ptr Deref	$\mathcal{A}_a^x \rightarrow \mathcal{A}_d^x$
Double Free	$\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow \mathcal{A}_d$	Invalid Free	\mathcal{A}_d
Out of Bounds (1)	$\mathcal{A}_a \rightarrow \mathcal{A}_{iw}^* \rightarrow \mathcal{A}_{ir}$	Memory Leak (1)	\mathcal{A}_a^*
Out of Bounds (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw}^* \rightarrow \mathcal{A}_{pr}$	Memory Leak (2)	$\mathcal{A}_a \rightarrow \mathcal{A}_{pw} \rightarrow \mathcal{A}_d$
Uninitialized Read	$\mathcal{A}_a \rightarrow \mathcal{A}_r$	-	-

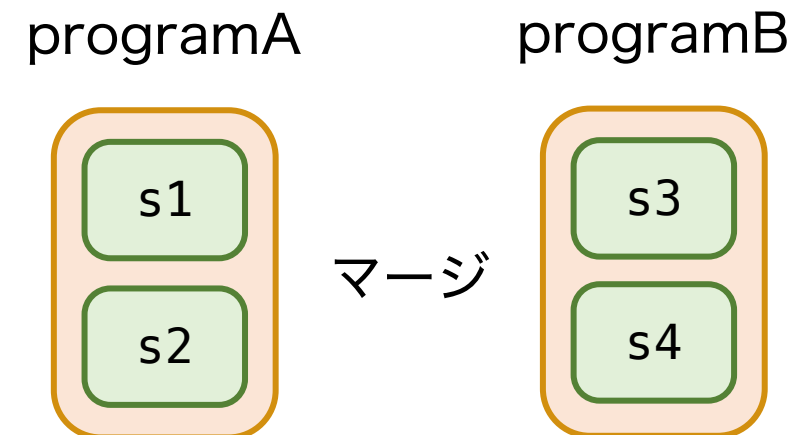
RQ5 : Syscall affinity

RQ5 : Actor は Syzkaller の発見できないシステムコール間の関係を発見できるか？

- ・ Dart グループのマージ時

$(s1, s3), (s1, s4), (s2, s3), (s2, s4)$ のペア

- 同時に現れる確率が低い (平均以下)
- Syzkaller では発見できない関係



- ・ 8,082 / 36,649 (22.05%) は Syzkaller では発見できない関係

Syzkaller [1]

- ・ あるシステムコールが別のシステムコールより先に呼び出される
確率を記録

→ 静的にも動的にも更新

choice table

		syscall		
		A	B	C
syscall	A			
	B			
	C			

- ・ **静的**：システムコールとその引数・返値の型の情報を持っている
→ 同じ型を共有していれば確率を上げる
- ・ **動的**：新しいカバレッジが得られたとき、そこに現れる
2つのシステムコールの確率を上げる

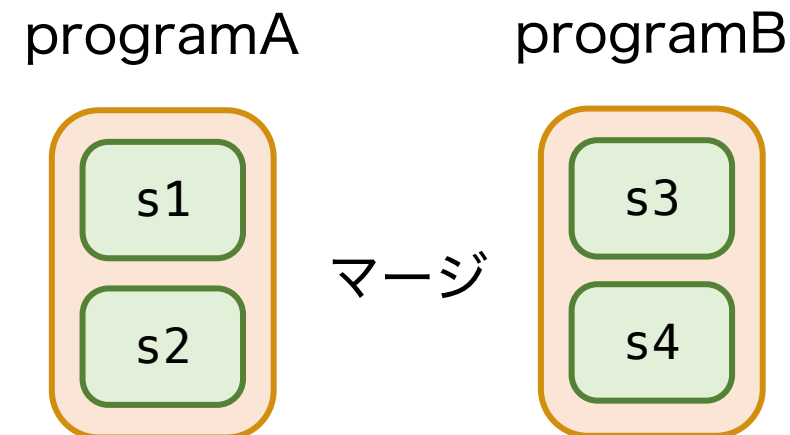
RQ5 : Syscall affinity

RQ5 : Actor は Syzkaller の発見できないシステムコール間の関係を発見できるか？

- ・ Dart グループのマージ時

(s1, s3), (s1, s4), (s2, s3), (s2, s4) のペア

- 同時に現れる確率が低い (平均以下)
- Syzkaller では発見できない関係



- ・ 8,082 / 36,649 (22.05%) は Syzkaller では発見できない関係

RQ6 : Comparison

RQ6 : Actor と既存のカーネルファザーとの比較

比較対象 :

