

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

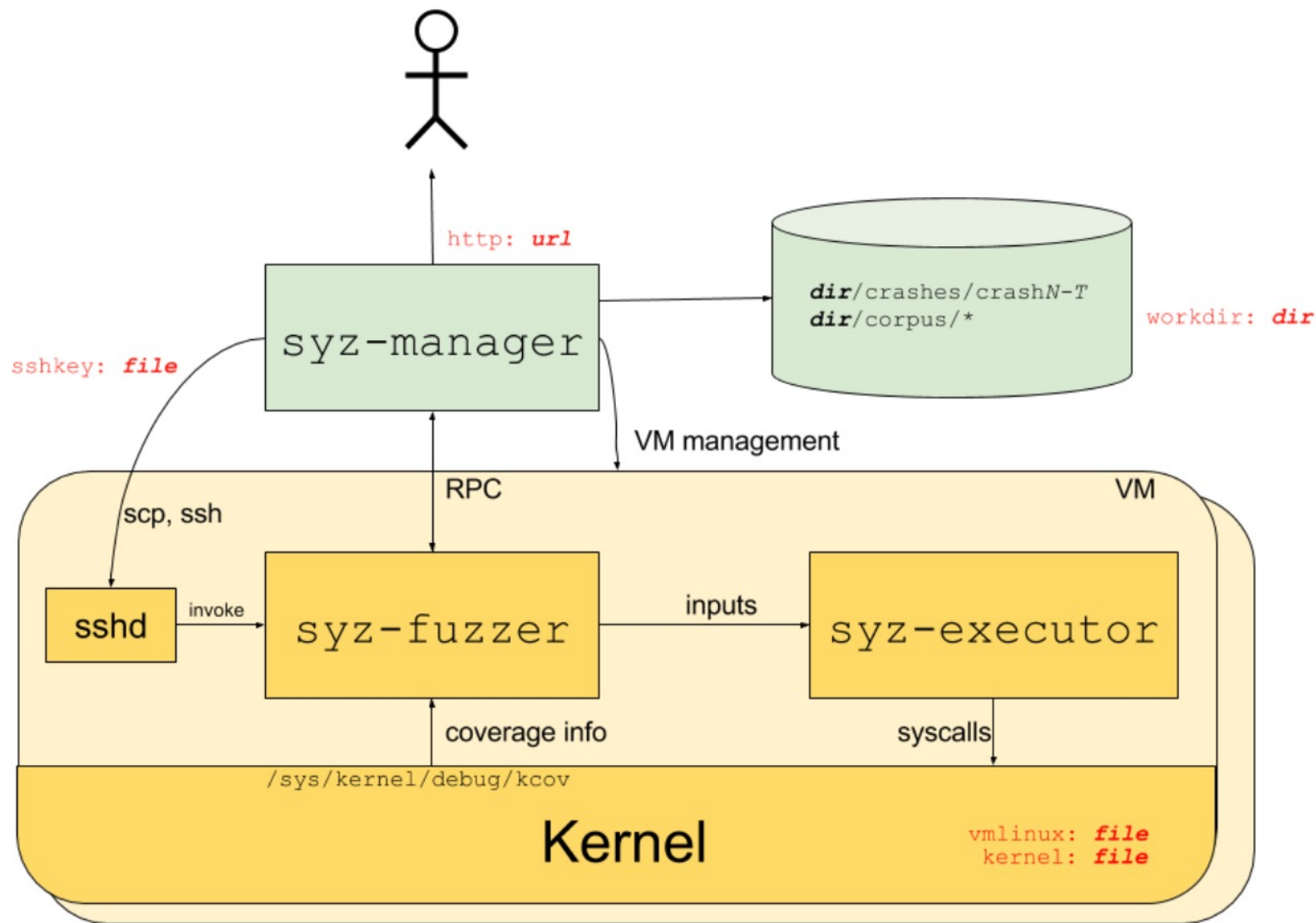
```
1 fd = openat(0, "...", ...)  
2 write(fd, &{CREATE}, ...)  
3 write(fd, &{LISTEN}, ...)  
4 write(fd, &{DESTROY}, ...)
```

- ・ 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>

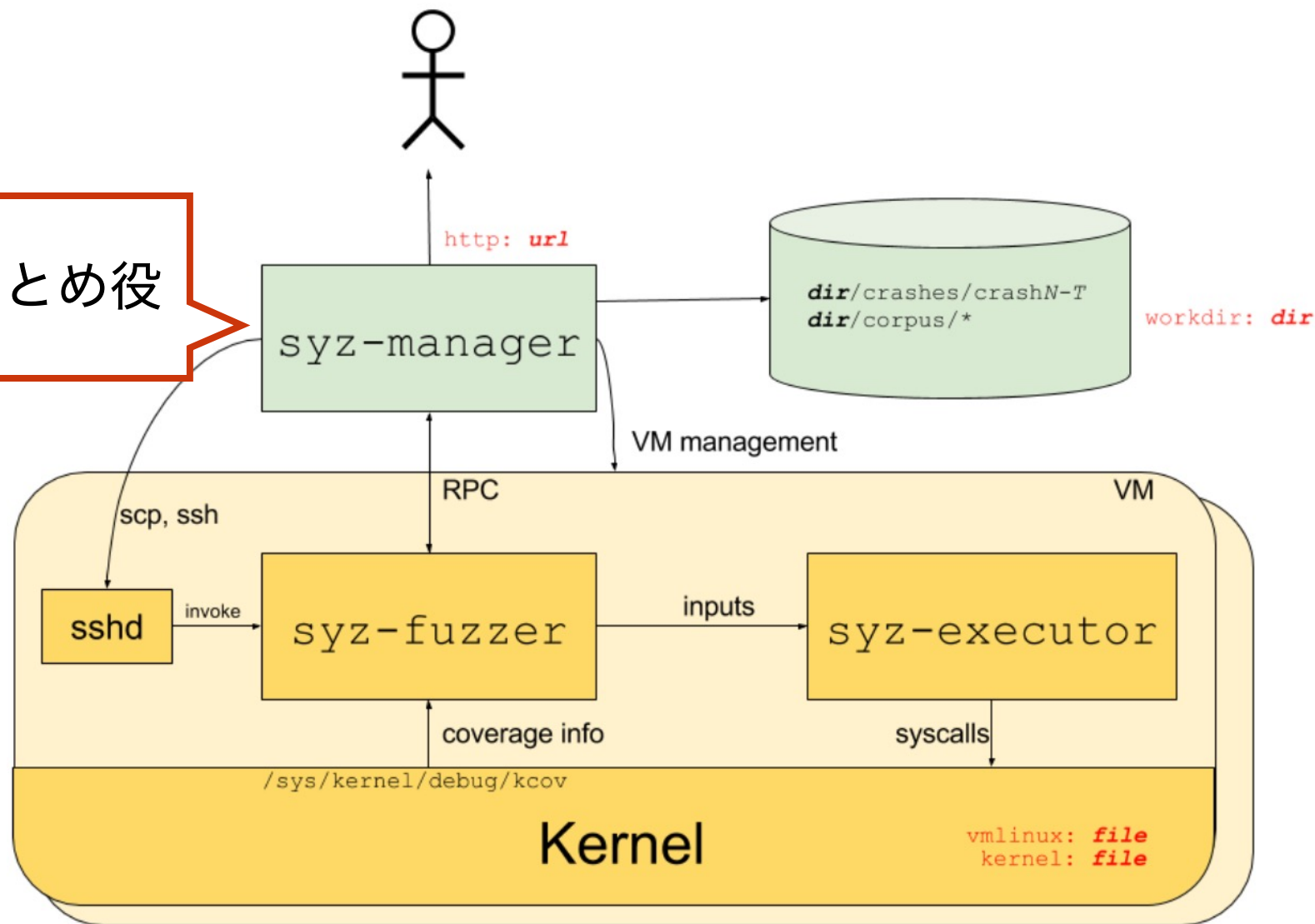
[3] <https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>

