

高脅威メモリリークの バイナリレベル動的検知法

東京工業大学

小泉雄太 荒堀喜貴 権藤克彦

2019/01/31

発表概要

問題:

- 高脅威度を考慮したメモリリーク検出技術は不十分
- 既存の高脅威リーク検出技術は型情報・GCに依存

提案:新しいメモリリーク検出技術Pikelet

- 呼出し文脈でグループ化したオブジェクト群の成長率に基づき高脅威リークを検出
- 型情報・GCに依存せず(C/C++)の動的バイナリ解析で実現

実験:

- 精度:高脅威リークを既存研究よりも高精度に検出
- 実行オーバーヘッド:
 - 既存研究と同程度の実行オーバーヘッドを達成

問題背景： メモリリークとは

メモリリークとは

- ヒープ領域上に動的に確保されたオブジェクトが解放されずに残り続けるバグ

症状：パフォーマンスの低下/クラッシュ

- Webサーバなどには致命的

原因：開発者によるメモリ解放忘れ

- が、
複雑な制御フローから
解放忘れを特定することは困難

```
int *gp;
int main(){
    gp = (int *)malloc(4);
    *gp = 100;
    //free(gp);
    return 0;
}
```

メモリリーク
(単純な解放忘れ)

```
void func(){
    int *ip;
    ip = (int *)calloc(10,4);
    if(FLAGS) return;
    free(ip);
    return;
}
int main(){
    func();
    return 0;
}
```

メモリリーク
(例外処理)

問題背景： リークの分類：到達可能性

リークの種類

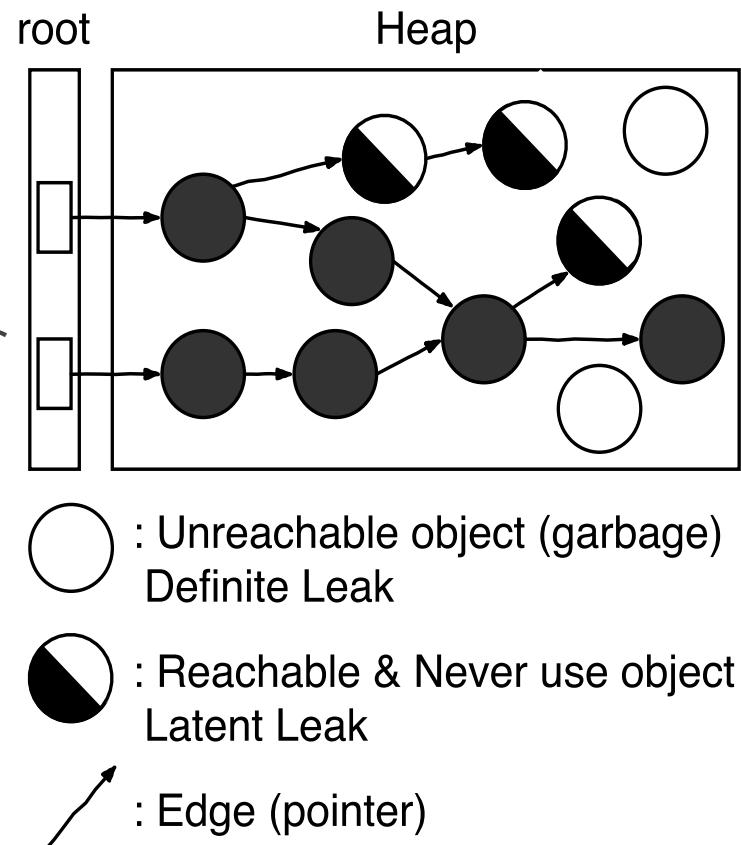
- 到達不可能なリーク
 - レジスタやstack変数内のポインタから到達できないオブジェクト
- 到達可能なリーク
 - 到達可能だが将来アクセスがないオブジェクト

到達可能性の判定

- 型情報に基づきポインタを特定

ガベージコレクション(GC)

- 到達不可能オブジェクトを回収
- **到達可能なリークは回収不可**
 - 脅威度の高い到達可能リーク



問題背景： リークの分類：到達可能性

リークの種類

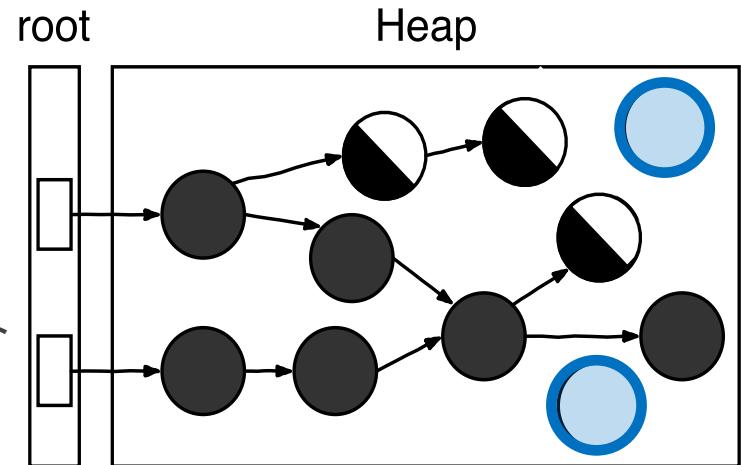
- 到達不可能なリーク
 - レジスタやstack変数内のポインタから到達できないオブジェクト
- 到達可能なリーク
 - 到達可能だが将来アクセスがないオブジェクト

到達可能性の判定

- 型情報に基づきポインタを特定

ガベージコレクション(GC)

- 到達不可能オブジェクトを回収
- **到達可能なリークは回収不可**
 - 脅威度の高い到達可能リーク



: Unreachable object (garbage)

Definite Leak

: Reachable & Never use object

Latent Leak



: Edge (pointer)

問題背景： リークの分類：到達可能性

リークの種類

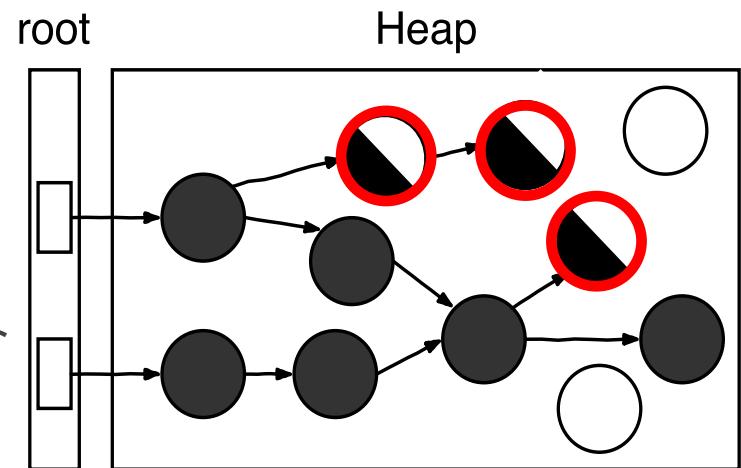
- 到達不可能なリーク
 - レジスタやstack変数内のポインタから到達できないオブジェクト
- 到達可能なリーク
 - 到達可能だが将来アクセスがないオブジェクト

到達可能性の判定

- 型情報に基づきポインタを特定

ガベージコレクション(GC)

- 到達不可能オブジェクトを回収
- **到達可能なリークは回収不可**
 - 脅威度の高い到達可能リーク



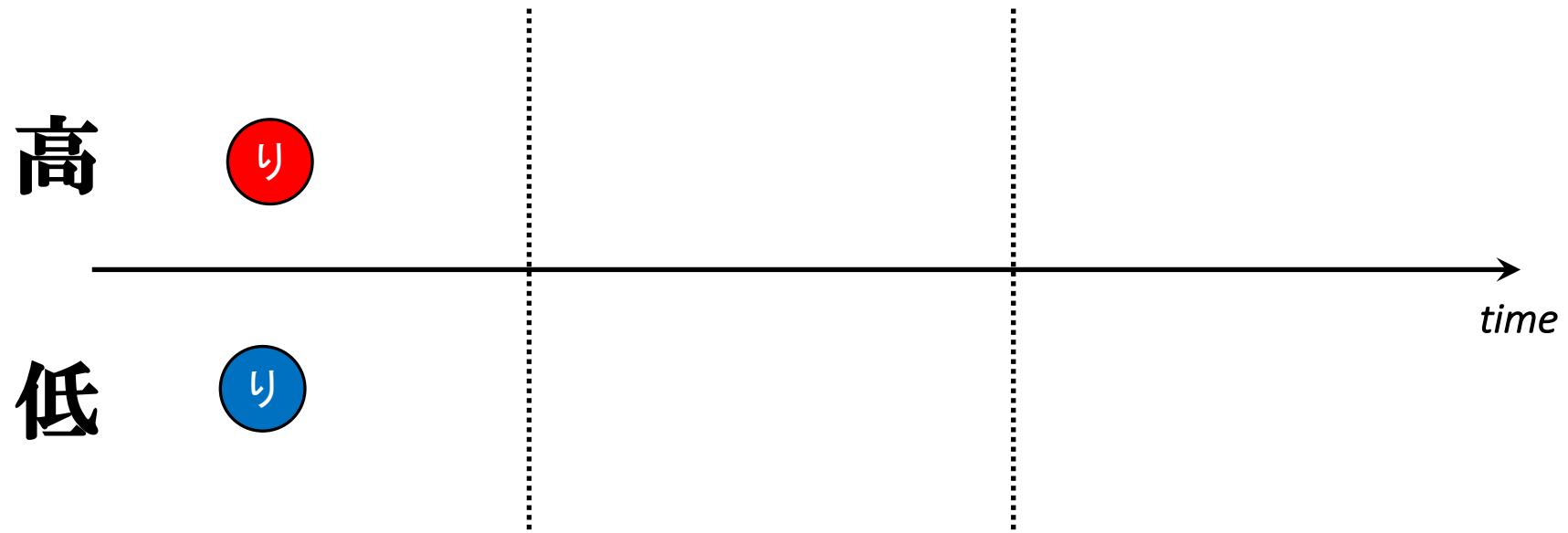
○ : Unreachable object (garbage)
Definite Leak

○ : Reachable & Never use object
Latent Leak

↗ : Edge (pointer)

