



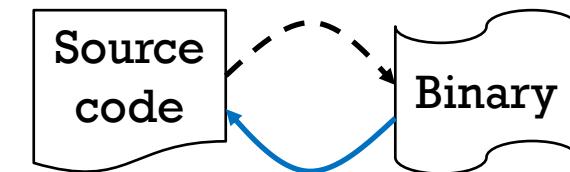
# **HOWARD: A DYNAMIC EXCAVATOR FOR REVERSE ENGINEERING DATA STRUCTURES (SLOWINSKA NDSS'11)**



# -1: 予備知識

---

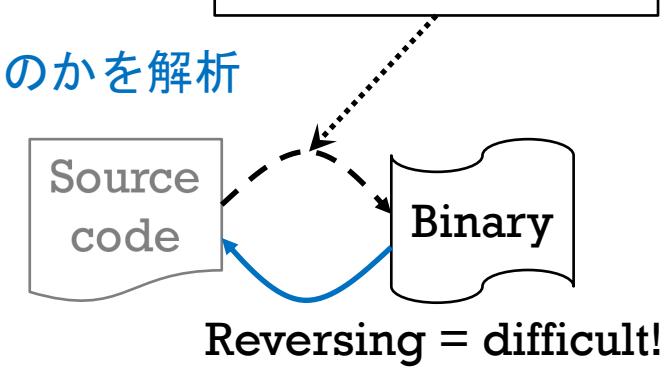
- (ここでいう)データ構造解析:
  - 対象バイナリにどのようなデータ構造があるのかを解析
  - バイナリからソースコードを類推する一環
    - →リバースエンジニアリングの一部
    - 
    - 
    - 
    - 
    - 
    - 
    - 
    - 
    - 
    - 
    -



# -1: 予備知識

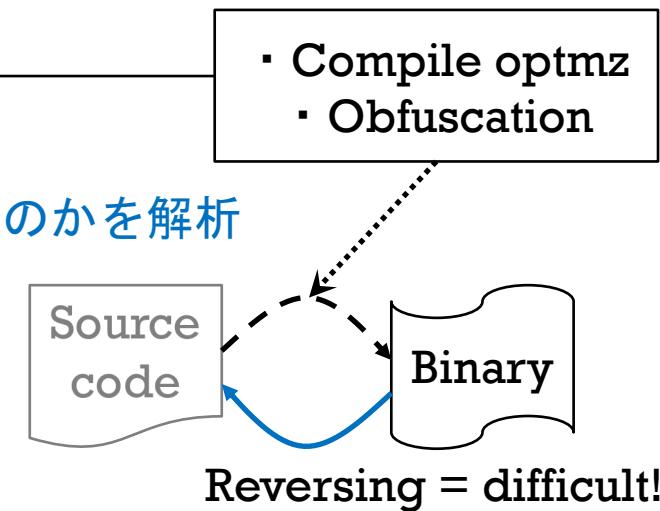
- (ここでいう)データ構造解析:
  - 対象バイナリにどのようなデータ構造があるのかを解析
  - バイナリからソースコードを類推する一環
    - →リバースエンジニアリングの一部
    - ソースコードがない前提
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  -

- Compile optmz
- Obfuscation



# -1: 予備知識

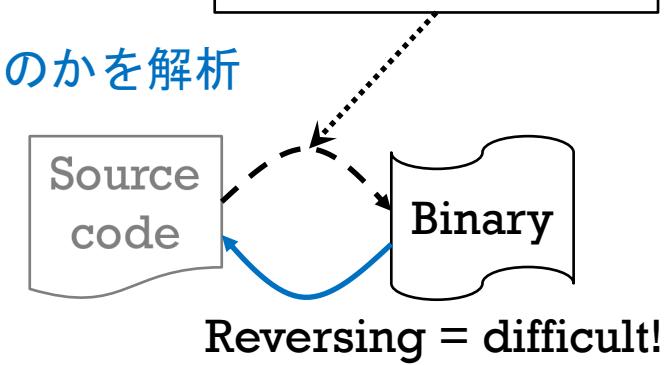
- (ここでいう)データ構造解析:
  - 対象バイナリにどのようなデータ構造があるのかを解析
  - バイナリからソースコードを類推する一環
    - →リバースエンジニアリングの一部
    - ソースコードがない前提
  - データ構造:
    - 変数とその型
    - 構造体とそのフィールド
    - 配列とその要素数
    - Etc.etc...
  - 解析対象:
    - 静的領域(グローバル変数, 静的変数)
    - 動的領域(ヒープオブジェクト)
    - スタックフレーム(ローカル変数, 引数)