Syzkaller [1]



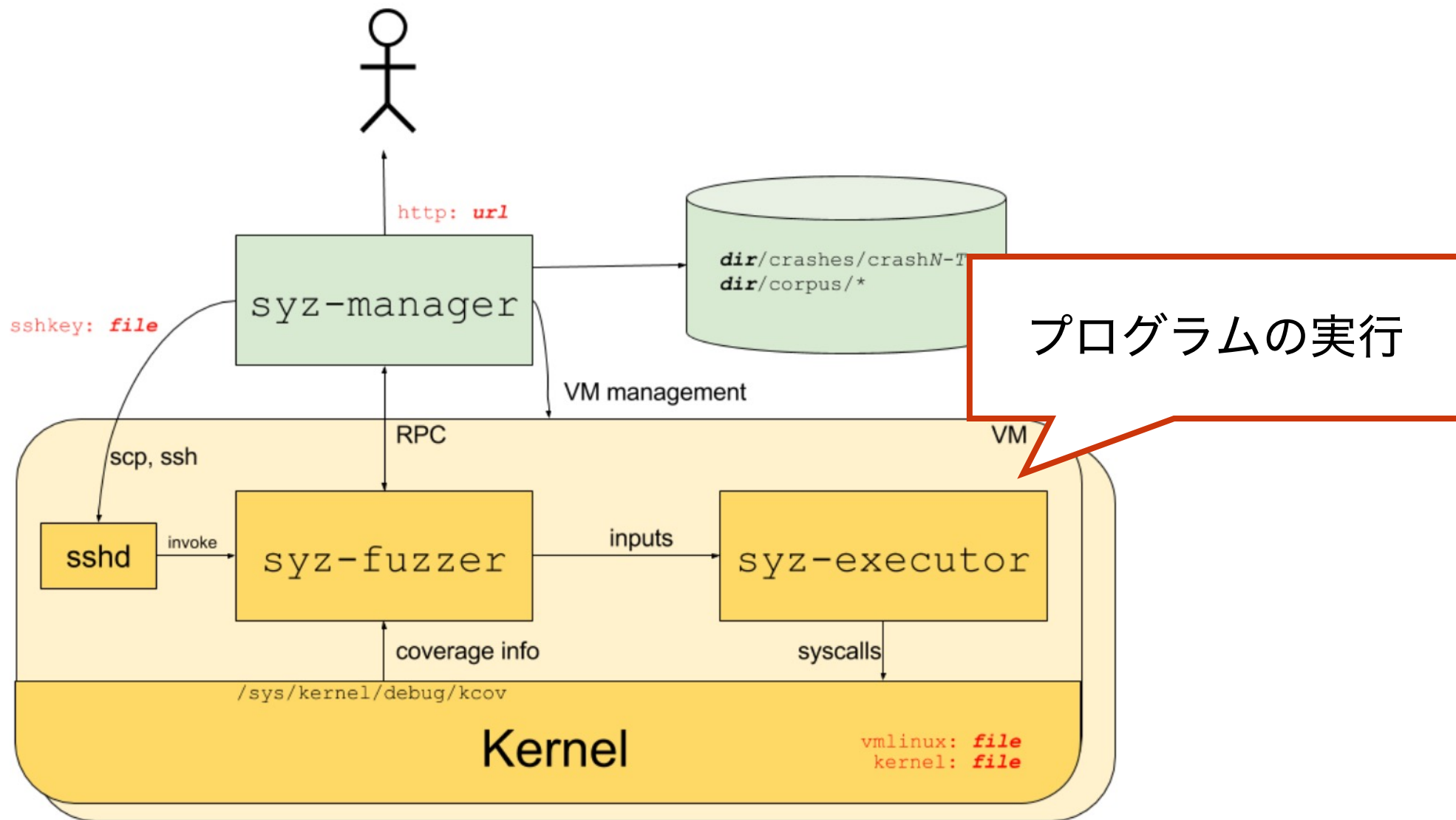
Syzkaller [1]

複数 VM のまとめ役





Syzkaller [1]