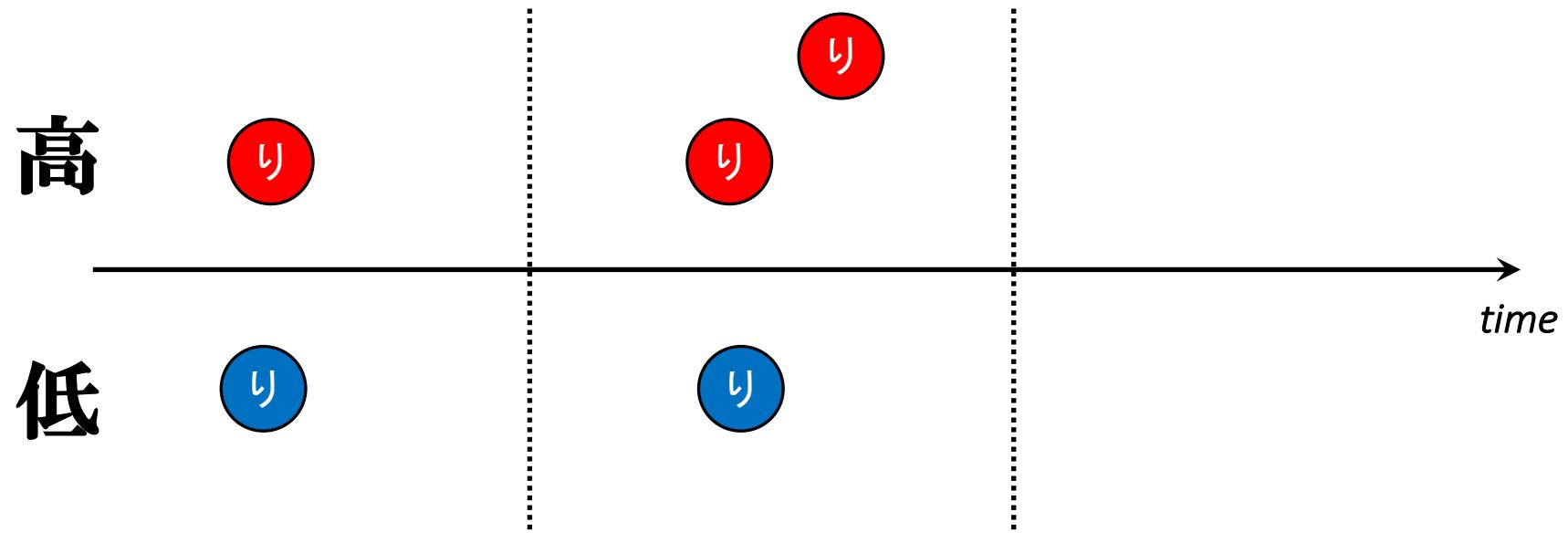
問題背景： リークの分類：脅威度

- ・**高脅威リーク**: 将来に渡って不要オブジェクトが増加するリーク
- ・**低脅威リーク**: 不要オブジェクトは存在するが増加はしないリーク



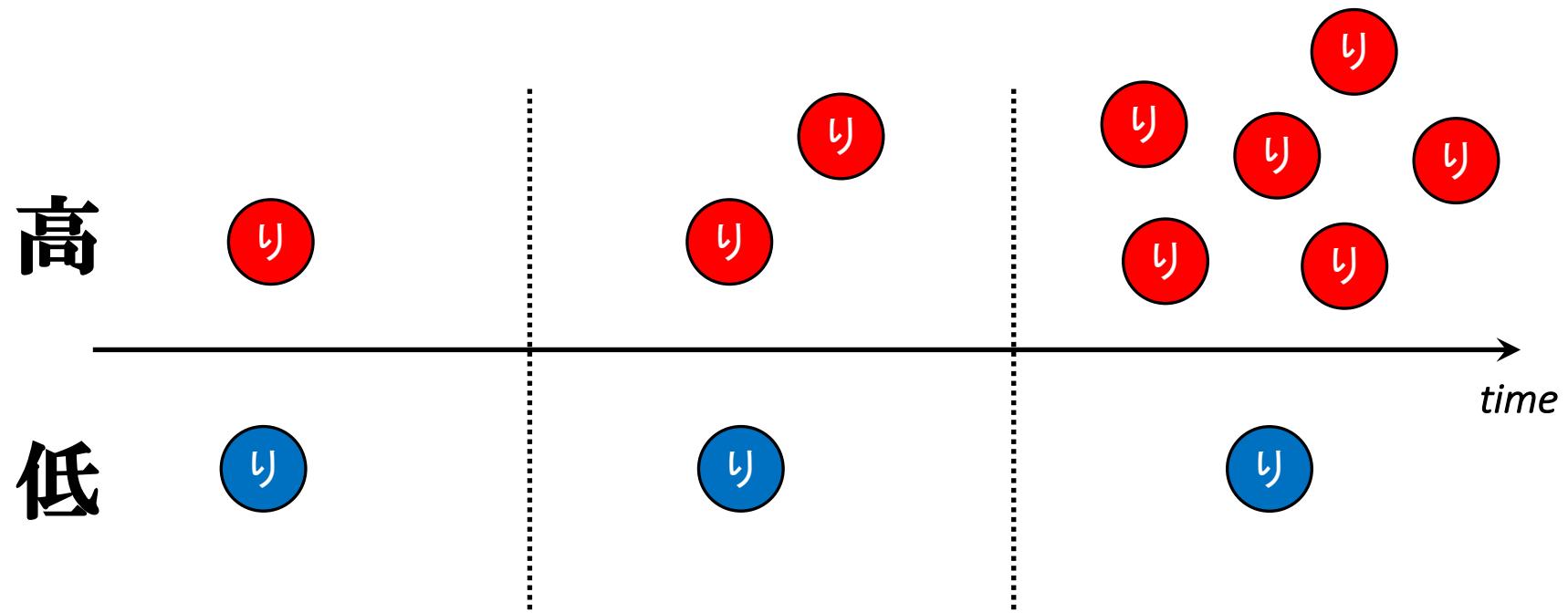
問題背景： リークの分類：脅威度

- ・**高脅威リーク**: 将来に渡って不要オブジェクトが増加するリーク
- ・**低脅威リーク**: 不要オブジェクトは存在するが増加はしないリーク



問題背景： リークの分類：脅威度

- ・**高脅威リーク**: 将来に渡って不要オブジェクトが増加するリーク
- ・**低脅威リーク**: 不要オブジェクトは存在するが増加はしないリーク



問題背景： 既存研究(動的リーク検出)

	GC	型情報	対象種類	対象脅威度	判定粒度	尺度
Java/ C#	Garbage Collection	必要	到達不可	区別なし	オブジェクト	Pointer-Reachability
	Cork [POPL '07]	必要	到達可	高	グループ	Growth
	Leak Pruning [ASPLOS'09]	必要			オブジェクト	Staleness & データ型
C/C++	LEAKPOINT [ICSE '10]	不要	到達不可	区別なし	オブジェクト	Reference-Counter
	SWAT [ASPLOS'04]	不要	到達可 / 到達不可			Staleness
Pikelet	不要	不要	到達可 / 到達不可	高/低	グループ	Growth & Staleness

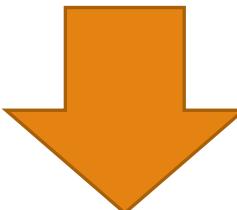
問題背景： 既存研究(動的リーク検出)

	GC	型情報	対象種類	対象脅威度	判定粒度	尺度
Java/ C#	Garbage Collection	必要	到達不可	区別なし	オブジェクト	Pointer-Reachability
	Cork [POPL '07]	必要	到達可	高	グループ	Growth
	Leak Pruning [ASPLOS'09]	必要			オブジェクト	Staleness & データ型
C/C++	LEAKPOINT [ICSE '10]	不要	到達不可	区別なし	オブジェクト	Reference-Counter
	SWAT [ASPLOS'04]	不要	到達可 / 到達不可		オブジェクト	Staleness
Pikelet	不要	不要	到達可 / 到達不可	高/低	グループ	Growth & Staleness

問題背景： 既存研究(動的リーク検出)

	GC	型情報	対象種類	対象脅威度	判定粒度	尺度
Java/ C#	Garbage Collection	必要	到達不可	区別なし	オブジェクト	Pointer-Reachability
	Cork [POPL '07]	必要	到達可	高	グループ	Growth
	Leak Pruning [ASPLOS'09]	必要			オブジェクト	Staleness & データ型
C/C++	LEAKPOINT [ICSE '10]	不要	到達不可	区別なし	オブジェクト	Reference-Counter
	SWAT [ASPLOS'04]	不要	到達可 / 到達不可			Staleness
Pikelet	不要	不要	到達可 / 到達不可	高/低	グループ	Growth & Staleness

モチベーション： 既存手法の問題点

- ・**脅威度を未考慮** [ASPLOS'04, ICSE'10] または
 - ・高脅威リークを検出する仕組みはあるが**GC依存**
[POPL'07, ASPLOS'09]
- 
- ・型情報/GCなしで実現可能な脅威度を考慮した検出法がない
 - 意訳：
 - C/C++バイナリ(型情報なし)を対象に高脅威リークを検出したいっす

提案手法： Calling Contextによるグループ化

脅威度をオブジェクトグループに適用

- オブジェクト個々では脅威度は評価不可
- Calling Contextによってグループ化

Calling Context (CC) :

- コールスタック情報+プログラムポイント情報
- 提案手法内では実質コールスタックと等価
 - 割り付け関数のコール時のCCによって分類するため
- アドレス値から導出可能
 - =動的バイナリ解析で実現可能

```
1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
```

提案手法： Calling Contextによるグループ化

脅威度をオブジェクトグループに適用

- オブジェクト個々では脅威度は評価不可
- Calling Contextによってグループ化

Calling Context (CC) :

- コールスタック情報+プログラムポイント情報
- 提案手法内では実質コールスタックと等価
 - 割り付け関数のコール時のCCによって分類するため
- アドレス値から導出可能
 - =動的バイナリ解析で実現可能

malloc():2

init():7

main():

CC:
G1

```
1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
```

提案手法： Calling Contextによるグループ化

脅威度をオブジェクトグループに適用

- オブジェクト個々では脅威度は評価不可
- Calling Contextによってグループ化

Calling Context (CC) :

- コールスタック情報+プログラムポイント情報
- 提案手法内では実質コールスタックと等価
 - 割り付け関数のコール時のCCによって分類するため
- アドレス値から導出可能
 - =動的バイナリ解析で実現可能

malloc():2

init():7

main():

CC:
G1

calloc():4

action():9

main():

CC:
G2

```
1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
```

提案手法： スナップショット

スナップショット：

- 各グループへの 計測結果評価/リーク判定 处理

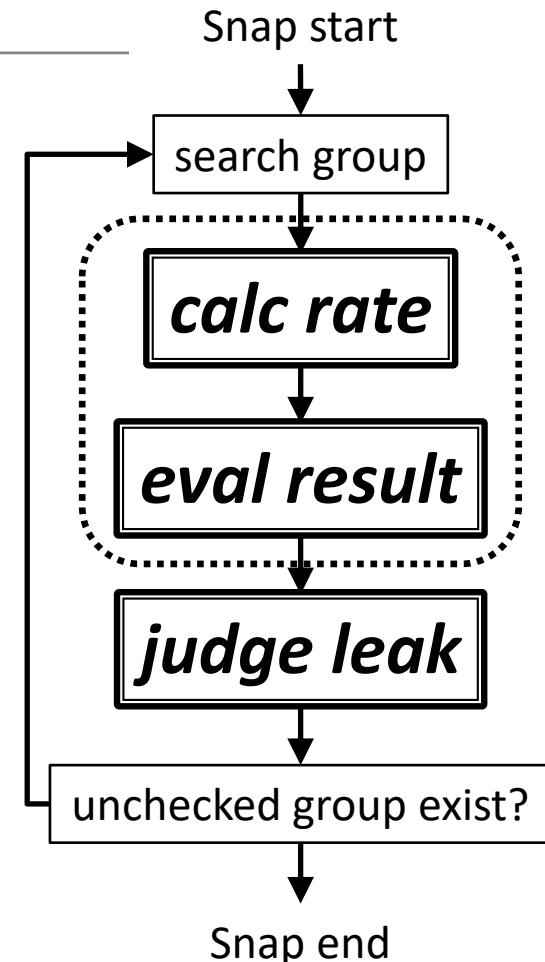
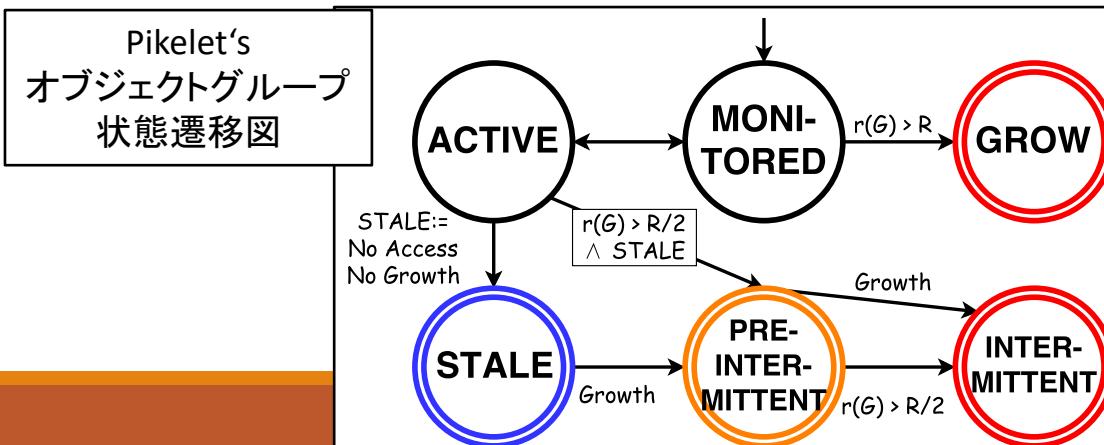
calc rate : 成長率の計算