# -1: 予備知識

- (ここでいう)データ構造解析:
  - 対象バイナリにどのようなデータ構造があるのかを解析
  - バイナリからソースコードを類推する一環
    - →リバースエンジニアリングの一部
    - ソースコードがない前提
  - データ構造:
    - 変数とその型
    - 構造体とそのフィールド
    - 配列とその要素数
    - Etc.etc...
  - 解析対象:
    - 静的領域(グローバル変数, 静的変数)
    - 動的領域(ヒープオブジェクト)
    - スタックフレーム(ローカル変数, 引数)

- Compile optmz
- Obfuscation



データ構造には  
色々なパターンが存在

- 構造体配列
- 配列フィールド
- 入れ子構造体
- コンテナ
- 多次元配列
- ...

解析対象により検出方法も変わる

# 0: 概要

---

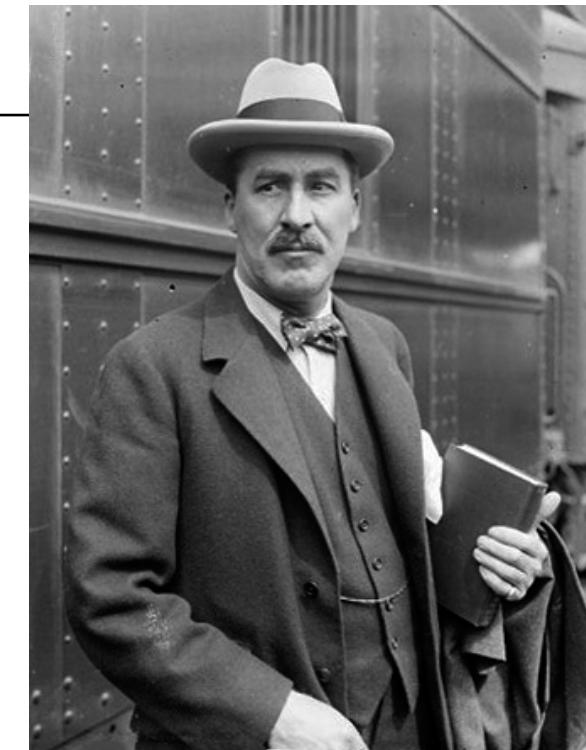
- 手法名:
  - **Howard**
- 提案手法目的:
  - アクセスパターンから動的にデータ構造(変数/配列/構造体)を識別
  - シンボルテーブルを再構築
- 想定言語/環境:
  - **C binary without source code & symbol table**
- 貢献
  - Cバイナリからデータ構造を抽出できることを実証
    - かつカバレッジを確保した
  - この技術の応用先を提案
    - マルウェア解析(reverse engineering)
    - レガシーコードの保護(protection from buffer-over-run)

セキュリティ

# 1: 関連研究

## 既存研究の問題点

- 商用: (IDA Pro[24], OllyDbg[2])
  - (リバースエンジニアリングの観点から)  
そもそもデータ構造情報が欠落
- 静的データ構造解析:
  - 配列を適切に判定できない
- (既存の)動的データ構造解析:
  - 実行される一部しか判定できない
- **Howard**
  - データ構造を検出
    - 配列も検出
    - 構造体フィールドも検出
  - KLEE[13]と組み合わせて実行力バレッジも確保



提案手法名元ネタの  
Howard Carter  
(1874-1939)

ツタンカーメンの墓発掘

[24] <http://www.hex-rays.com/idapro/> datastruct/datastruct.pdf, 2005

[2] <http://www.ollydbg.de/>

[13] KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.