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

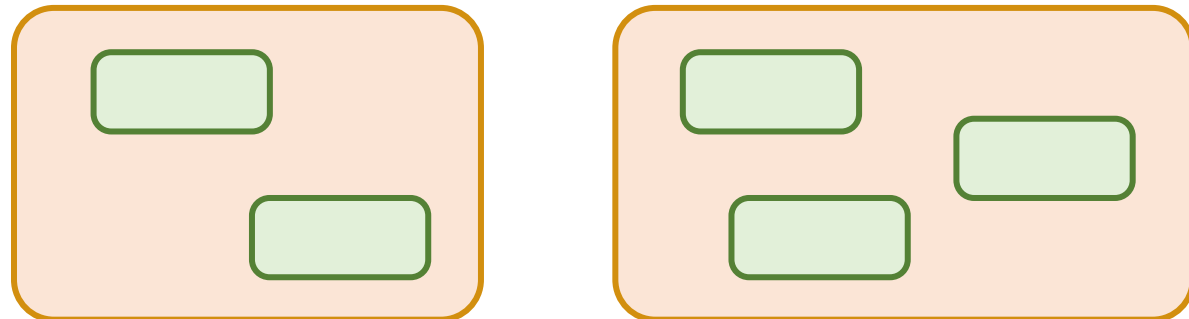
- ・ システムコールと action の関係 (dart) を動的に収集

→ カバレッジガイド戦略を用いる



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

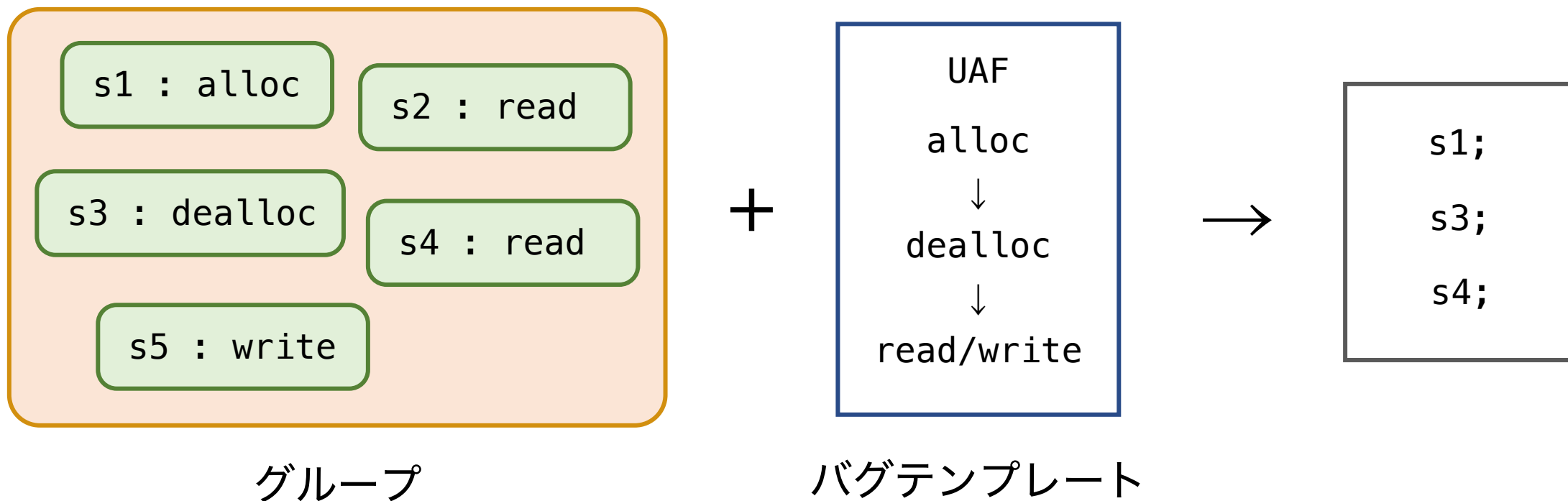
- ・ 同じメモリ領域を操作しそうな dart をグループ化



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



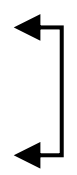
Actor

Actor の2つのフェーズ

(1) Action mining

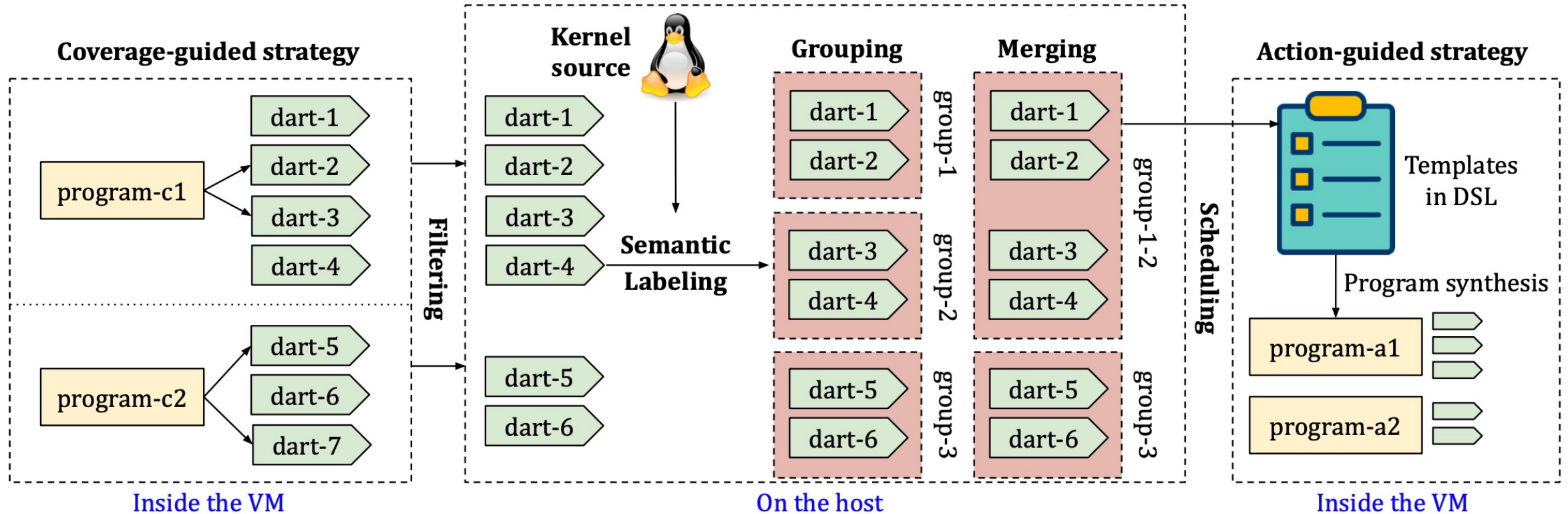
(2) プログラム合成

→ 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

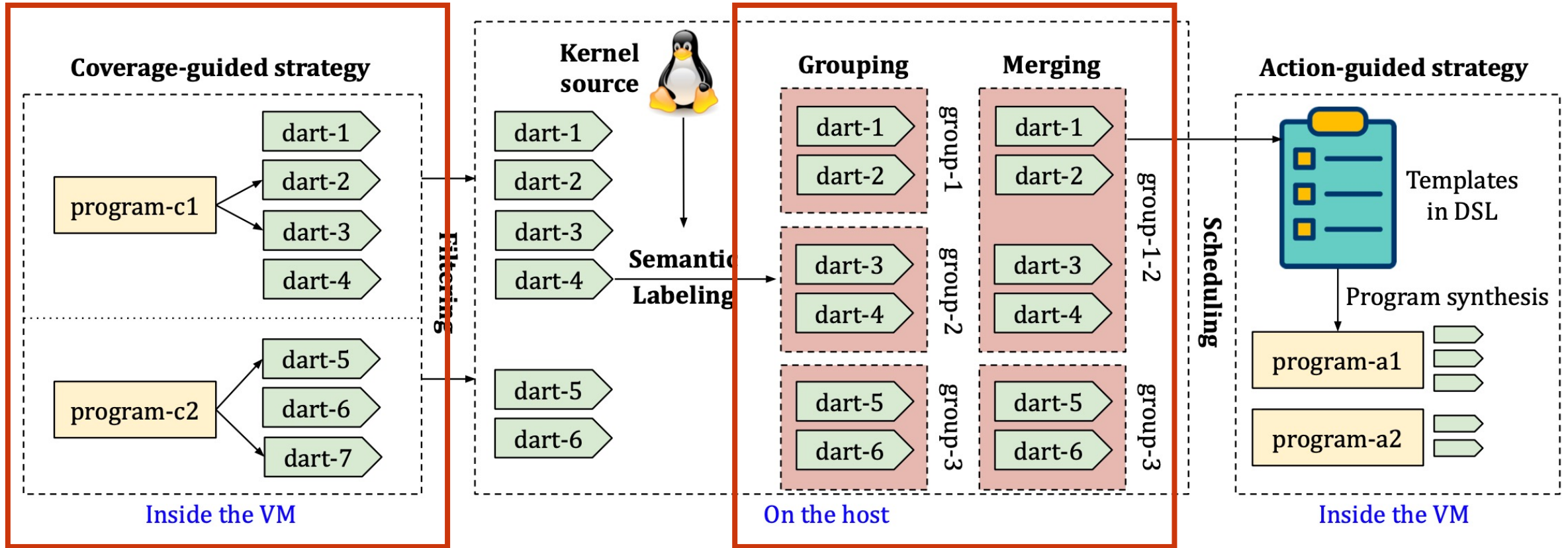
- ・カバレッジガイド的入力生成・変異
 - ・アクションガイド的入力生成・変異
- 
- ランダム