- 後述: Growth/Intermittence Approach内で使用

eval result : 計測結果を評価、履歴に記録

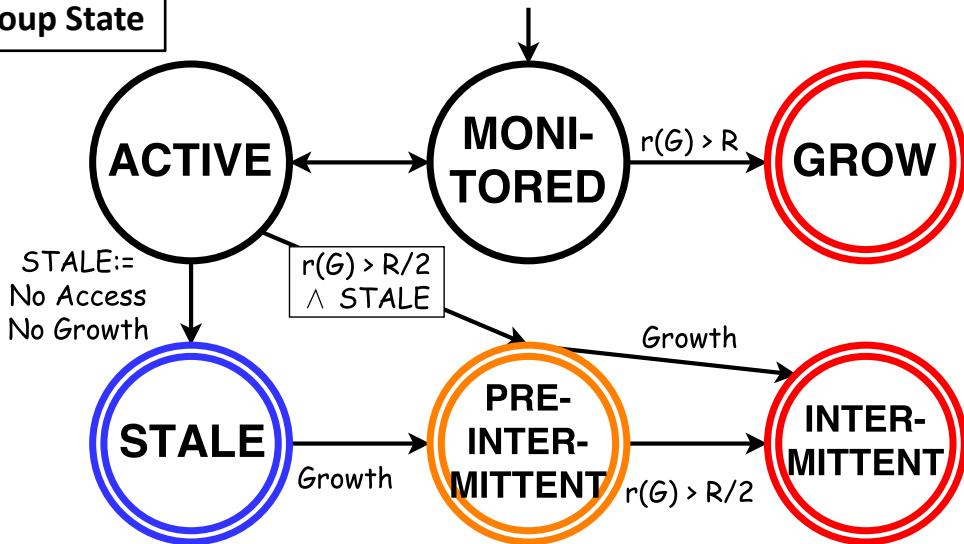
- Staleness Approach内で使用

judge leak : レート/状態評価からリーク判定

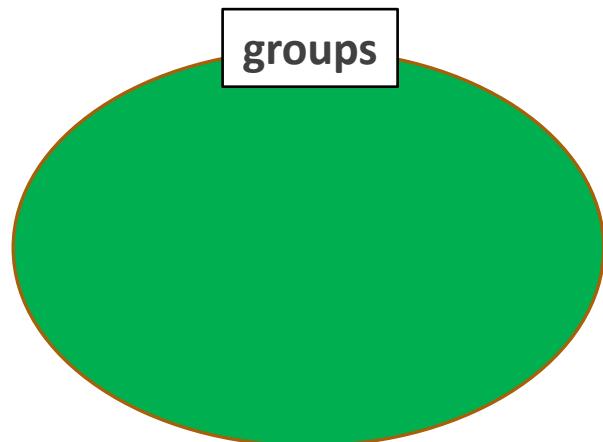
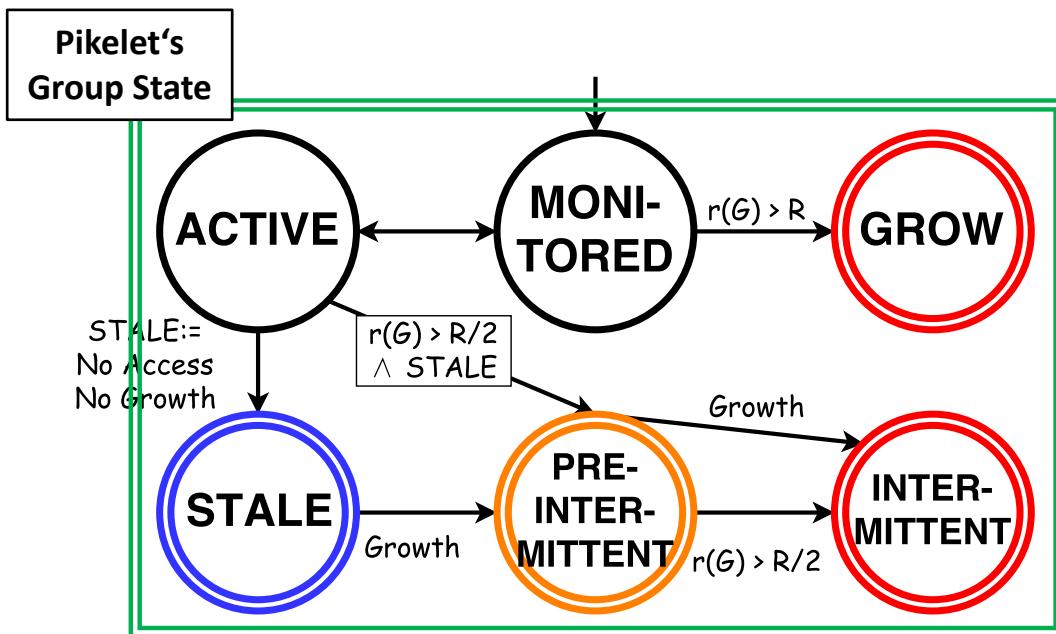