# 1: 関連研究(星取表)

	静 動	配列	構造体/ フィールド	変数	領域 (静/動/ス)	カバレッジ
VSA[8]	静	×	▲/▲	○	○/×/▲	○
ASI[38]	静	▲	○/○	▲	○/×/▲	○
Laika[22]	動	○	○/×	○	○/○/○	×
Rewards[31]	動	○	○/○	○	○/○/○	×
Howard with KLEE*	動	○	○/○	○	○/○/○	○*
俺method	動	○	○/?	○	?/○/?	?

[8] Analyzing memory accesses in x86 binary executables

[22] Digging for data structures

[31] Automatic reverse engineering of data structures from binary execution

[38] Aggregate structure identification and its application to program analysis

# 1: 関連研究(星取表)

	静 動	配列	構造体/ フィールド	変数	領域 (静/動/ス)	カバレッジ
VSA[8]	静	×	▲/▲	○	○/×/▲	○
ASI[38]	静	▲	○/○	▲	○/×/▲	○
Laika[22]	動	○	○/×	○	○/○/○	×
Rewards[31]	動	○	○/○	○	○/○/○	×
Howard with KLEE*	動	○	○/○	○	○/○/○	○*
俺method	動	○	○/?	○	?/○/?	?

[8] Analyzing memory accesses in x86 binary executables

[22] Digging for data structures

[31] Automatic reverse engineering of data structures from binary execution

[38] Aggregate structure identification and its application to program analysis

# 1: 関連研究(星取表)

	静 動	配列	構造体/ フィールド	変数	領域 (静/動/ス)	カバレッジ
VSA[8]	静	×	▲/▲	○	○/×/▲	○
ASI[38]	静	▲	○/○	▲	○/×/▲	○
Laika[22]	動	○	○/×	○	○/○/○	×
Rewards[31]	動	○	○/○	○	○/○/○	×
Howard with KLEE*	動	○	○/○	○	○/○/○	○*
俺method	動	○	○/?	○	?/○/?	?

[8] Analyzing memory accesses in x86 binary executables

[22] Digging for data structures

[31] Automatic reverse engineering of data structures from binary execution

[38] Aggregate structure identification and its application to program analysis

## 2: 問題設定

---

- Howardがデータ構造を解析するために必要なテクニック
  - Motivationそのものは
    - 既存技術にできなかったCバイナリでのデータ構造解析 (with coverage)
- Memory Allocation Context (コンテキストの識別)
- Pointer Identification (ポインタの識別)
  - Missing Base Pointers (ベースポインタの伝播)
  - Multiple Base Pointers (ベースポインタの選択)
- Array Detection (配列の検知)
  - Loop Detection (ループの検知)

## 3: 提案手法

---

- Howardがデータ構造を解析するために必要なテクニック
  - Motivationそのものは
    - 既存技術にできなかったCバイナリでのデータ構造解析 (with coverage)
- Memory Allocation Context (コンテキストの識別)
  - →3.1
- Pointer Identification (ポインタの識別)
  - Missing Base Pointers (ベースポインタの伝播)
  - Multiple Base Pointers (ベースポインタの選択)
  - →3.2
- Array Detection (配列の検知)
  - Loop Detection (ループの検知)
  - →3.3

## 3.1: 提案手法(HSTACK)

---

- Function Call Stack : HStack

- Howard内で再現構成するコールスタック
  - call/retを元に構築

- 用途① : ヒープオブジェクトの分類 by Context ( $\Leftarrow$  Calling Context)

- For Heap

- 用途② : 関数の同定 (関数フレームの記録/再利用)