(1) → (2) の順番で明確に区別されるわけではない

Actor Workflow



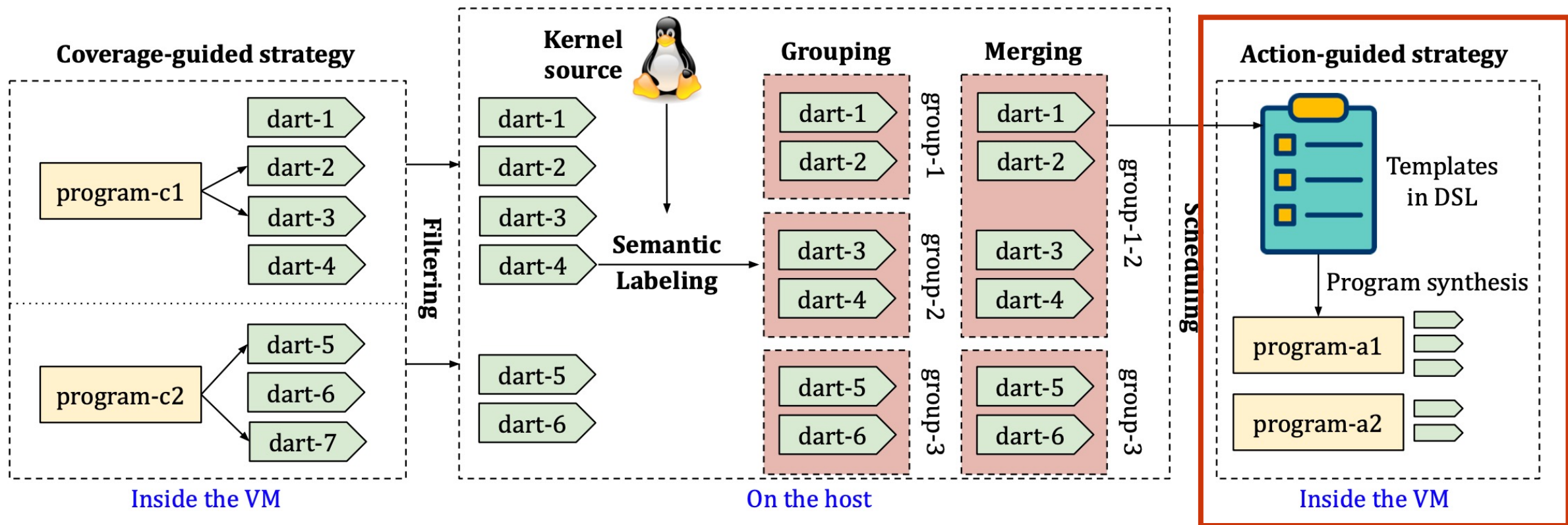
Actor Workflow



dart 収集

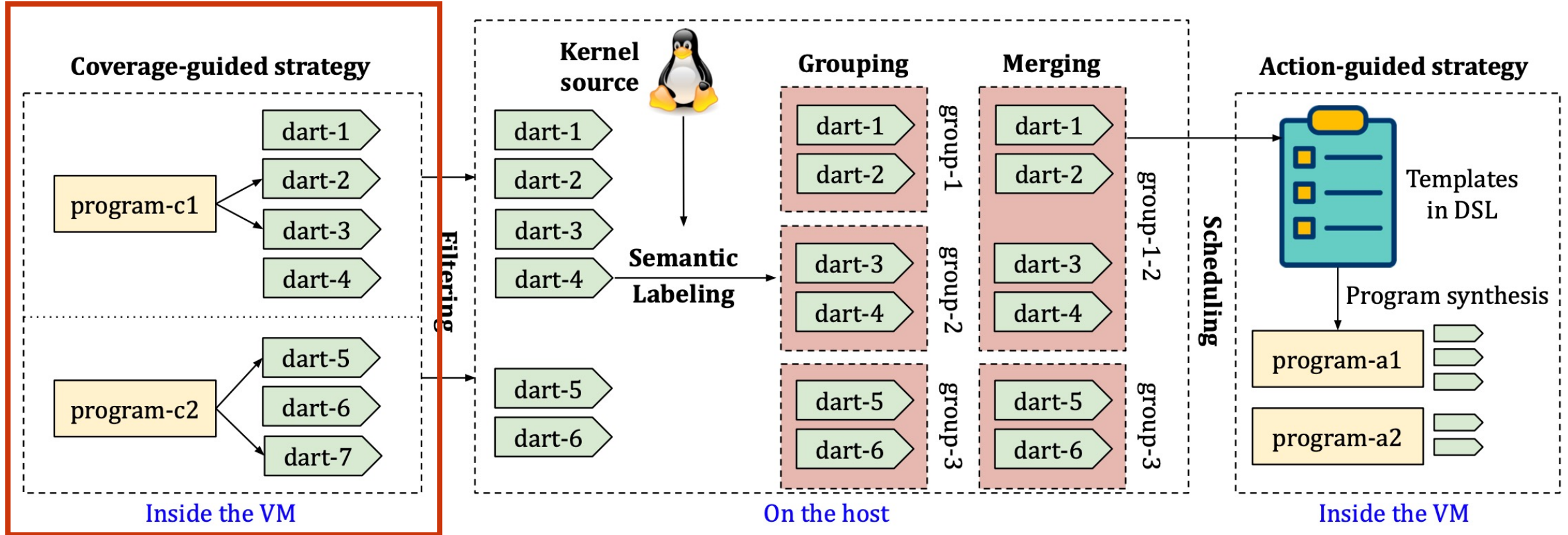
dart グループ化

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

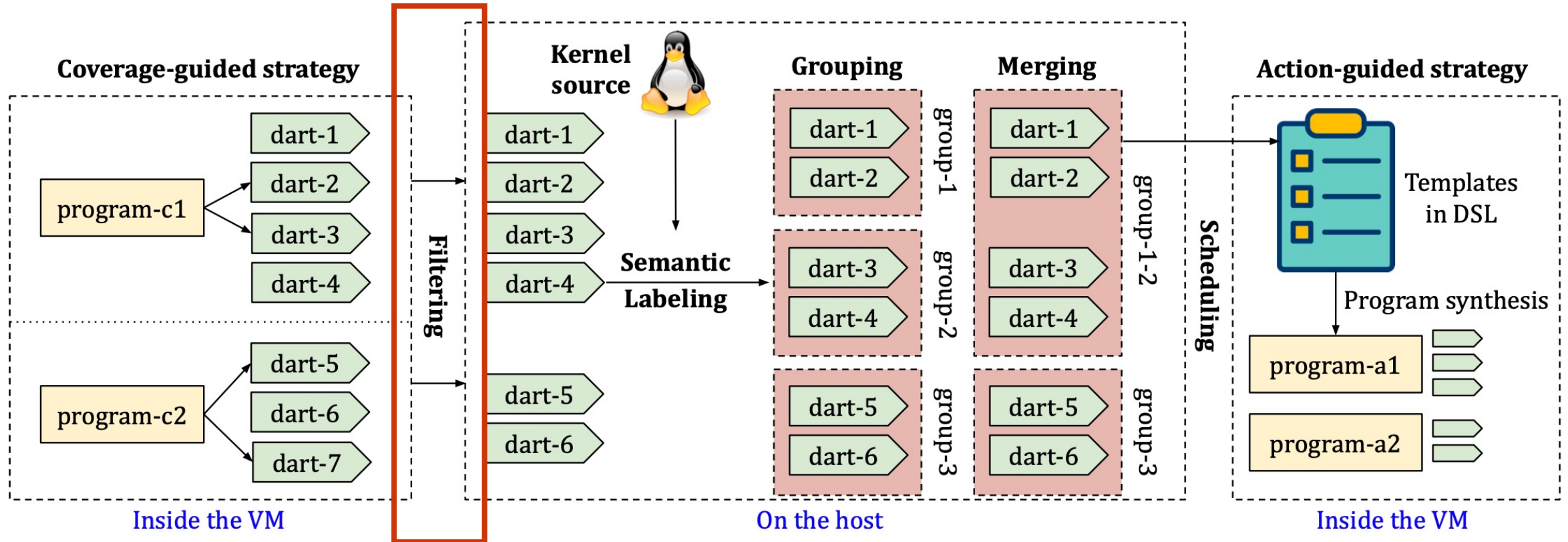
Dart

- ・ 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 (とそれによって実行される操作)