提案手法： グループの状態遷移

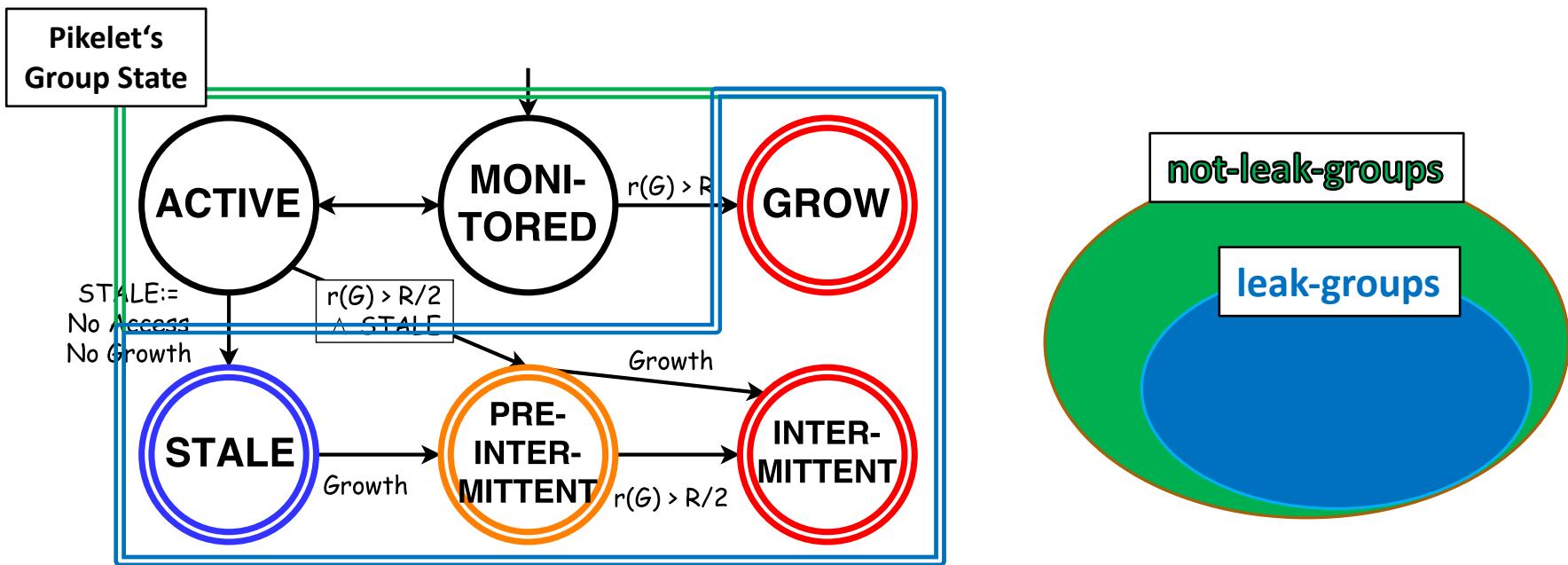
Pikelet's
Group State



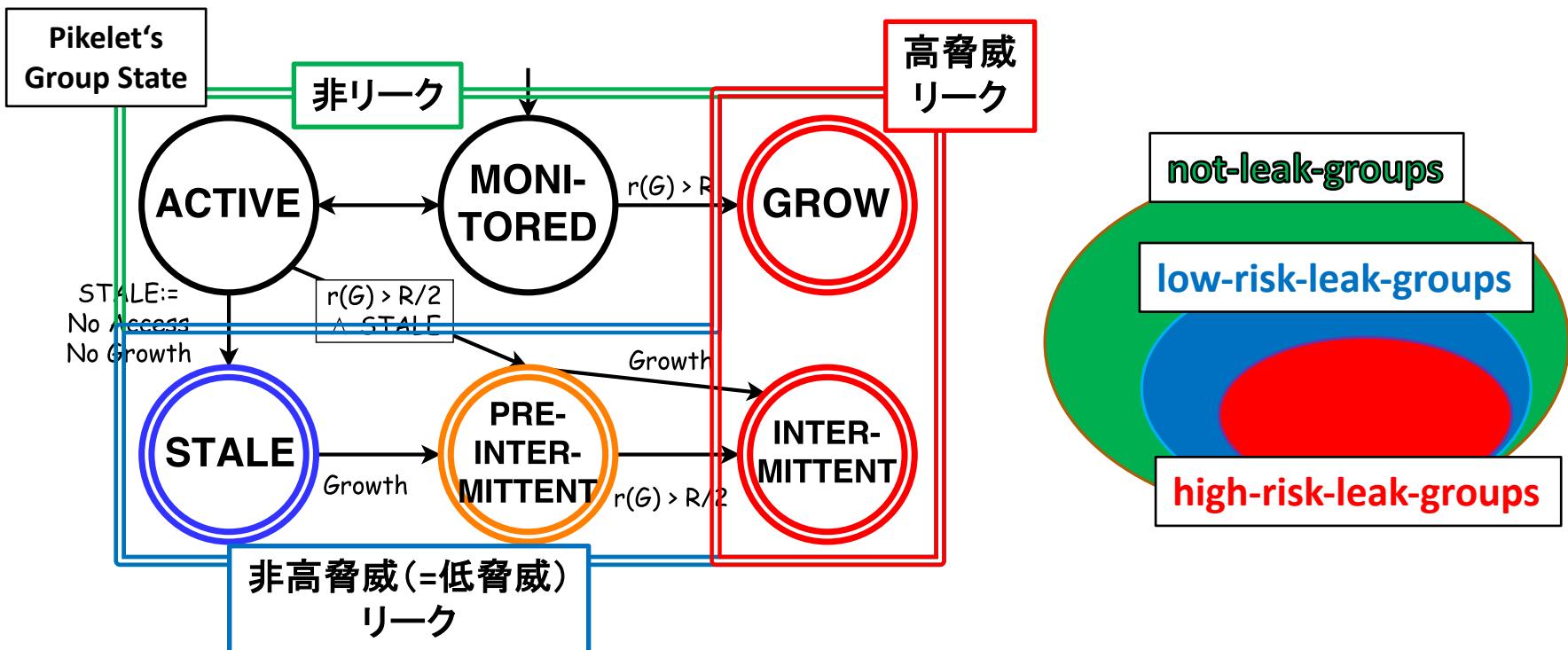
提案手法： グループの状態遷移



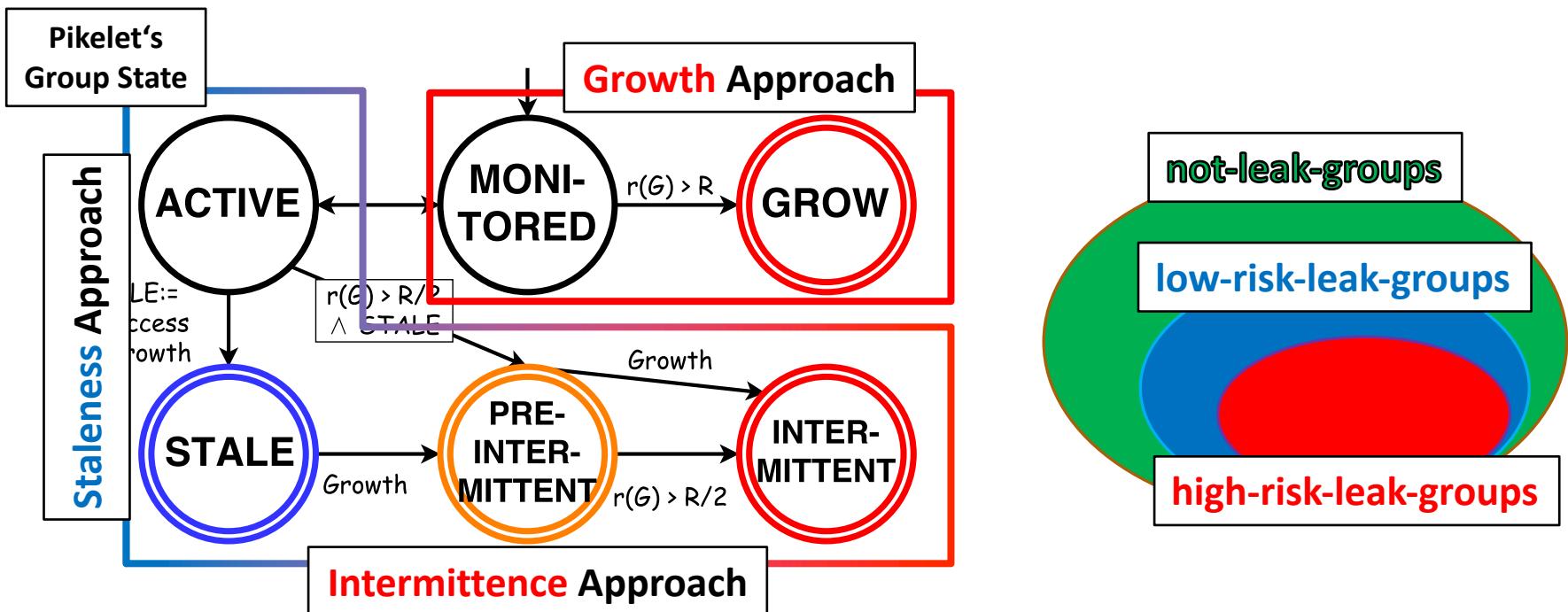
提案手法： グループの状態遷移



提案手法： グループの状態遷移



提案手法： グループの状態遷移



高脅威リークの検出手法：

- Growth Approach
- Intermittence Approach

低脅威リークの検出手法：

- Staleness Approach

提案手法： Growth Approach

高脅威リークとなるグループ

- グループの総サイズ(V)が増加する(成長する)過程が存在

Cork[POPL'07]のRatio Ranking Techniqueを基礎に成長率(r)を算出

- 閾値を超えたなら高脅威リークと判定
- GCなし言語なので各グループの最大サイズの推移を測定

i 回目のスナップショット(計測)で各グループについてレートを算出

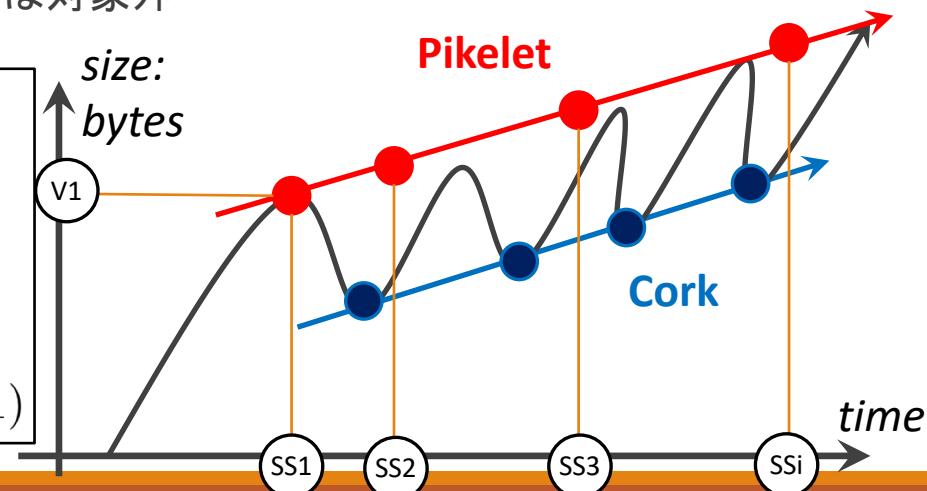
- 未解放オブジェクトを持たないグループは対象外

$$V_i^{\max}(G) = \max(V_{i-1}^{\max}(G), V_i(G))$$

$$Q_i^{\max}(G) = \frac{V_i^{\max}(G)}{V_{i-1}^{\max}(G)}$$

$$p_i(G) = i - \text{Spawn}(G)$$

$$r_i(G) = r_{i-1}(G) + p_i(G) * (Q_i^{\max}(G) - 1)$$



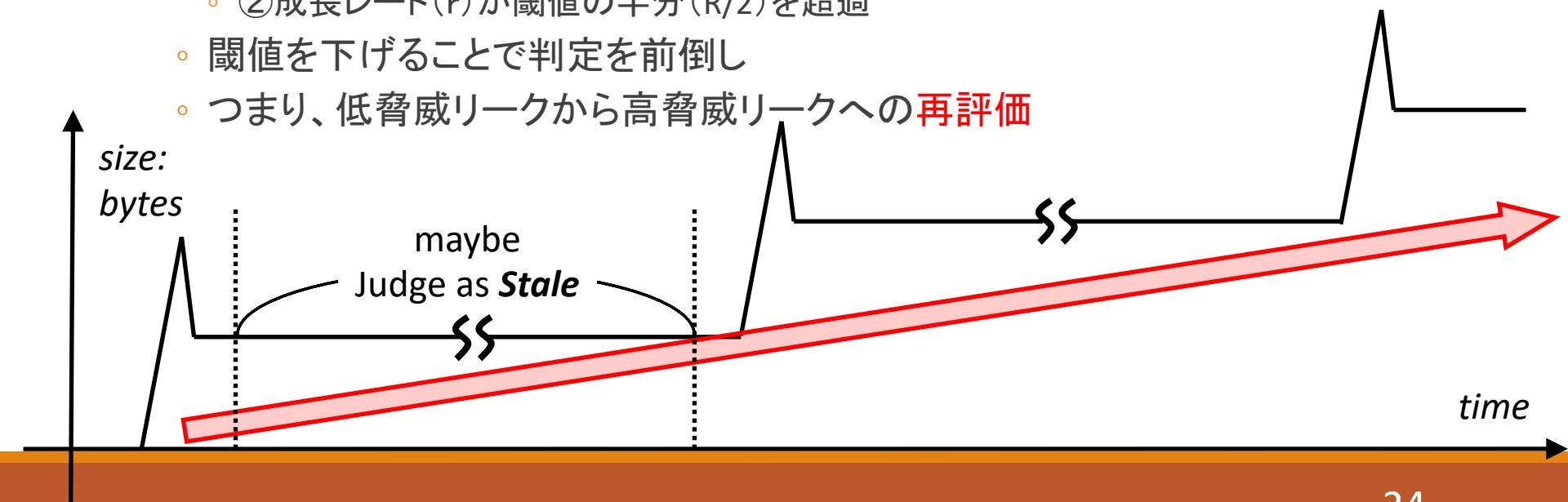
提案手法： Intermittence Approach

低頻度の成長(Intermittence)があるグループ

- Staleness Approachによって低脅威と判定(後述)される機会が存在

高脅威リーク状態 : **INTERMITTENT**

- 次の二条件の両方満たしたら遷移(同時に満たされることはない)
 - ①STALE判定を受けた後に成長
 - ②成長率(r)が閾値の半分($R/2$)を超過
- 閾値を下げることで判定を前倒し
- つまり、低脅威リークから高脅威リークへの**再評価**



提案手法： Staleness Approach

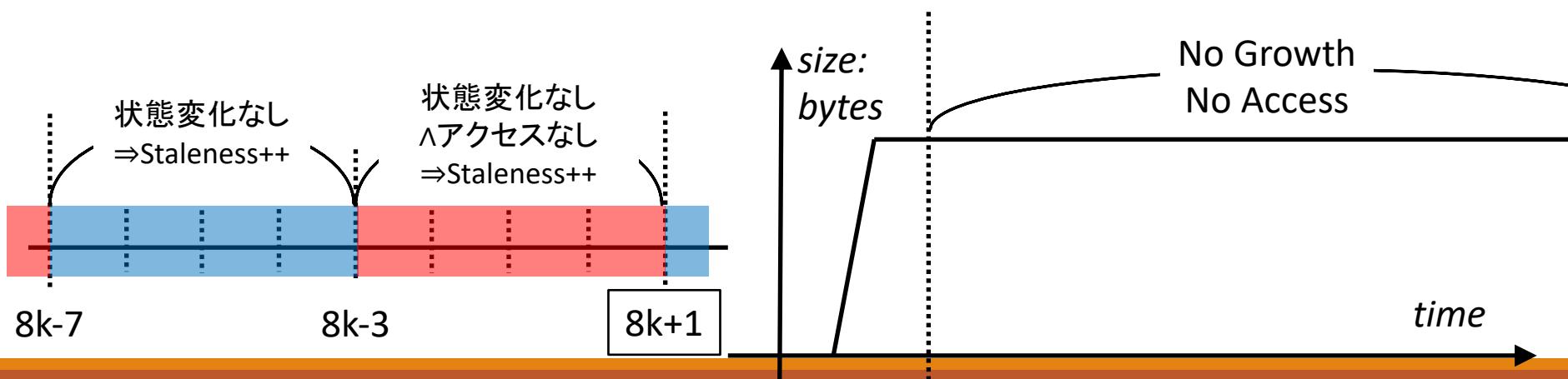
低脅威リークとなるグループ

- = 高脅威リークではない = 成長がないグループ
- = 非リークではない = アクセスがないグループ

グループ単位のStalenessで評価

- Staleness := 成長もアクセスもない状態で経過したスナップショットの数
 - 正確にはアクセスがないと想定される期間

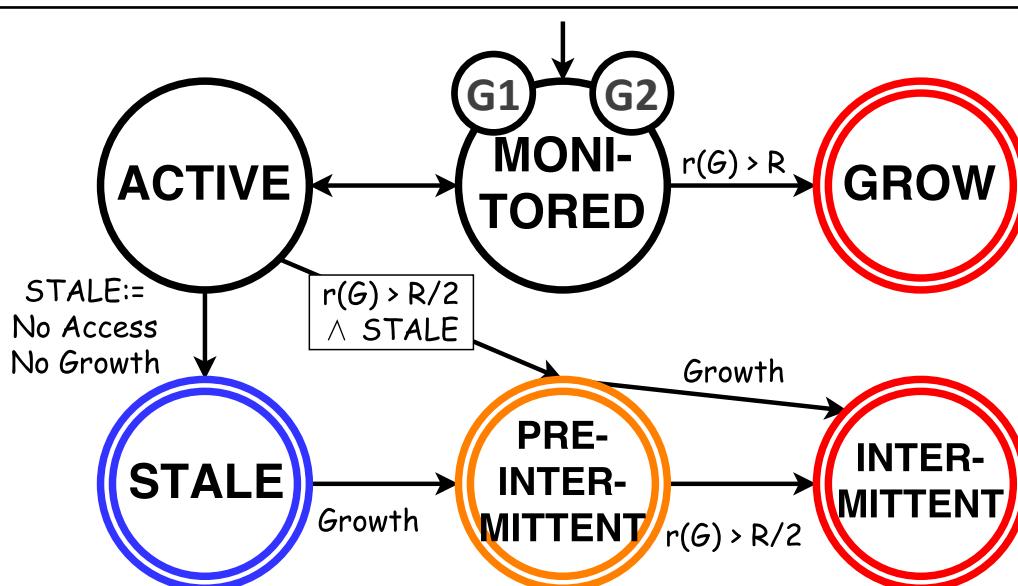
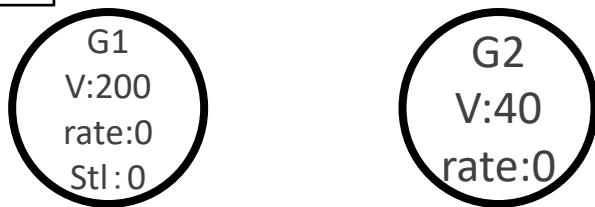
8k+1回目のスナップショットでStalenessが8以上なら低脅威リーク



提案手法: CCでグループ化したオブジェクト群の成長状態管理

Snap:1
入力毎

R=10



```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }

```

malloc():2
init():7
main():

CC: G1

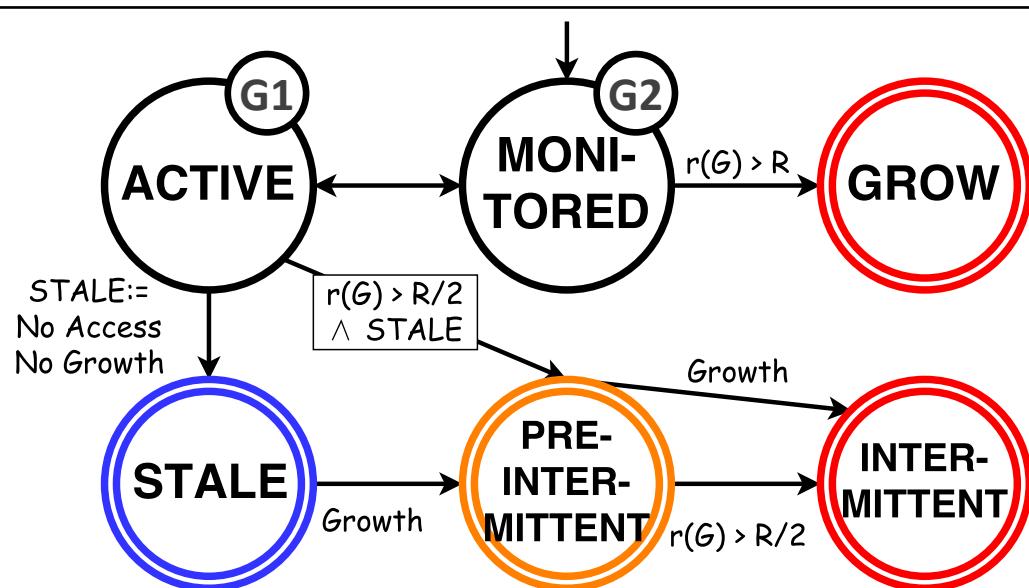
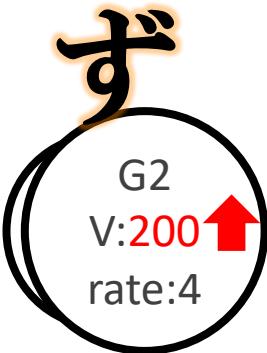
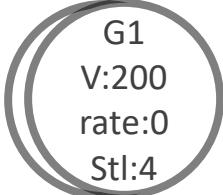
calloc():4
action():9
main():

CC: G2

提案手法: CCでグループ化したオブジェクト群の成長状態管理

Snap:5
入力毎

R=10



```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
```

malloc():2
init():7
main():

CC: G1

calloc():4
action():9
main():