- For Stack

- allocaは一旦スルー

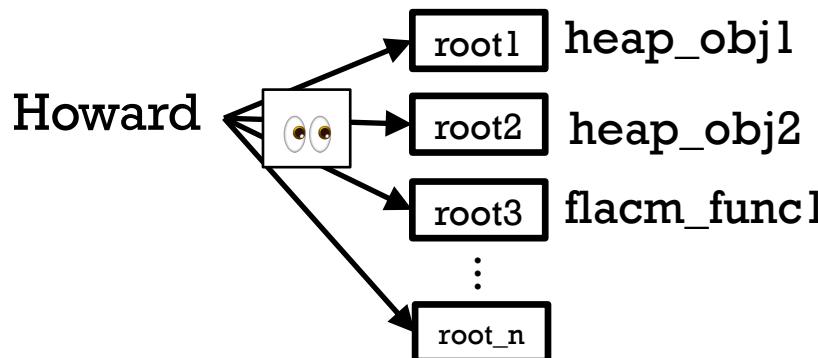
- 用途③ : ループ情報の記録

- For Array

# 3.2: 提案手法(POINTER TRACKING) OVERVIEW

---

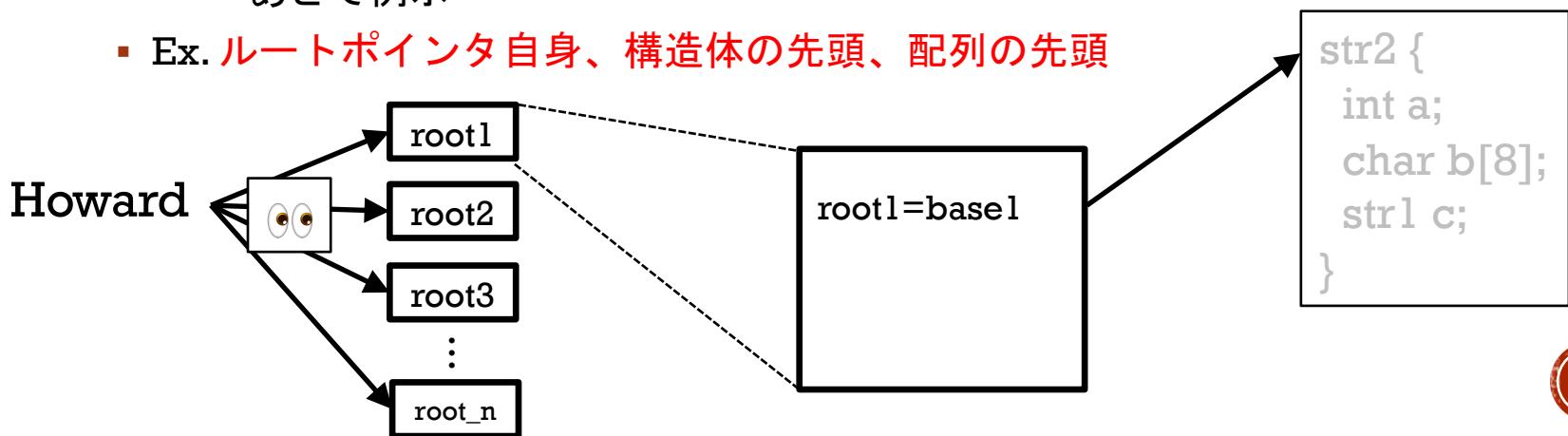
- Pointer Tracking
- 用語: ルートポインタ
  - 他のポインタから導出できないポインタ(アドレス)
    - i.e. ルートポインタが異なる  $\Leftrightarrow$  監視対象が異なる
    - Ex. 静的/グローバル変数、ヒープオブジェクト、各関数フレーム
- 用語: ベースポインタ
  - 共通のルートポインタ内の区割りを示すポインタ(アドレス)
    - i.e. ベースポインタが異なる  $\Leftrightarrow$  区割りにおける帰属が異なる
      - →あとで例示
  - Ex. ルートポインタ自身、構造体の先頭、配列の先頭