- ・ 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

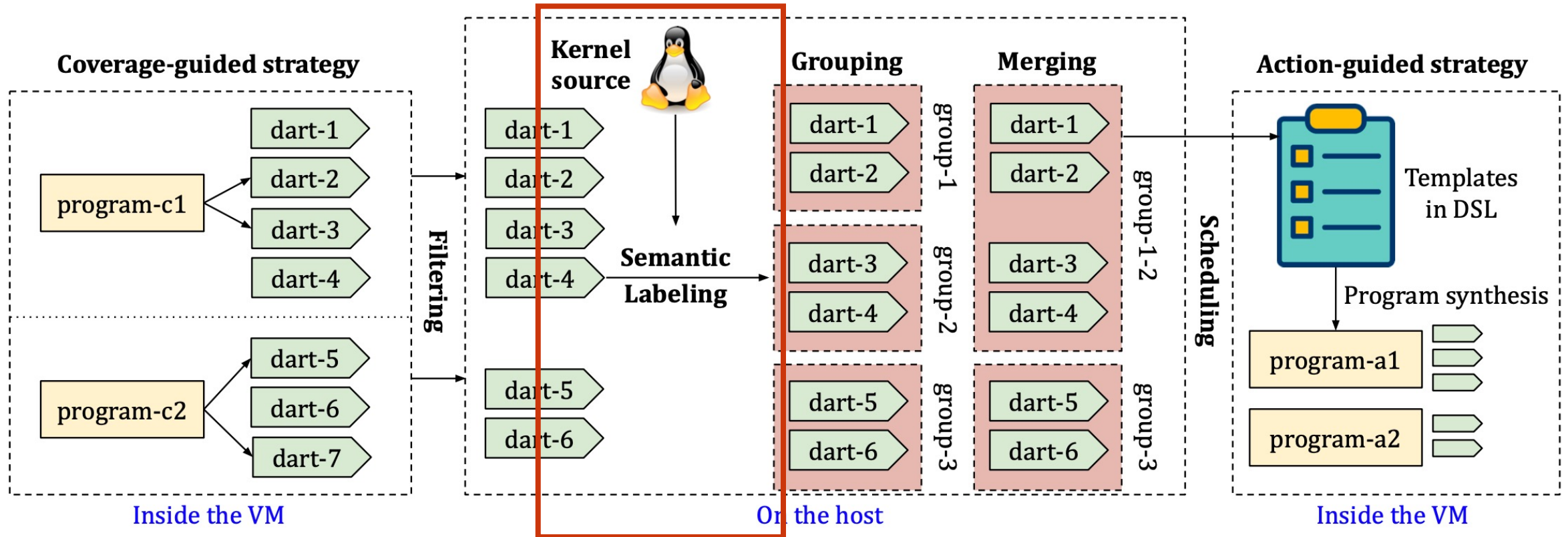
Actor Workflow



Dart Reduction

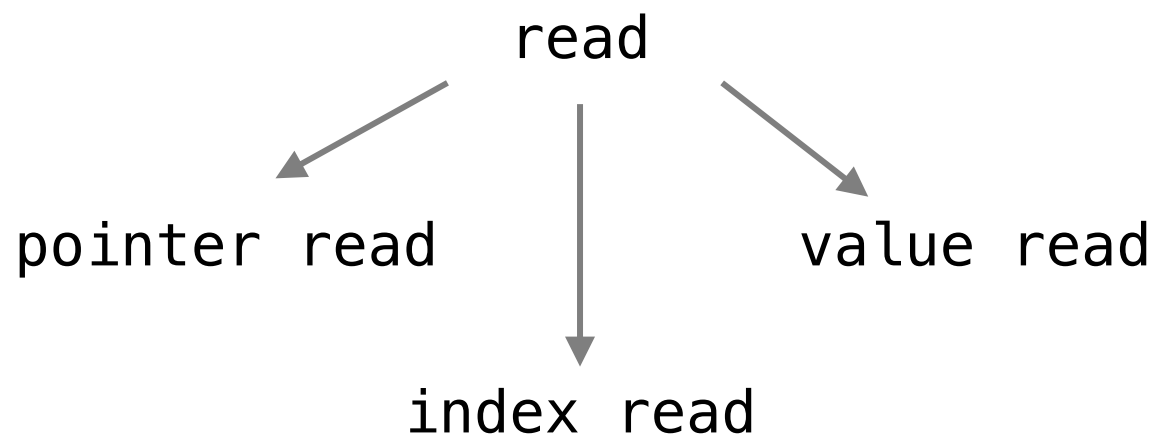
- ・ プログラムが大量の dart を生成する可能性があるため削減したい
- ・ 問題点
 1. VM からホストへの dart の転送オーバーヘッドが増加
 2. 効果的なプログラム合成が難しくなる
- ・ 削減方法
 1. それまでに alloc アクションによって割り当てられたメモリ領域を操作する dart のみを収集
 2. システムコールごと, 同じメモリ領域ごとに最初の read/write のみを記録