CC: G2

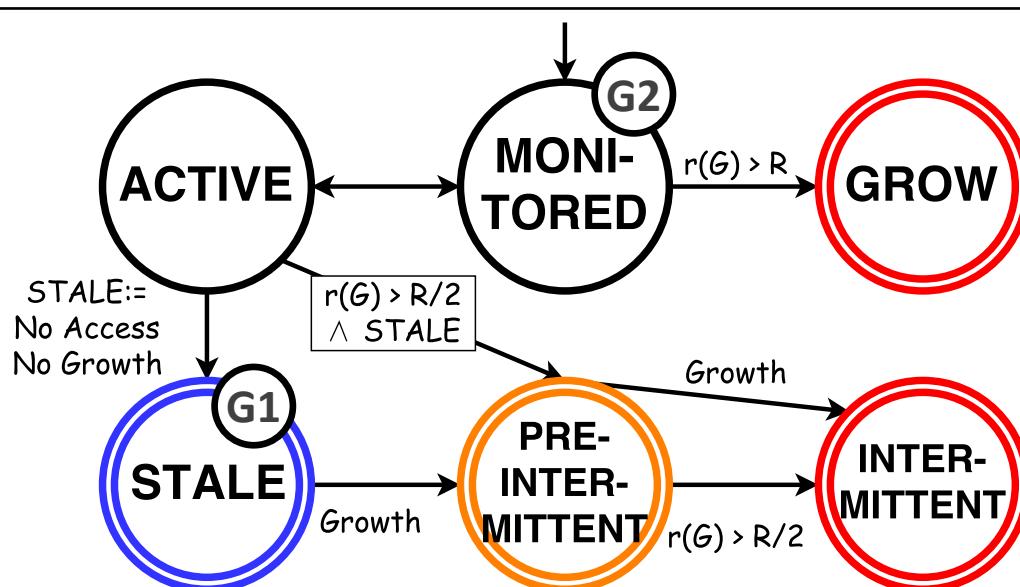
提案手法: CCでグループ化したオブジェクト群の成長状態管理

Snap:9
入力毎

R=10

G1
V:200
rate:0
Stl:8

すも
G2
V:360
rate=8

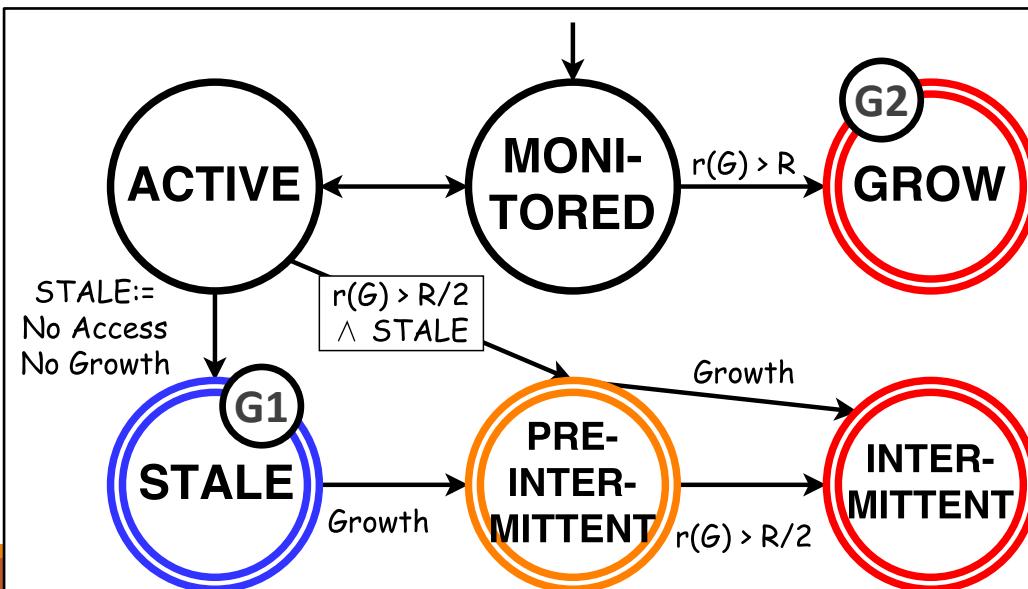
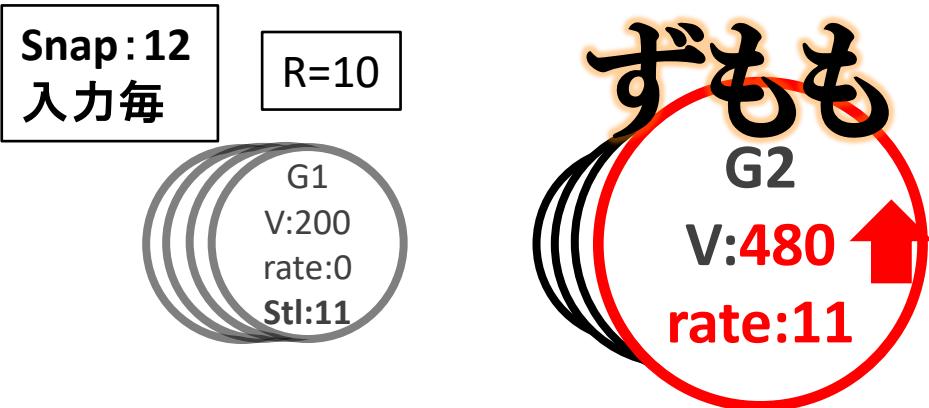


```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
  
```

malloc():2	calloc():4
init():7	action():9
main(): CC: G1	main(): CC: G2

提案手法: CCでグループ化したオブジェクト群の成長状態管理



```

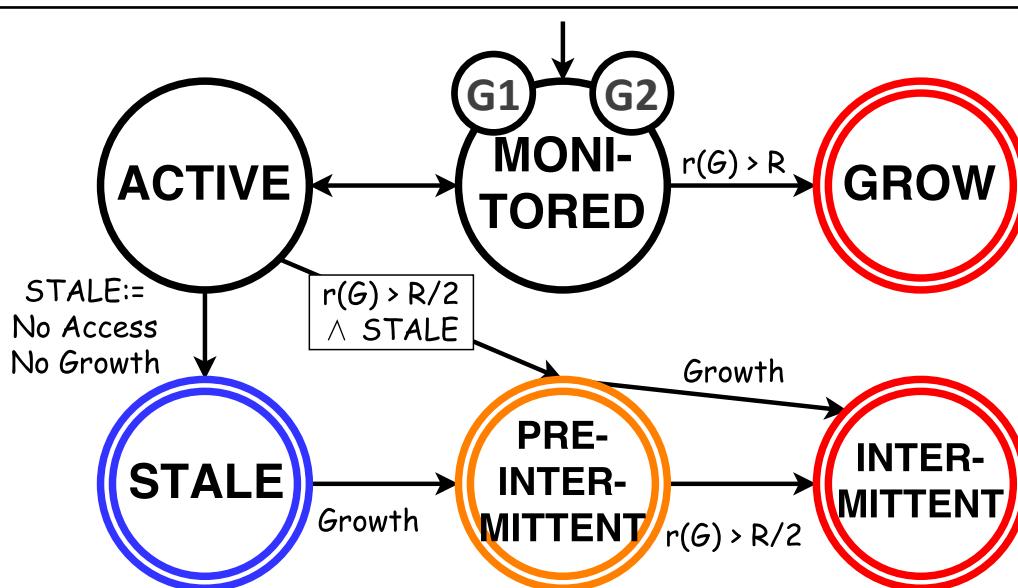
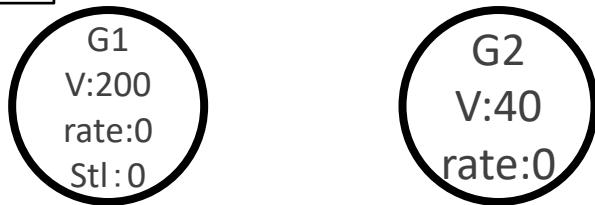
1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
  
```

malloc():2	calloc():4
init():7	action():9
main():	CC: G1
main():	CC: G2

提案手法: CCでグループ化したオブジェクト群の成長状態管理

Snap:1
入力毎

R=10



```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
```

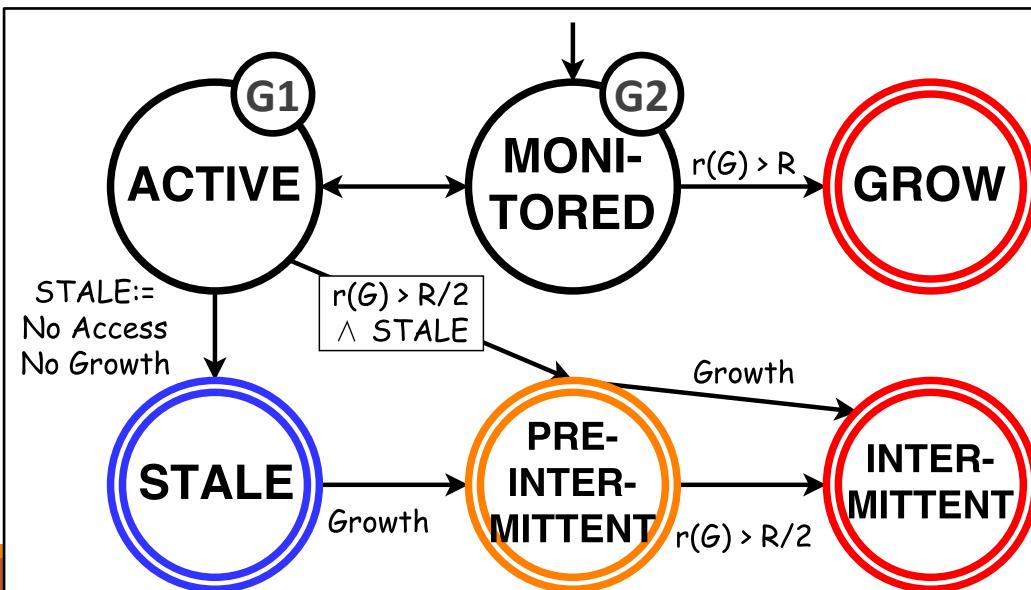
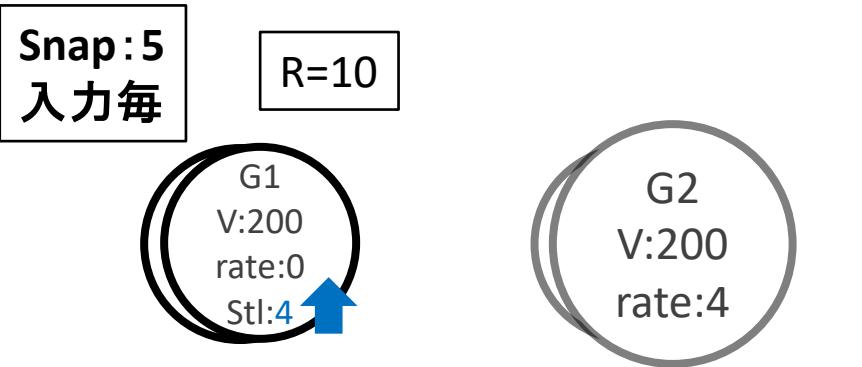
malloc():2
init():7
main():

CC: G1

calloc():4
action():9
main():

CC: G2

提案手法: CCでグループ化したオブジェクト群の成長状態管理

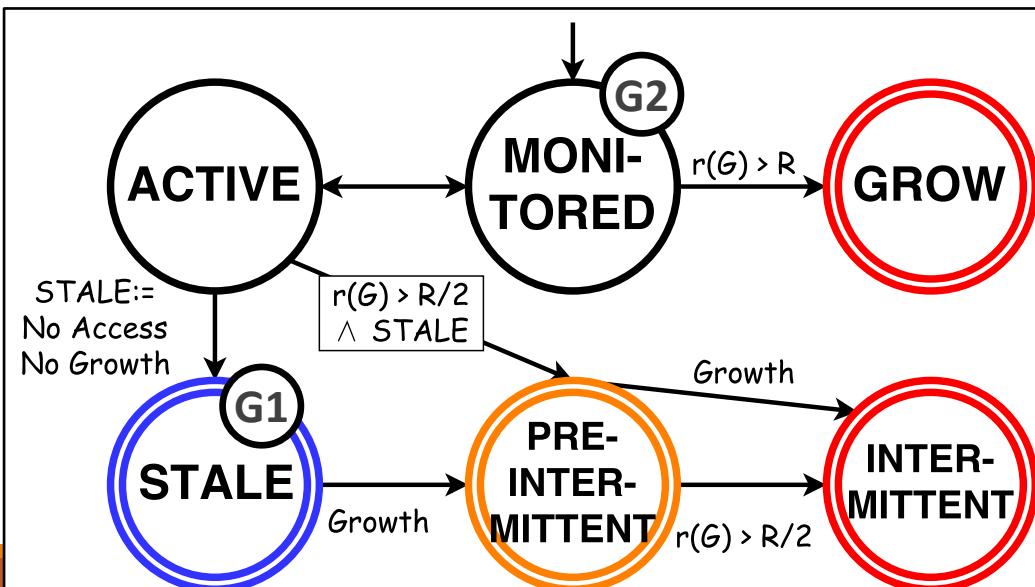
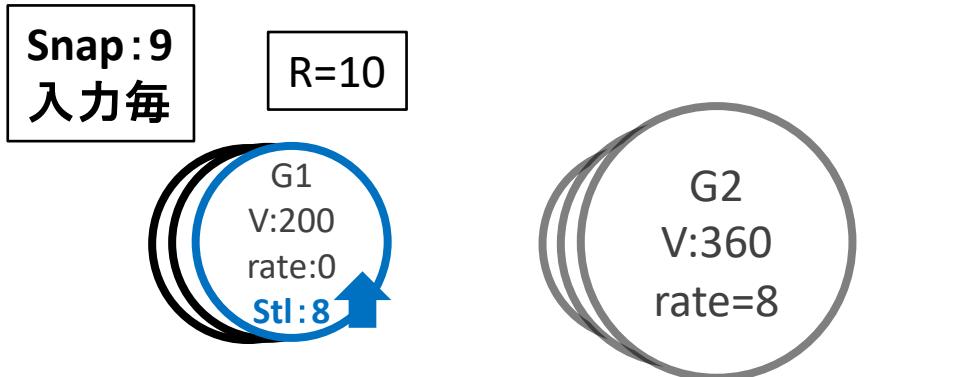


```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
  
```

malloc():2	calloc():4
init():7	action():9
main(): CC: G1	main(): CC: G2

提案手法: CCでグループ化したオブジェクト群の成長状態管理



```

1 int *p, *q;
2 init(){ p=(int *)malloc(200); }
3 action(int input){
4     q=(int *)calloc(10, 4);
5 }
6 main(){
7     init();
8     while(1){
9         action(input());
10    }
11    fini();
12    return 0;
13 }
  
```

malloc():2	calloc():4
init():7	action():9
main():	CC: G1
	CC: G2

実装：

Pin[PLDI'05](ver.3.2-81201)を用いてPikeletとSWAT[ASPLOS'04]を実装

- calling contextはPCC[OOPSLA'07]を用いて識別
 - PCC(Probabilistic Calling Context) : calling contextのハッシング技術
- 各種メタデータの用意

	管理方式	フィールド
オブジェクト	平衡二分木	先頭アドレス、サイズ、最終アクセス時刻...
グループ	平衡二分木	PCC値、オブジェクト数、総サイズ、成長レート...