- Actor
- Syzkaller
- Healer
- Moonshine

選定基準 :

- (1) システムコール間の関係を学習する
- (2) 特定のバグクラス / サブシステムをターゲットにしない汎用ファザー

RQ6 : Comparison

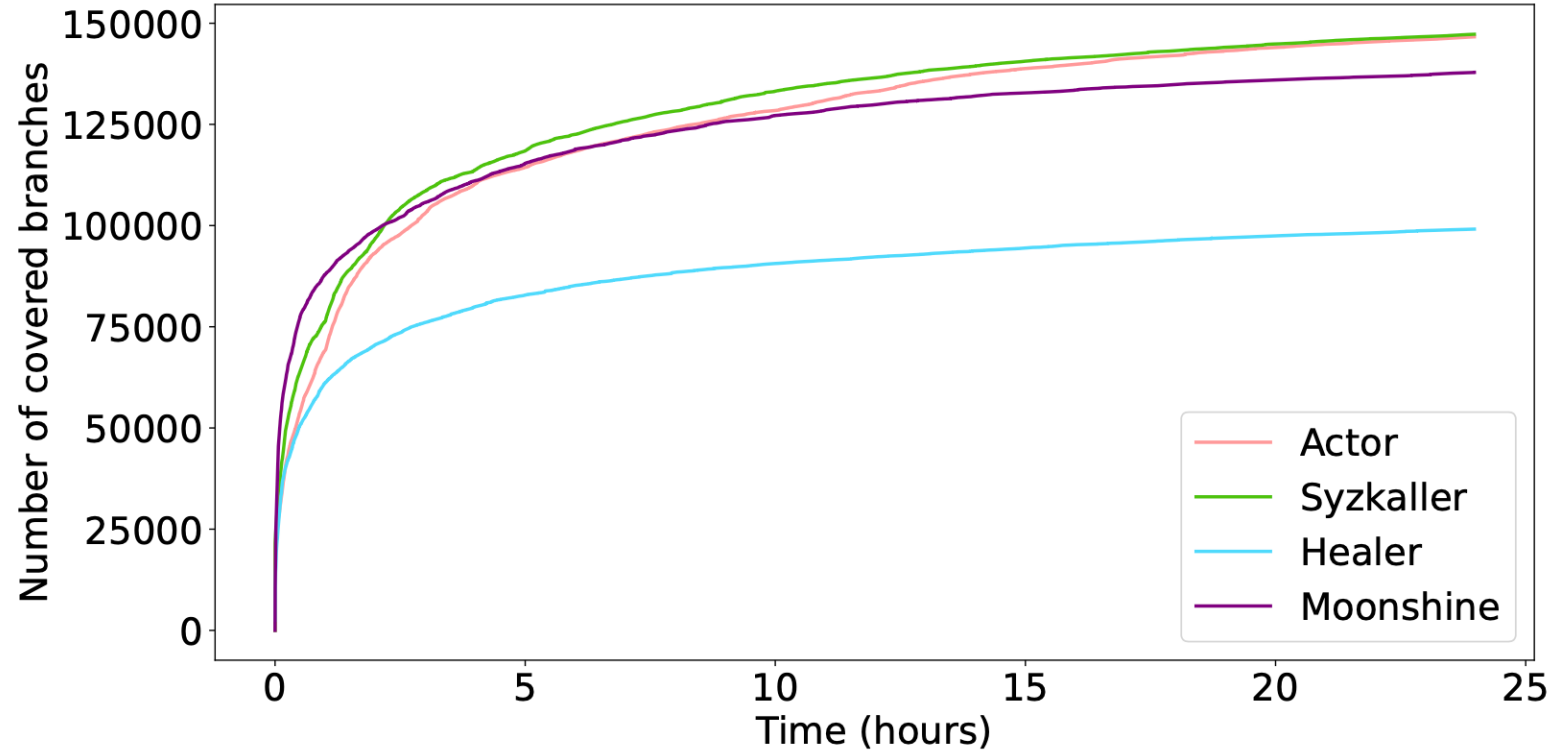


Figure 4: Coverage attained by ACTOR, SYZKALLER, HEALER, and MOONSHINE over 24 hours