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

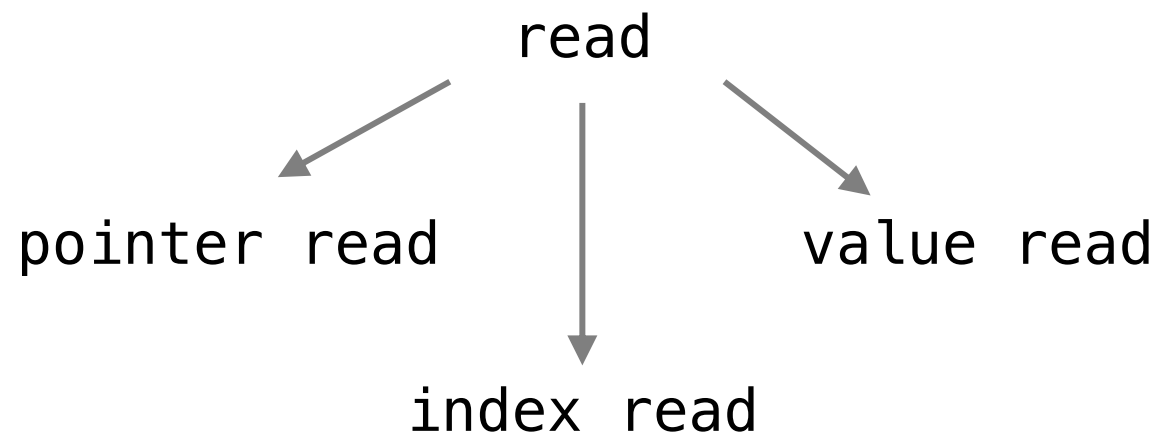
index write

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

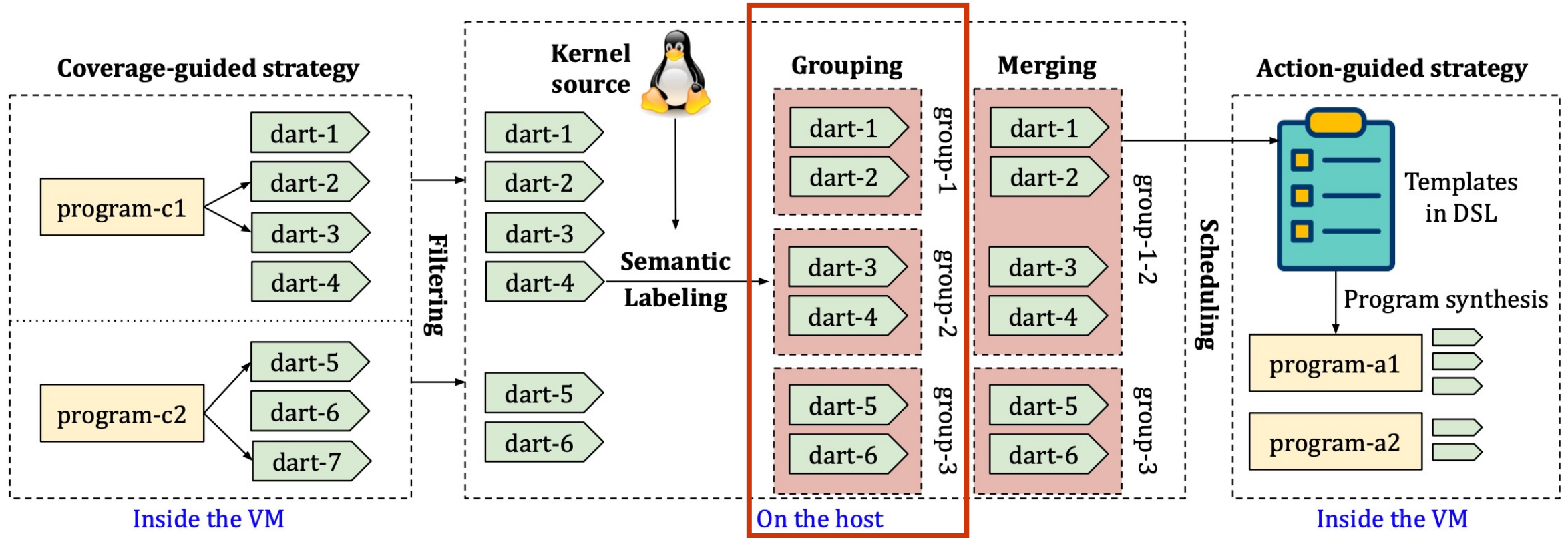
index read

Dart Labeling

- ・ 静的解析で各命令が index read/write か pointer read/write が分かった
- ・ 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