- SWAT[ASPLOS'04] : 比較用既存研究
 - 特徴: Staleness解析、サンプリング
 - Staleness解析: 長時間アクセスがなければリークと判定
 - サンプリング: アクセス監視を間引くことで実行オーバーヘッドを軽減

実験： 実験環境

実施した実験は2種類

- 精度評価実験
- 実行オーバーヘッド評価実験

実験環境	VM Host	VM Guest
OS	High Sierra 10.13.2	Ubuntu14.04 Linux 3.13.0-137-generic
Processor	Intel Core i7 7567u	Intel Core i7 7567u
Core / Freq	2 / 3.5GHz	2 / 3.5GHz
Memory	16GB	3.9GB
Storage	500GB	101.3GB
	OS Emulator	
name	VirtualBox	
version	5.1.30 r118389 (Qt5.6.3)	

実験： 精度評価

目的:Pikeletが既存手法よりも精度的に優れることを示す

指標:Recall/Precision、RO-Recall/Precision

- RO: Risk-Oriented: 高脅威リークのみに焦点を当てた指標
- オブジェクト数から算出

共通手法:各検体の正解セットを用意

- 非リーク/低脅威リーク/高脅威リークのグループを手動で仕分け

一部の検体に用いた手法:freeの間引き

- 対象:lighttpd-1.4.19
- 処理:実行されるfreeをランダムに間引く(間引き率20%)
- 意図:高脅威リークの注入

実験: 精度評価

Risk-Oriented

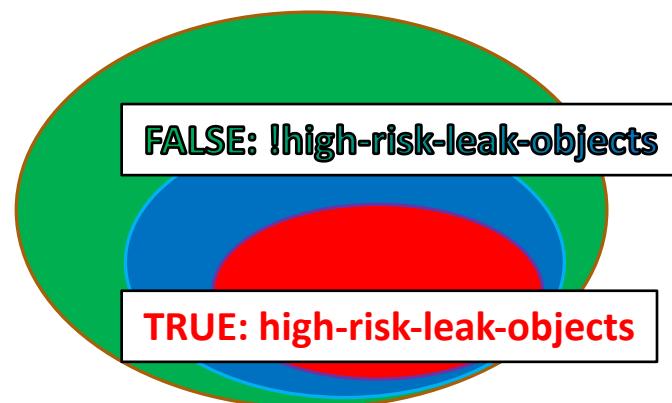
Normal-Recall/Precision

- リークか否かの精度
- 低高区別なし



Risk-Oriented-Recall/Precision

- 高脅威リークのみの精度
- 低脅威リークを高脅威リークと報告すると誤判定(False Positive)
- 具体例：
 - **ずもも(G2)**を高脅威と判定なら正解
 - **ずももではないG1**を高脅威と判定なら誤答

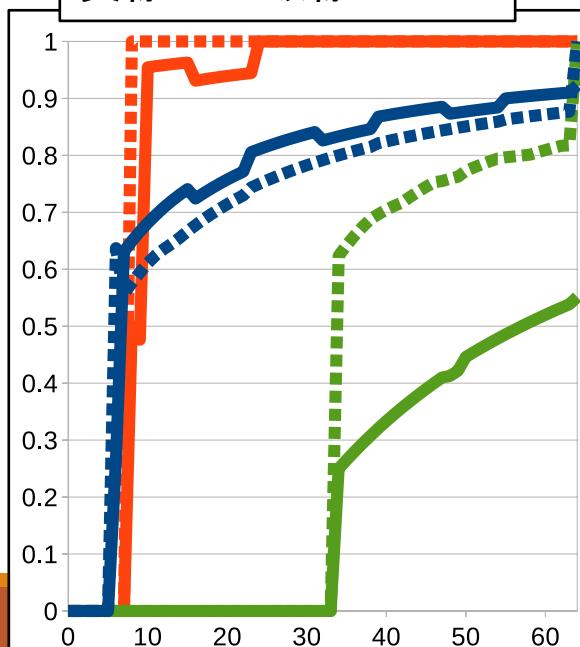


実験: 精度(リークのY/N)

Normal-Recall/Precision for Small Benchmark

program	Static LOC	Dynamic instructions	# of Snap	Groups			Objects		
				Group	# of Leak	High/Low	Object	# of Leak	High/Low
Leak_SEINL	239	95,756,868	64	4	3	2/1	8,800	7,800	6,800/1,000
lighttpd-1.4.19	54,698	35,272,515	50	445	94	18 /76	7,426	2,670	2,541/129

赤: Pikelet 青緑: SWAT
実線: NR 破線: NP



結果:

- SWATは見逃し、誤検出が存在
- Pikeletは両指標で100%を達成

要因:

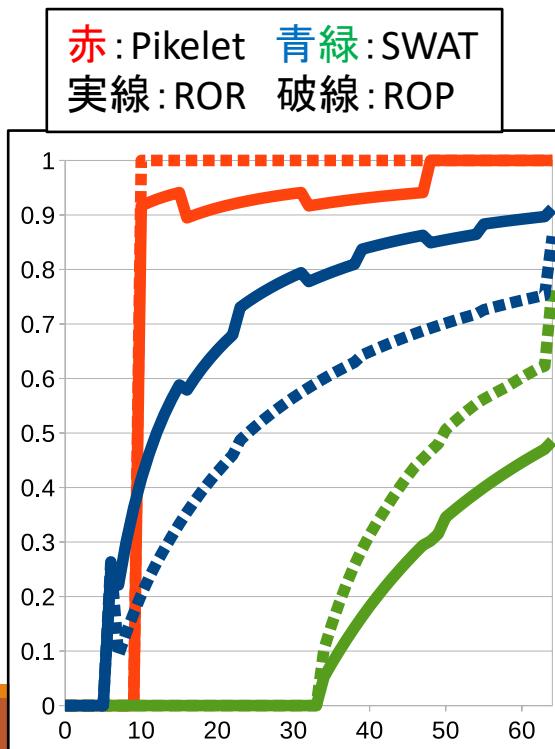
- SWATは即時判定できず見逃しが発生
- Pikeletはグループ化により即時判定

結論:

- グループ化で見逃しと誤検出が抑制可能

実験: 精度(高脅威: Risk-Oriented) RORecall/ROPrecision for Small Benchmark

program	Static LOC	Dynamic instructions	# of Snap	Groups			Objects		
				Group	# of Leak	High/Low	Object	Leak	High/Low
Leak_SEINL	239	95,756,868	64	4	3	2/1	8,800	7,800	6,800/1,000
lighttpd-1.4.19	54,698	35,272,515	50	445	94	18 /76	7,426	2,670	2,541/129



結果:

- SWATは見逃し、誤検出が存在
- Pikeletは両指標で100%を達成

Normal-
Recall/Precision
とほぼ同じ動き

要因:

- SWATは即時判定できず見逃しが発生
- Pikeletはグループ化により即時判定

結論:

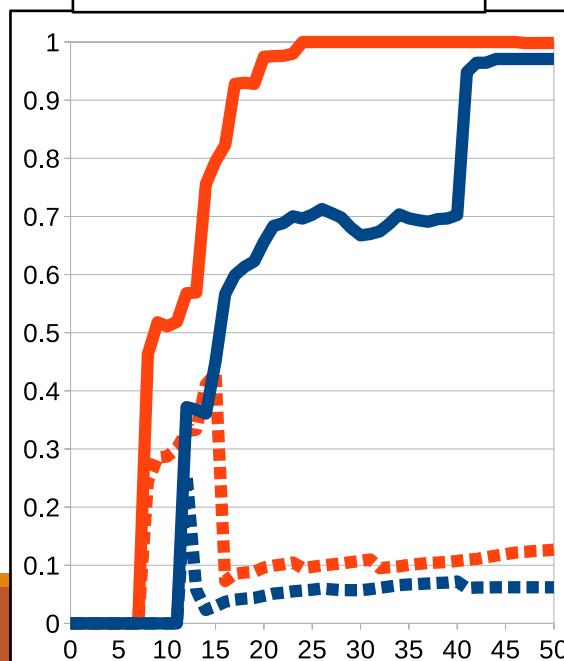
- グループ化で見逃しと誤検出が抑制可能

実験: 精度(リークのY/N)

Normal-Recall/Precision for lighttpd-1.4.19

program	Static LOC	Dynamic instructions	# of Snap	Groups			Objects		
				Group	# of Leak	High/Low	Object	# of Leak	High/Low
Leak_SEINL	239	95,756,868	64	4	3	2/1	8,800	7,800	6,800/1,000
lighttpd-1.4.19	54,698	35,272,515	50	445	94	18 /76	7,426	2,670	2,541/129

赤: Pikelet 青: SWAT
実線: NR 破線: NP



結果:

- Recallは両検出器で高精度を達成
- Precisionは両検出器で低精度

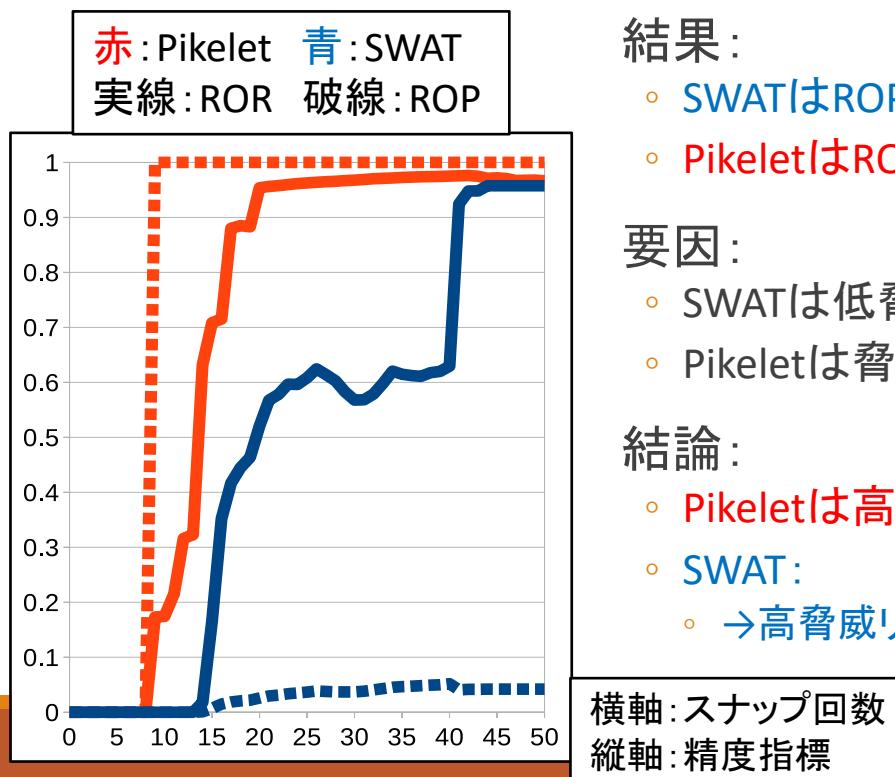
要因: SWAT/Pikeletが積極的に判定を行ったため

結論:

- Pikelet:
 - ROPrecisionの高さから誤検出は低脅威リークでのみ発生
- SWAT:
 - Precisionの低さはROPrecisionに引継

実験: 精度(高脅威: Risk-Oriented) RORRecall/ROPrecision for lighttpd-1.4.19

program	Static LOC	Dynamic instructions	# of Snap	Groups			Objects		
				Group	# of Leak	High/Low	Object	# of Leak	High/Low
Leak_SEINL	239	95,756,868	64	4	3	2/1	8,800	7,800	6,800/1,000
lighttpd-1.4.19	54,698	35,272,515	50	445	94	18 /76	7,426	2,670	2,541/129