RQ6 : Comparison

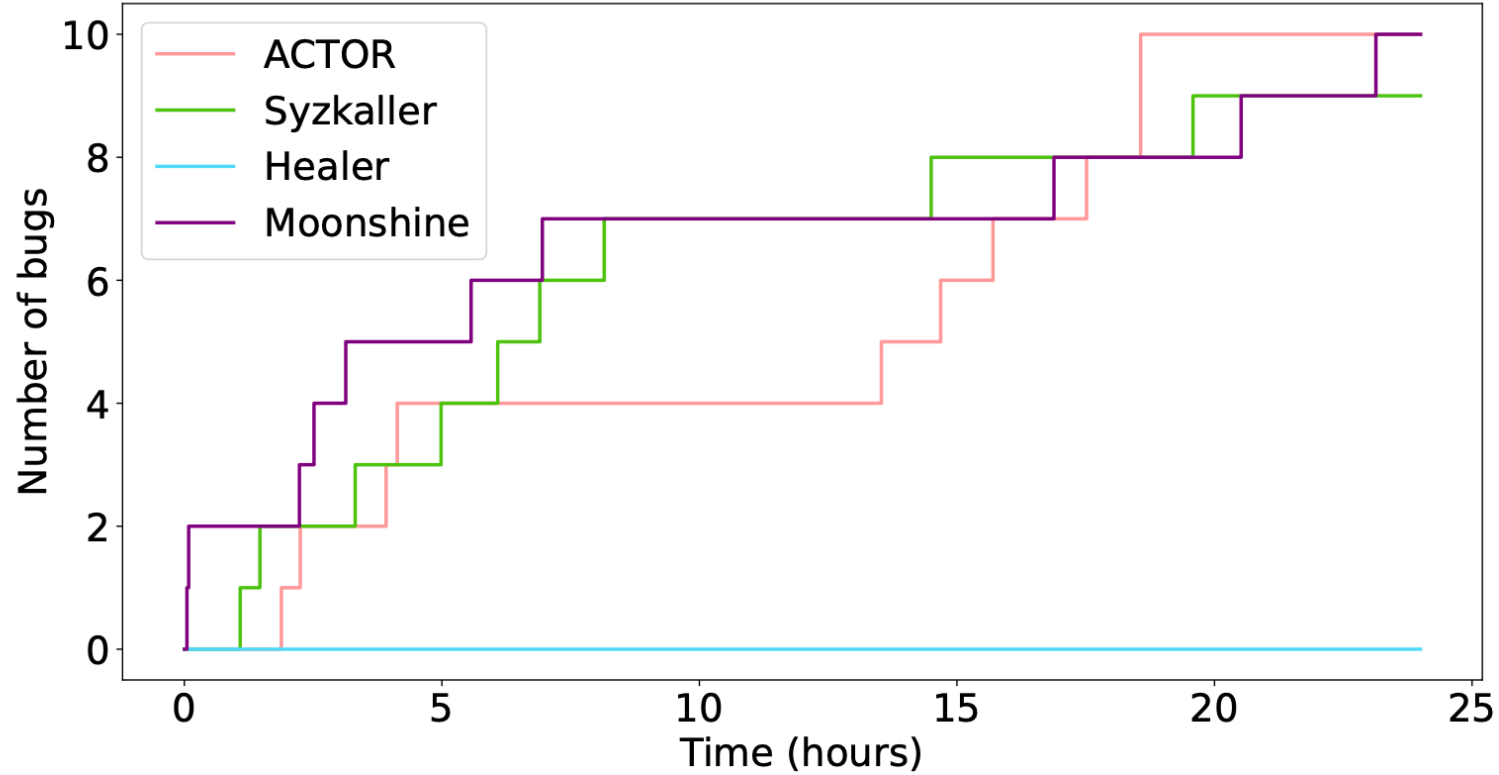


Figure 5: Bugs found by ACTOR, SYZKALLER, HEALER and MOONSHINE over 24 hours

Discussion and Limitation

- より多くの action type のサポート
 - グローバル変数・参照カウンタに関するバグなど
 - actrack モジュールの拡張
 - dart labeling の拡張
- 入れ替え不可能な action
 - 同じシステムコールから得られる2つの action は常に同じ順番
 - 常に同じ順番に実行されるか判断するのは難しい
 - 静的解析などはファジングループ内に実装する必要あり

関連研究

- ・ 汎用カーネルファザー
 - ・ Syzkaller, Healer, Moonshine
- ・ サブシステムをターゲットにしたファザー
 - ・ ドライバ, ファイルシステム
- ・ 特定の OS に依存しないカーネルファザー
- ・ 競合状態をターゲットとしたカーネルファザー