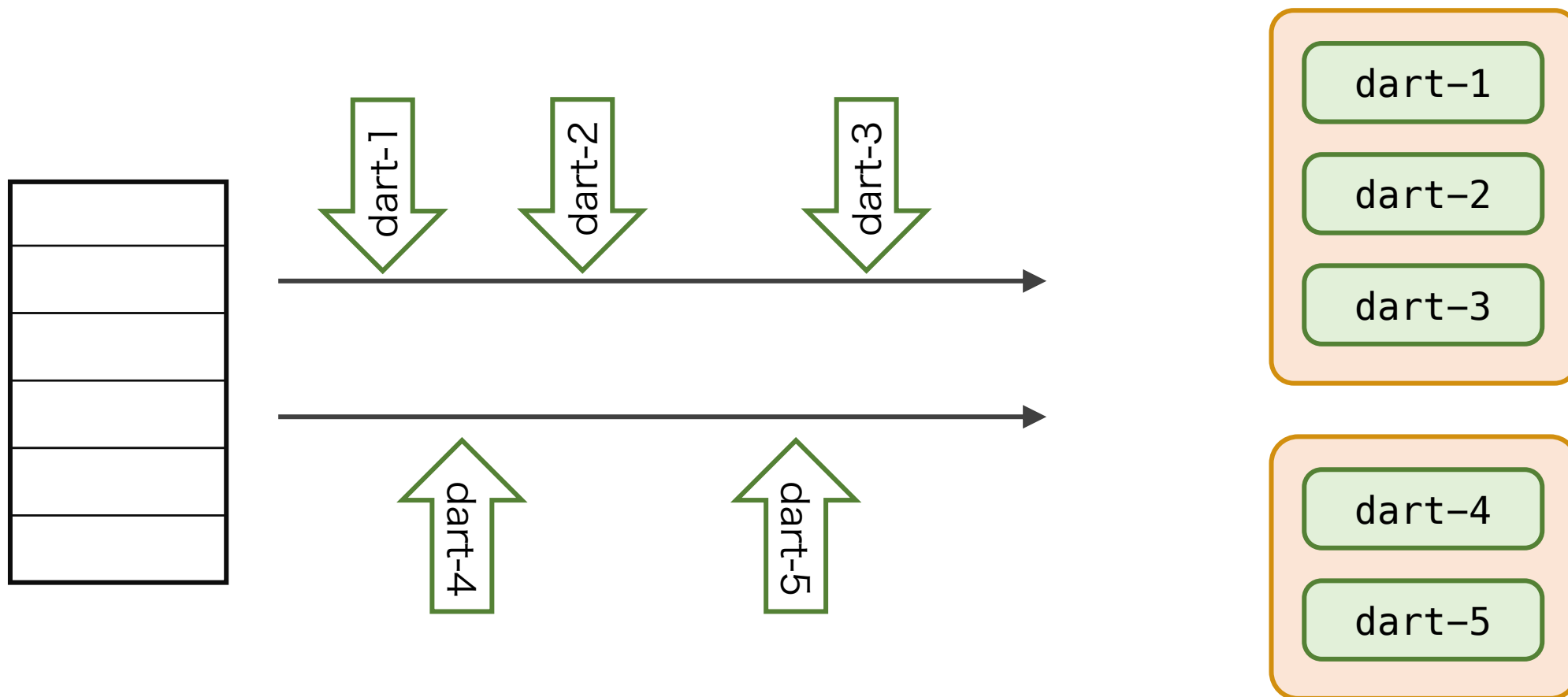


Actor Workflow

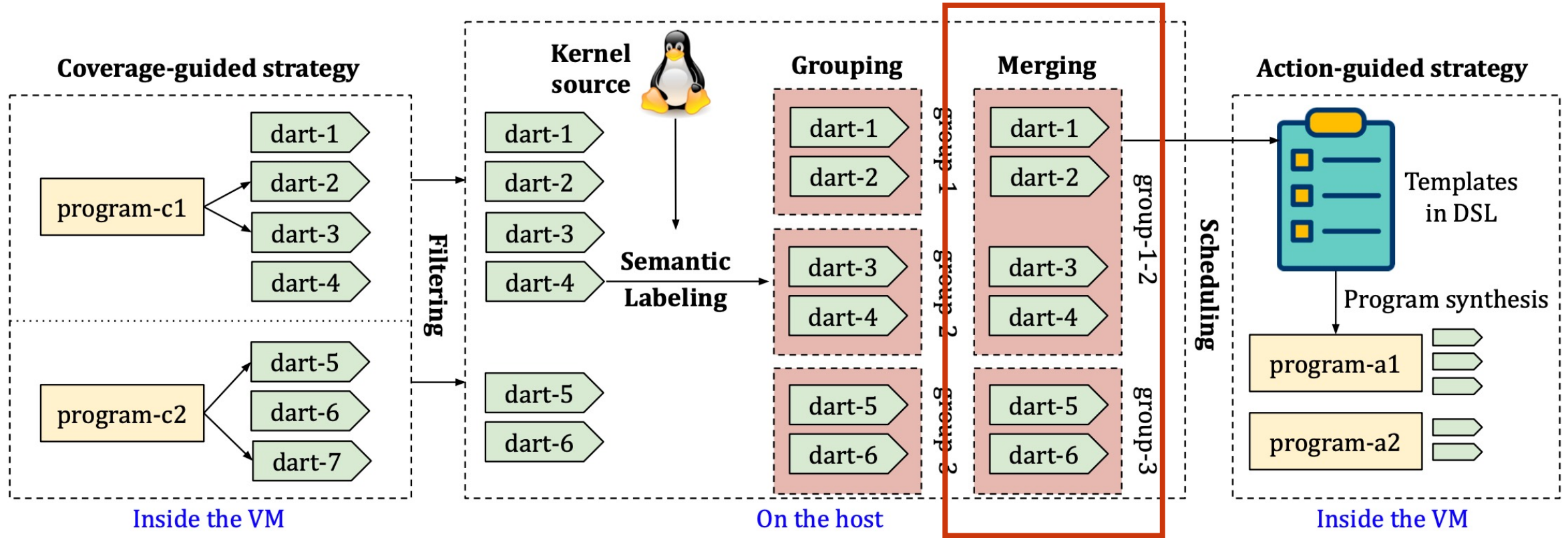


Dart Grouping

- ・ 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)