結果:

- SWATはROPが小さい
- PikeletはROP/RORで高精度を達成

要因:

- SWATは低脅威も無差別で報告
- Pikeletは脅威度により区別し報告

結論:

- Pikeletは高脅威リークを正確に抽出
- SWAT:
 - →高脅威リークが低脅威リーク/誤検出に埋没

実験： 実行オーバーヘッド評価

目的：

- Pikeletが既存手法と同程度の実行オーバーヘッドを達成することを示す

指標：

- time関数から得られた実行時間

手法：

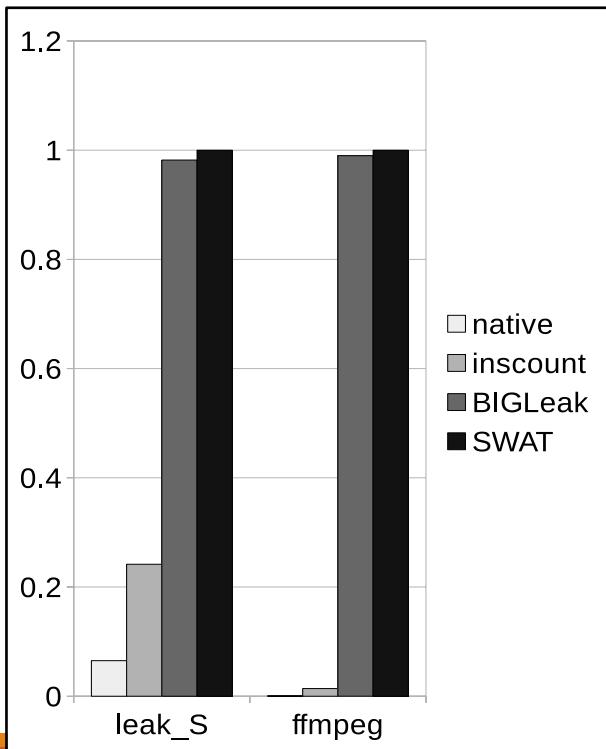
- 検体の**実行5回の計測時間平均**を算出
- Pikelet/SWATでのスナップショット(計測)は各検体32回ずつ実行

比較対象：

- native: 何も適用しない
- inscount: Pikelet、SWATの基礎部分(動的命令数計測)
- Pikelet: 提案手法
- SWAT: 比較手法(グラフはこれを1として正規化)

実験： 実行オーバーヘッド評価

program	LOC	instructions	# of			average time(s)			
			Snap	Groups	Objects	native	inscount	BIGLeak	SWAT
Leak_S	107	391,482,939	32	1	10,000	0.3368	1.2514	5.085	5.1782
FFmpeg	1,268,917	3,369,223,075	32	488	5,168	0.56	11.2276	793.019	801.0892



結果：

- 両プログラムで僅かにPikeletの方が高速

要因：

- スナップショットの高速化
 - $O(N_{obj}) \rightarrow O(N_{grp})$
- (あるいはうまいことキャッシング)

結論：

- 既存研究と同程度のオーバーヘッドを達成**
- アクセス監視を省くことで高速化の可能性
 - アクセス監視は低脅威リーク検出用

結論：

Pikeletの貢献

- C/C++で高脅威リークを検出する技術の提示
- 有用性評価：
 - 既存手法と比較して同程度のオーバーヘッドでより高精度を達成
 - バイナリ解析で実現可能なグループ化の提示

展望：

- マルチスレッドなど未対応技術の消化
- 現在シングルスレッドのみ対応

結論：

Pikeletの貢献

- C/C++で高脅威リークを検出する技術の提示
- 有用性評価：
 - 既存手法と比較して同程度のオーバーヘッドでより高精度を達成
 - バイナリ解析で実現可能なグループ化の提示

展望：

- 検体増やし
 - メモリリークが存在するリアルベンチマークに適用してみる
- マルチスレッドなど未対応技術の消化
 - 現在シングルスレッドのみ対応

参考文献

- [1] Hauswirth, M. and Chilimbi, T. M.: Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling, SIGPLAN Not., Vol. 39, No. 11, pp. 156–164 (online)
- [2] Jump, M. and McKinley, K. S.: Cork: Dynamic Memory Leak Detection for Garbage-collected Languages, Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07, New York, NY, USA, ACM, pp. 31–38 (online)
- [3] Bond, M. D. and McKinley, K. S.: Leak Pruning, SIGARCH Comput. Archit. News, Vol. 37, No. 1, pp. 277–288 (online)
- [4] Clause, J. and Orso, A.: LEAKPOINT: Pinpointing the Causes of Memory Leaks, Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, New York, NY, USA, ACM, pp. 515–524 (online)
- [5] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, New York, NY, USA, ACM, pp. 190–200 (online)
- [6] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. SIGPLAN Not., Vol. 42, No. 10, pp. 97–112, October 2007.

以下質問用スライド

問題背景： Garbage Collection

不要なオブジェクトを自動で解放してくれる機能の総称

- 貢献：メモリリークの一部解消、宙ぶらりんポインタの解消、etc etc...

Mark & Sweep GC:

- ルート(レジスタやスタック変数)から到達不可能なオブジェクトを解放
- 尺度：到達可能性

Reference Counter GC:

- オブジェクト単位で参照数を管理し、参照されなくなったオブジェクトを解放
- 尺度：参照カウント

実装には**ポインタの発見**が必須

- **C/C++ではポインタと整数値が区別できない**

提案手法：グループ化

Calling Contextとは

Calling Context(CC) :

- 特定のプログラムポイントを表現する方法の一つ
- Call Stack(Call Sites) + Target Site
- Targetが特定のCallだと、実質Call Stackの部分集合と等価
- (一般にSiteはソースコードの行目で表されることが多い)

The diagram illustrates the mapping of a source code snippet to a Calling Context (CC) table. On the left, a box contains the source code with line numbers 1 through 8. Line 2 contains a red highlighted word 'target;'. An orange arrow points from this box to a table on the right, which is labeled 'Call Stack' at the bottom.

CC	callee	site
PP	target	2
≡ Call Stack	funcC	4
	funcB	5
	funcA	7
	main	*

提案手法：グループ化

Probabilistic Calling Context

Probabilistic Calling Context[6] :

- 伝搬式を用いてCCをPCC値でハッシングする技術
- PCC値伝搬式: $V_{callee} = (3 * V_{caller} + Call\ Site) \bmod 2^{32}$
- 積和を使い十分な分散を確保
- Call Site値にコンパイラ定数を用いれば高速化も可能
- あくまでハッシングなのでPCC値はCCの中身を表しているわけではない

CC	callee	site
≡ Call Stack	target	2
	funcC	4
	funcB	5
	funcA	7
	main	*



callee	V_callee	V_caller	CS
target	248	82	2
funcC	82	26	4
funcB	26	7	5
funcA	7	0	7
main	0	0	0

提案手法：グループ化 Calling Contextの選択理由

小泉の考えるグループ化の条件

- グループ数が有限に収束
- 原因の特定に役立つ情報を内包=CC情報

Calling Contextの利点

- **バイナリ解析で実現可能**
 - 各アドレス値から導出可能
- ラッパー関数に対応可能
 - Call Siteでは対応不可能
- **型によるグループ化に近似可能**
 - 同一CCを持つならキャストを伴うので同一の型となる

提案手法：スナップショット スナップショットのタイミング

スナップショット(計測)：グループのサイズ(bytes数)を計測

- →成長率の算出、リーク判定、状態遷移も行う

スナップショット(計測)のタイミング：

- 一定数の外部入力毎と設定

一定時間や一定の命令数での実行は汎用性に欠く

- →プログラム特性に大きく依存するため

Webサーバ/データベースプログラムに共通する特徴

- 外部入力の存在
- プログラム特性によるブレをある程度緩和可能と想定

ただし、適切な「一定数」については未検証

提案手法： Ratio Ranking Technique [2]

Cork[2]で採用されている成長を評価する指標

- 成長：時間経過でグループ(型)の総サイズが増加すること

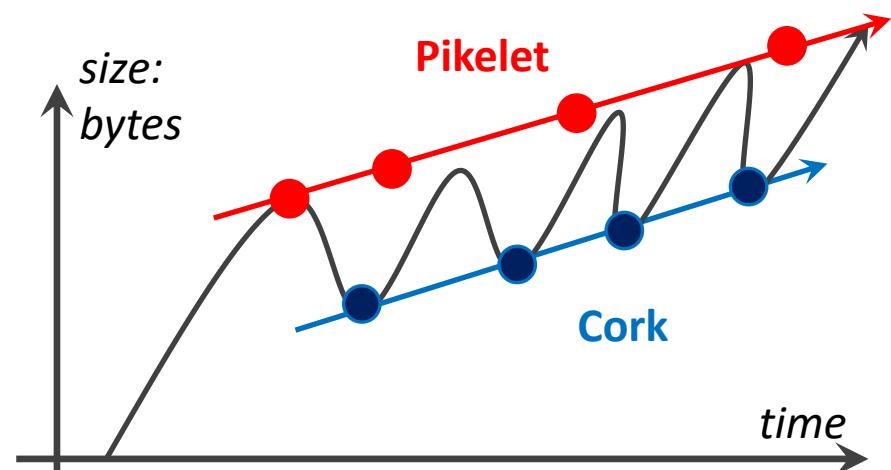
	Cork	Pikelet
グループ化基準	型情報	Allocation Calling Context
V_i	GC後のサイズ	Snapshot時の最大サイズ
タイミング(Snapshot)	GC直後	一定数の入力

Cork's RRT

$$Q_i(T) = \frac{V_i(T)}{V_{i-1}(T)}$$

$$g_i(T) = Q_i(T) - 1$$

$$r_i(T) = r_{i-1}(T) + p_i(T) * g_i(T)$$



提案手法： Intermittence Approach

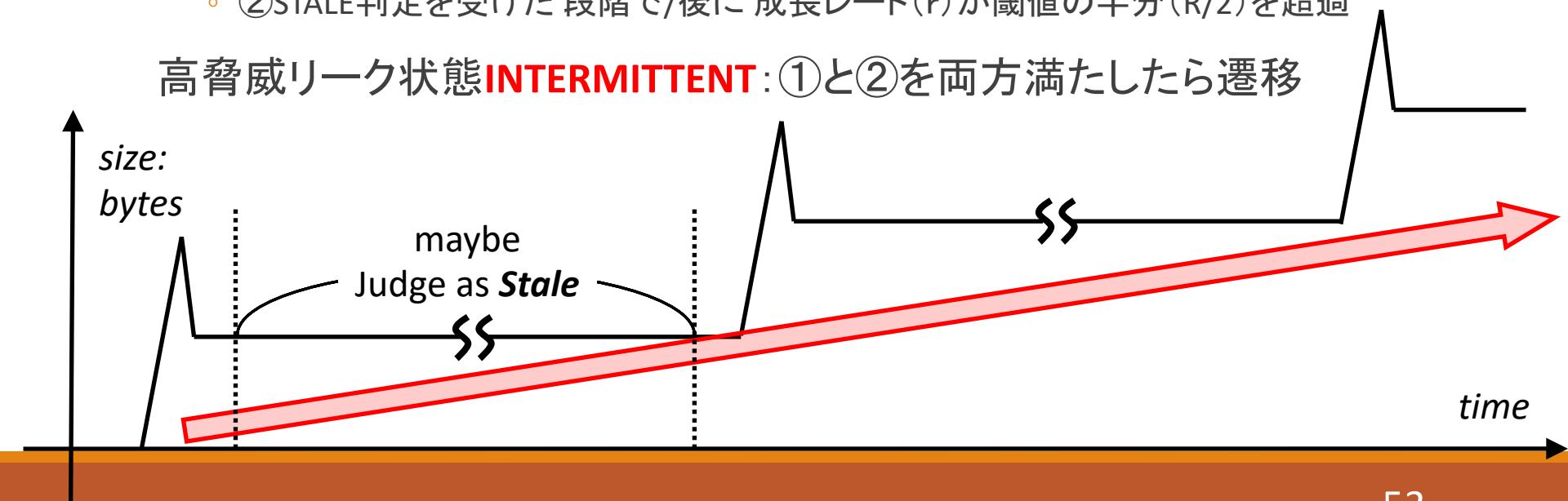
低頻度の成長(Intermittence)があるグループ

- Staleness Approachによって低脅威と誤判定される可能性(後述)

誤判定を解消/予防するための中間状態 **PRE-INTERMITTENT**

- 次の二条件の片方を満たしたら遷移(同時に満たされることはない)
 - ①STALE判定を受けた後に成長
 - ②STALE判定を受けた段階で/後に成長率(r)が閾値の半分($R/2$)を超過