# 3.2: 提案手法(POINTER TRACKING) OVERVIEW

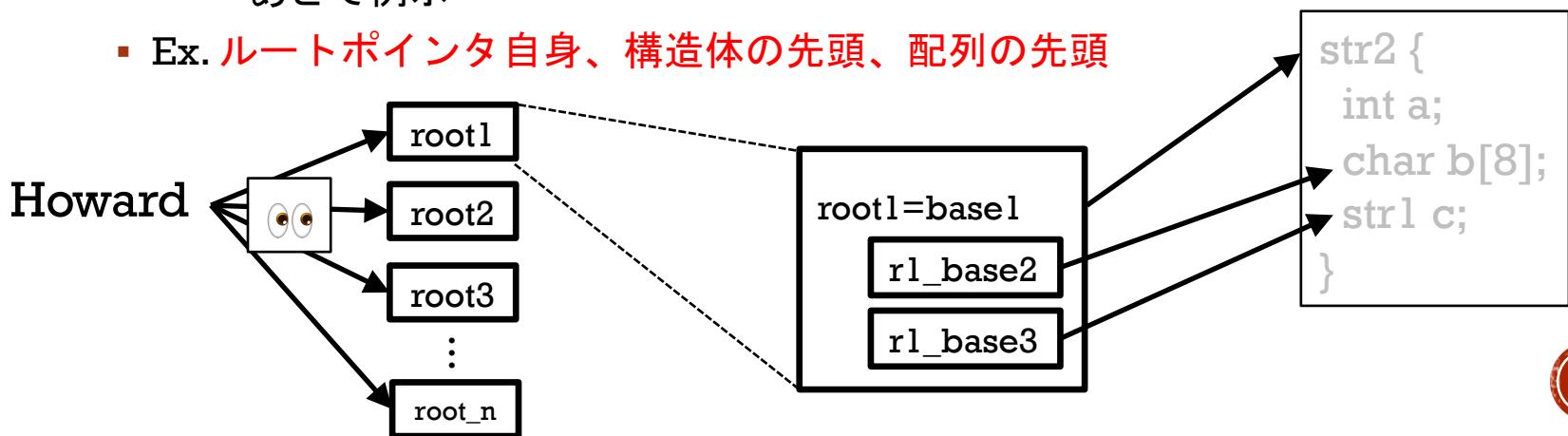
---

- Pointer Tracking
- 用語: ルートポインタ
  - 他のポインタから導出できないポインタ(アドレス)
    - i.e. ルートポインタが異なる  $\Leftrightarrow$  監視対象が異なる
    - Ex. 静的/グローバル変数、ヒープオブジェクト、各関数フレーム
- 用語: ベースポインタ
  - 共通のルートポインタ内の区割りを示すポインタ(アドレス)
    - i.e. ベースポインタが異なる  $\Leftrightarrow$  区割りにおける帰属が異なる
      - →あとで例示
  - Ex. ルートポインタ自身、構造体の先頭、配列の先頭



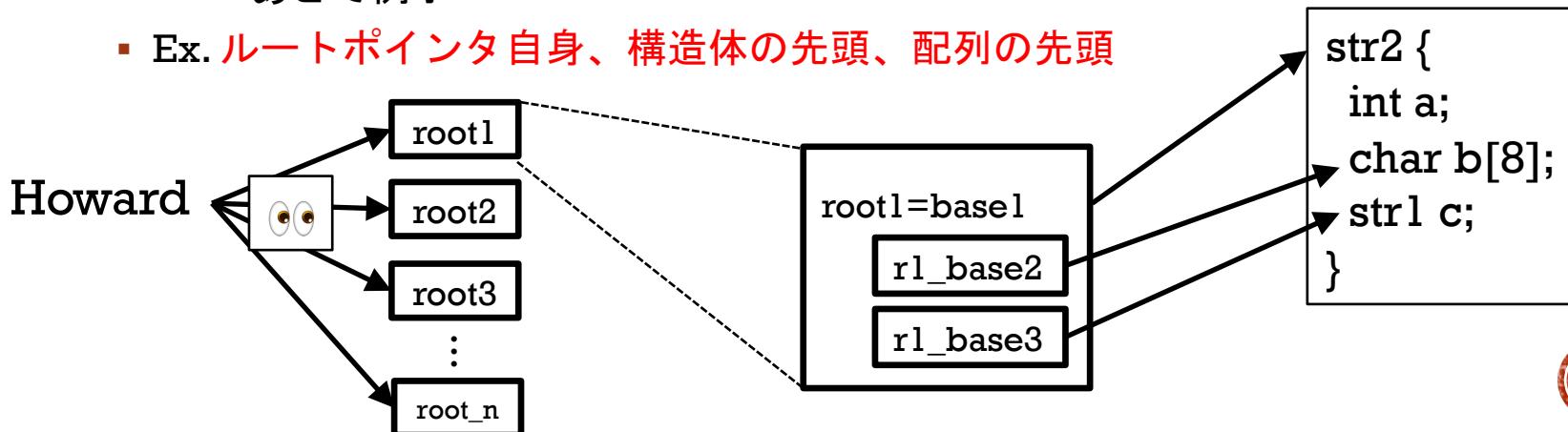
## 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- Pointer Tracking
- 用語: ルートポインタ
  - 他のポインタから導出できないポインタ(アドレス)
    - i.e. ルートポインタが異なる  $\Leftrightarrow$  監視対象が異なる
    - Ex. 静的/グローバル変数、ヒープオブジェクト、各関数フレーム
- 用語: ベースポインタ
  - 共通のルートポインタ内の区割りを示すポインタ(アドレス)
    - i.e. ベースポインタが異なる  $\Leftrightarrow$  区割りにおける帰属が異なる
      - →あとで例示
  - Ex. ルートポインタ自身、構造体の先頭、配列の先頭