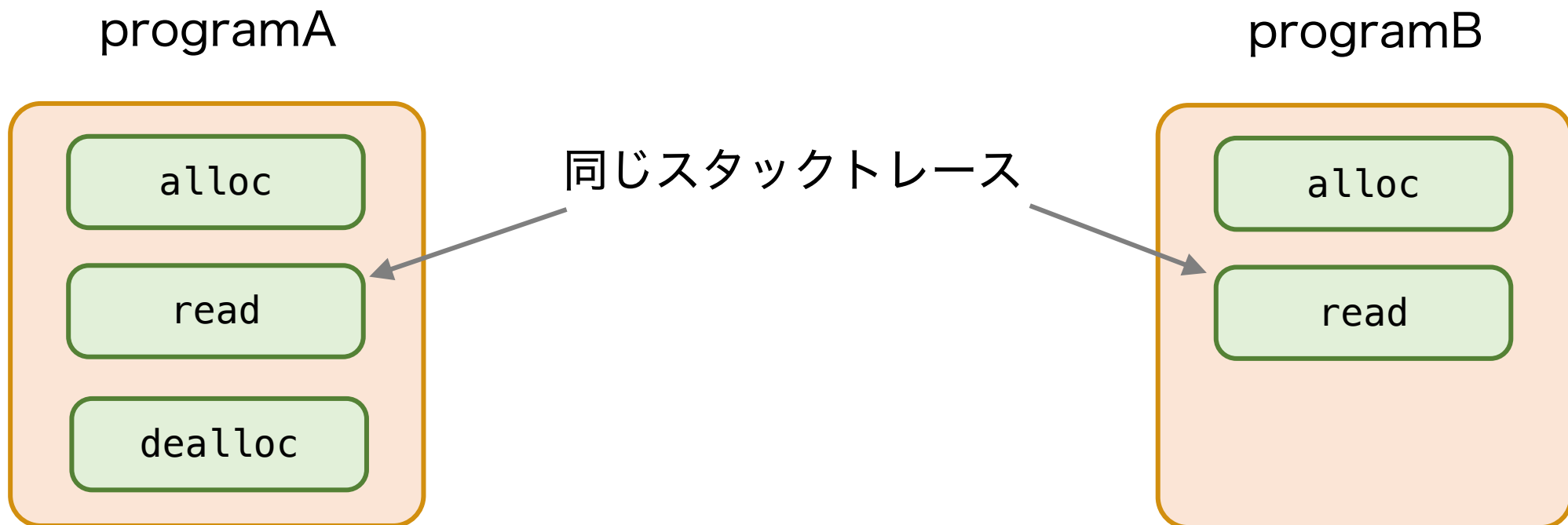


Actor Workflow



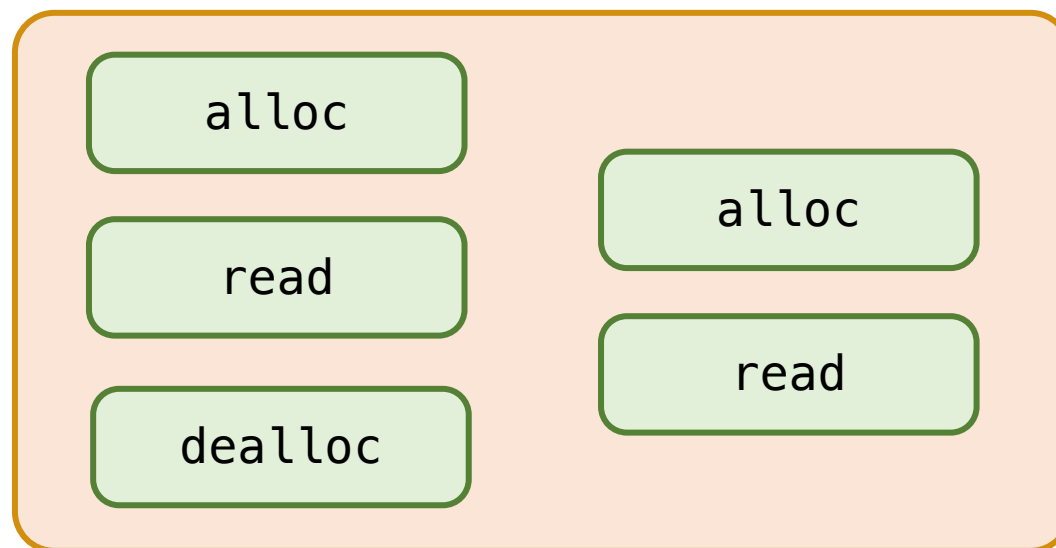
Dart Merging

- ・ 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

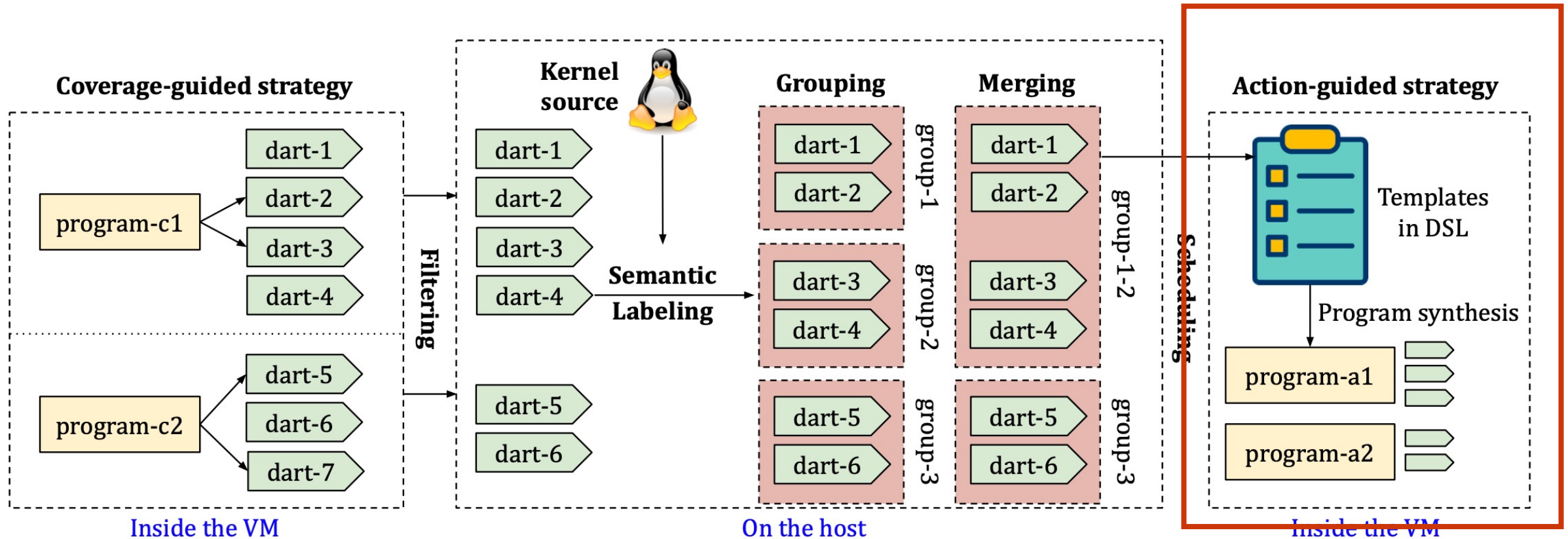
- ・ 複数のプログラムから得られる dart グループを結合
- ・ 2つのグループで同じスタックトレース + アクションタイプを持つ dart が存在するなら2つのグループをマージ



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

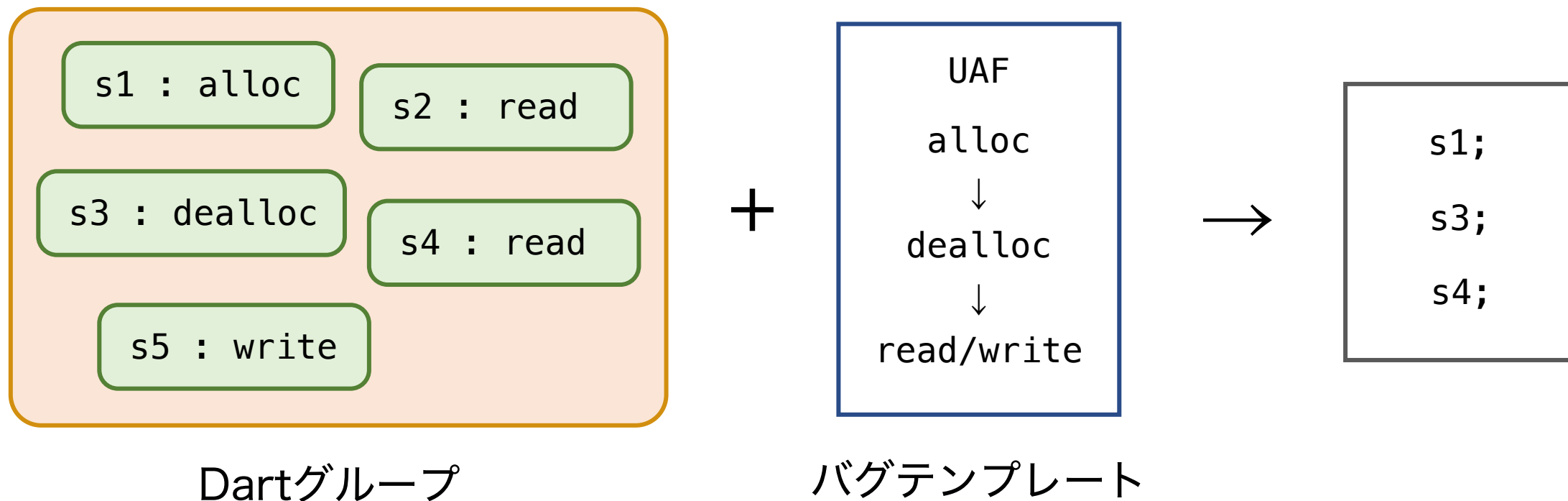
Actor Workflow



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`

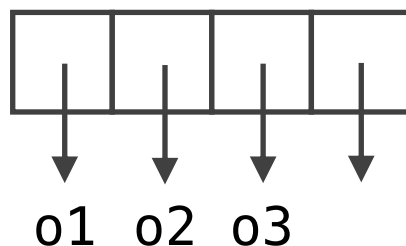
- `index` はループの中でインクリメントされることが多い
- `index write` を複数回繰り返すことで `array` の長さを超えることを期待

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



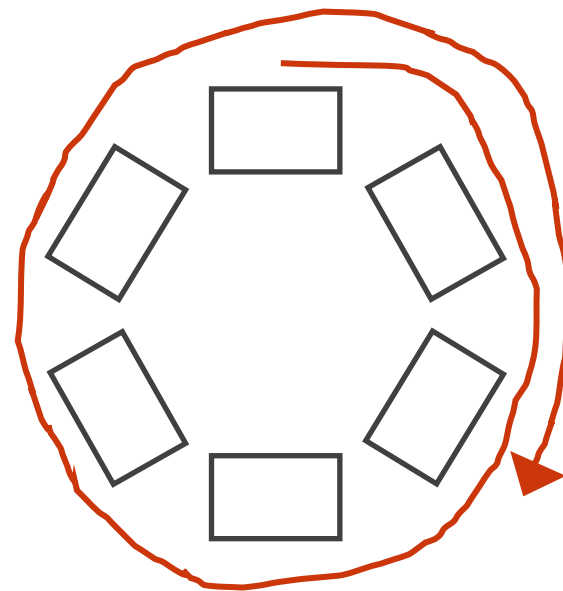
`c = 4`

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
- ・ 動機：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

残りの章

- Implementation
- Evaluation
- Discussion and Limitation
- Related Work

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