高脅威リーケ状態 **INTERMITTENT**: ①と②を両方満たしたら遷移



提案手法: Staleness Approach アクセスの監視

アクセス: オブジェクトへの参照

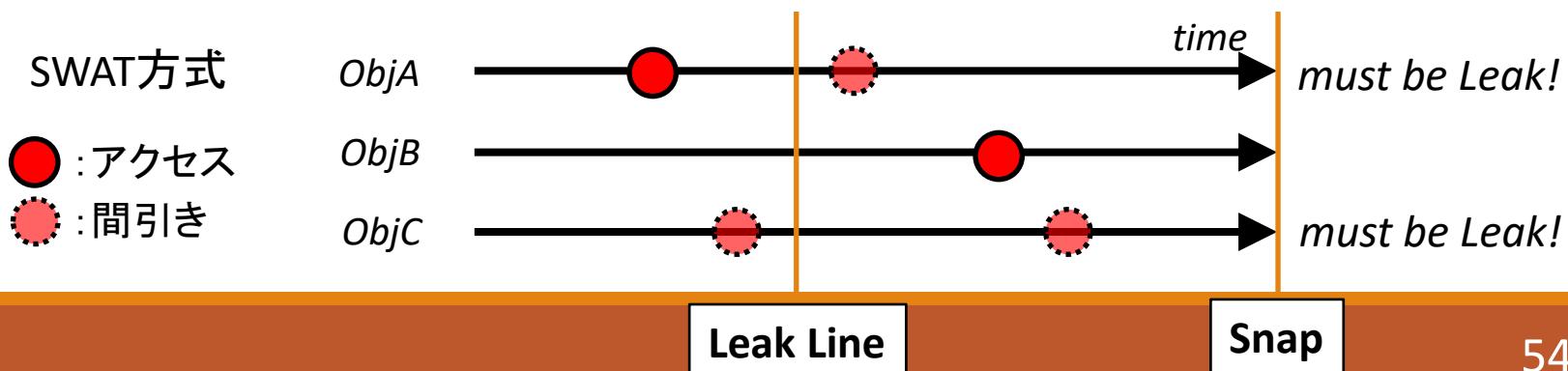
全アクセスの監視は大きな実行オーバーヘッドを生む[1]

- サンプリング技術を用いてオーバーヘッドを削減
- サンプリング技術: ABT(Adaptive Bursty Tracing)[1]

サンプリングによるリーク誤判定: **アクセスを見逃す可能性**

グループに対するアクセスの有無:

- 一定期間内(4 Snaps)のグループ内オブジェクトへのアクセスの**論理和**
- オブジェクト単体へのアクセスの見逃しによる**誤検出を抑制**



提案手法: Staleness Approach アクセスの監視

アクセス: オブジェクトへの参照

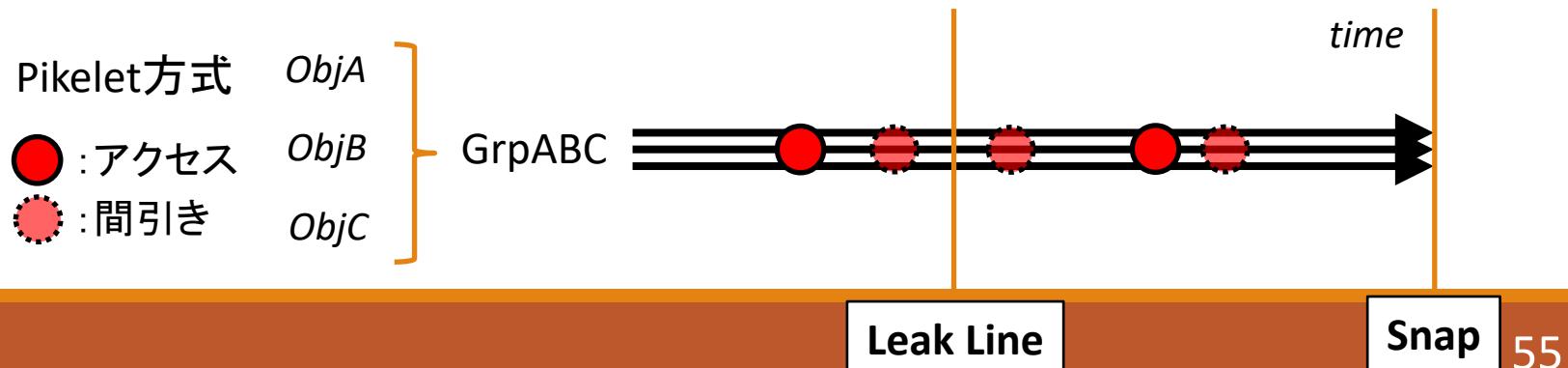
全アクセスの監視は大きな実行オーバーヘッドを生む[1]

- サンプリング技術を用いてオーバーヘッドを削減
- サンプリング技術: ABT(Adaptive Bursty Tracing)[1]

サンプリングによるリーク誤判定: **アクセスを見逃す可能性**

グループに対するアクセスの有無:

- 一定期間内(4 Snaps)のグループ内オブジェクトへのアクセスの**論理和**
- オブジェクト単体へのアクセスの見逃しによる**誤検出を抑制**



提案手法 : Staleness Approach

スナップショット : eval result

計測結果変化 := i回目のスナップショットの評価結果がi-1回目と異なる

評価結果 (SnapshotResult) := (NFG, FG, GN, GVm)

- NFG, FG := <Not> Freedish = 未解放/解放済オブジェクト数の比較結果
- GN := Grow Num of objects = 前回よりもオブジェクト数が増えたか
- GVm := Grow $V^{max}(G)$ = 最大サイズの更新があったか = 成長したか

評価結果を32bitの履歴 (SnapshotResultHistory) に格納

- i-7回目までの直近8回分を格納
- 特定のスナップショット (8k+1/8k-3回目) でSRHを評価
- 変化の有無とアクセスの有無からリーク判定



提案手法 : Staleness Approach NFGとFGの結果の意味合い

評価前提 :

- $G.\text{num_f} :=$ グループGの解放済みオブジェクト数
- $G.\text{num_nf} :=$ グループGの未解放オブジェクト数 ≥ 1

NFG	FG	T	F
T	$G.\text{num_f} == G.\text{num_nf}$	$G.\text{num_f} < G.\text{num_nf}$	
F	$G.\text{num_f} > G.\text{num_nf}$	$G.\text{num_f} == 0$ $G.\text{num_nf} == 1$	
	解放済/未解放オブジェクトが 同数のグループ	未解放オブジェクトが 多いグループ	単体オブジェクト グループ

実装： 動的バイナリ解析

概説：

- 実行を伴って機械語命令(instruction)を解析する手法

利点：

- リコンパイル不要で解析を実現
 - ソースコード入手が難しいプログラムも解析可能
- コンパイラによる最適化後の動作を追跡可能

欠点：

- 型情報の欠落
 - 付与するにはリコンパイルでデバッグ情報の追加が必要
- ソースコードの情報を推し量ることが困難
 - コメントや最適化前の情報など

```
_mymalloc:  
.cfi_startproc  
pushq %rbp  
Lcfi6:  
.cfi_def_cfa_offset 16  
Lcfi7:  
.cfi_offset %rbp, -16  
movq %rsp, %rbp  
Lcfi8:  
.cfi_def_cfa_register %rbp  
subq $16, %rsp  
movl %edi, -4(%rbp)
```

これはバイナリというかニーモニックだけども

実装： Intel Pin[5]

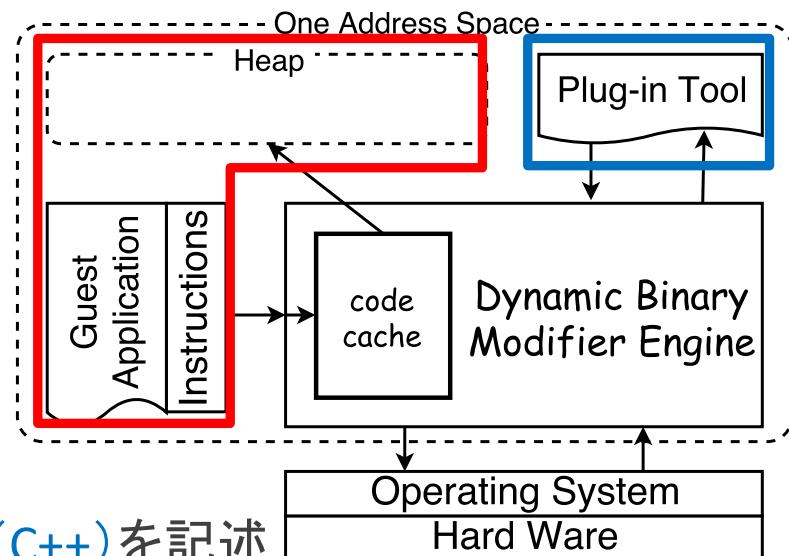
Intelが提供する動的バイナリ解析器(DBM:Dynamic Binary Modifier)

- <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

x86/x64に対応

検査対象と同じアドレス空間に配置

- DBMに共通の特徴
- メモリアドレス内のデータにアクセス可
- →ヒープとか機械語コードとか



ユーザは提供されるAPIなどでツール(C++)を記述

- Pikelet/SWATはこのツール部分として実装/再現
- タスク: CC/オブジェクトメタデータの管理、リーク判定など

実験：比較対象 SWAT[1]の選択理由

PikeletはSWATを基礎に実装

- アクセス監視のサンプリング
 - Stalenessの概念
- 
- SWATから継承

比較対象としてのSWAT

- 上記の基礎部分を継承したので比較がしやすかった
 - オブジェクト個々に対するStaleness解析の限界
 - 即時判定が不可能
 - サンプリングによるアクセスの見逃し→誤検出へ
- 
- グループ化の
意義主張

実験: リークの注入

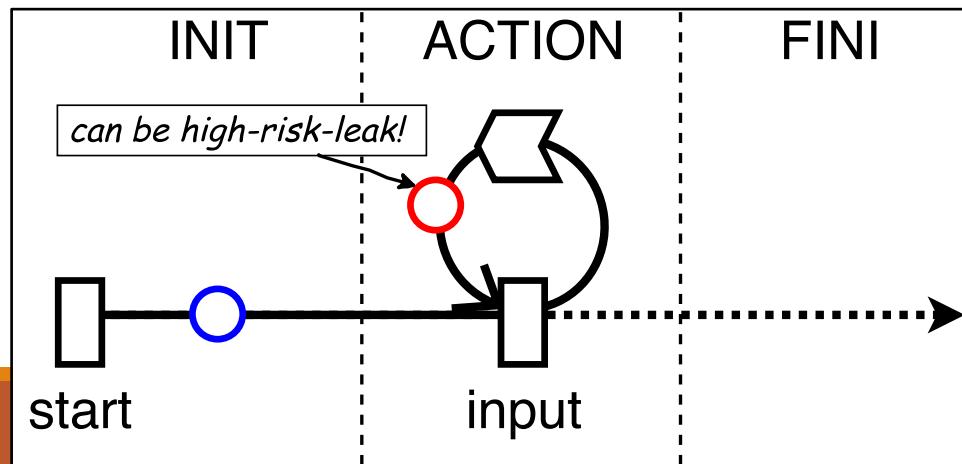
方法: ランダムにfreeを間引く(20%)

- ランダムに間引くことで恣意性を排除

目的: 高脅威リークの注入

Webサーバなどは入力ループを所持

- 入力ループ内のAlloc-Deallocの均衡が崩れる
- → **入力ループ内のグループが一部高脅威化**
- → 入力ループ外のグループについては一部低脅威化
- → 終了時に解放されるグループについては無期限実行の観点から無視



展望： カスタムアロケータへの対応

カスタムアロケータ(my-malloc/my-free) :

- ヒープ領域から大きめのオブジェクト(プール)を確保①し
プログラム自身でプールの管理②を行うテクニック

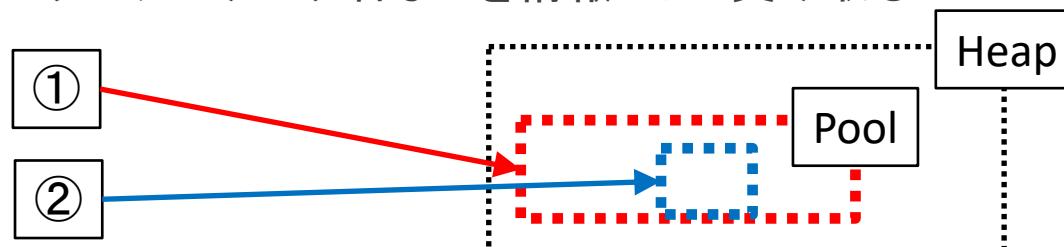
利点: プログラム挙動全体の高速化

欠点: プールの中身は透明性に欠ける

Pikeletは①は検知/リーク判定が可能だが②は未対応

規格が合っていて、かつ外部補助を許すなら実装可能[4]

- 規格例: mallocは領域確保可ならポインタを、不可ならNULLを返す、など
- 外部補助: カスタムアロケータ名などを情報として受け取る



展望： 何を報告するか

メモリリークの原因箇所の特定に役立つ情報

- Allocation Calling Context(グループ情報)
- Deallocation Calling Context
 - グループ内のオブジェクトに対する解放時のCCを報告(未実装)
- オブジェクトを参照していたポインタ(未実装)
- Allocation前のコールスタックの挙動(半分実装)

メモリの使用量の推移

などなど思案中