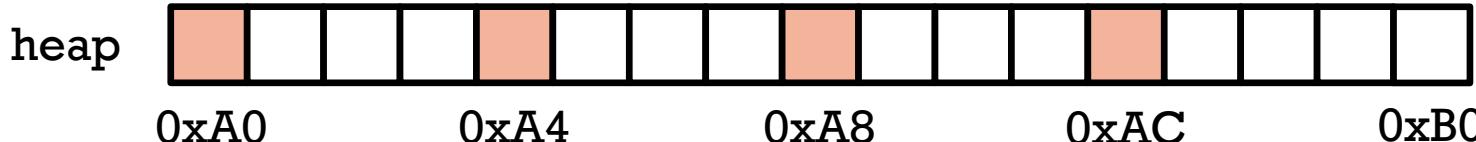
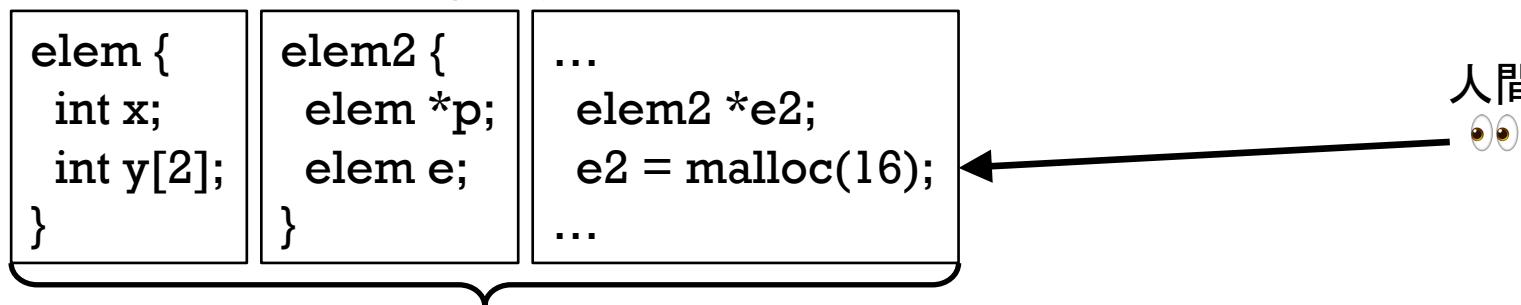
# 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- Pointer Tracking
- 用語: ルートポインタ
  - 他のポインタから導出できないポインタ(アドレス)
    - i.e. ルートポインタが異なる  $\Leftrightarrow$  監視対象が異なる
    - Ex. 静的/グローバル変数、ヒープオブジェクト、各関数フレーム
- 用語: ベースポインタ
  - 共通のルートポインタ内の区割りを示すポインタ(アドレス)
    - i.e. ベースポインタが異なる  $\Leftrightarrow$  区割りにおける帰属が異なる
      - →あとで例示
  - Ex. ルートポインタ自身、構造体の先頭、配列の先頭



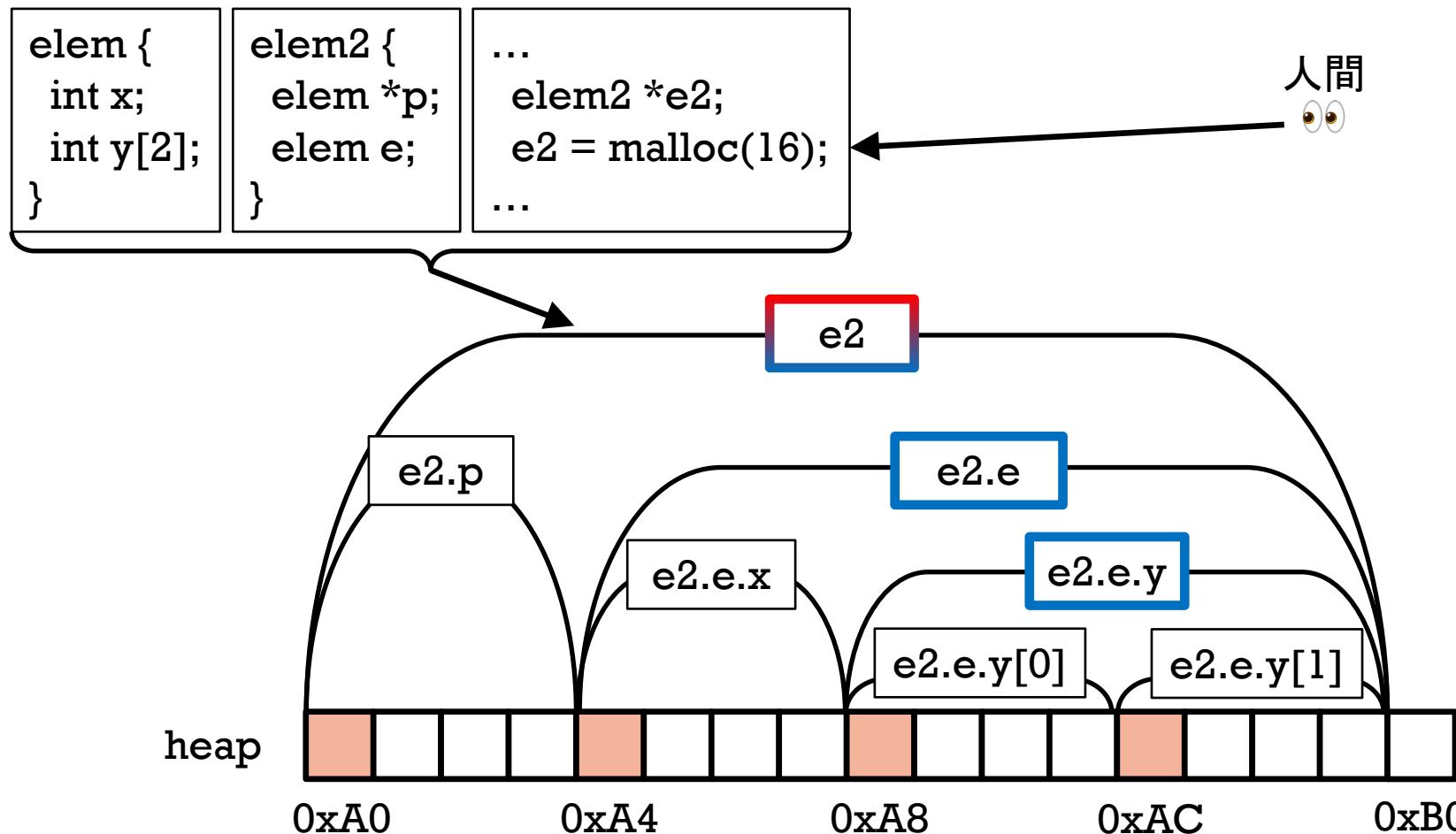
## 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



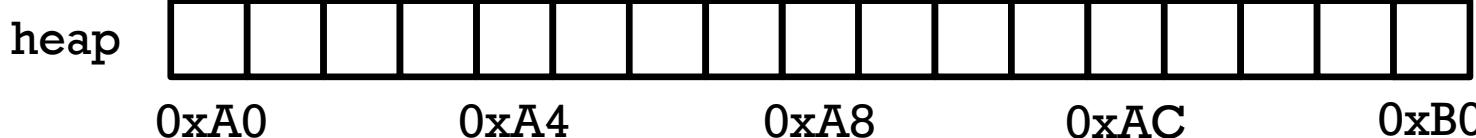
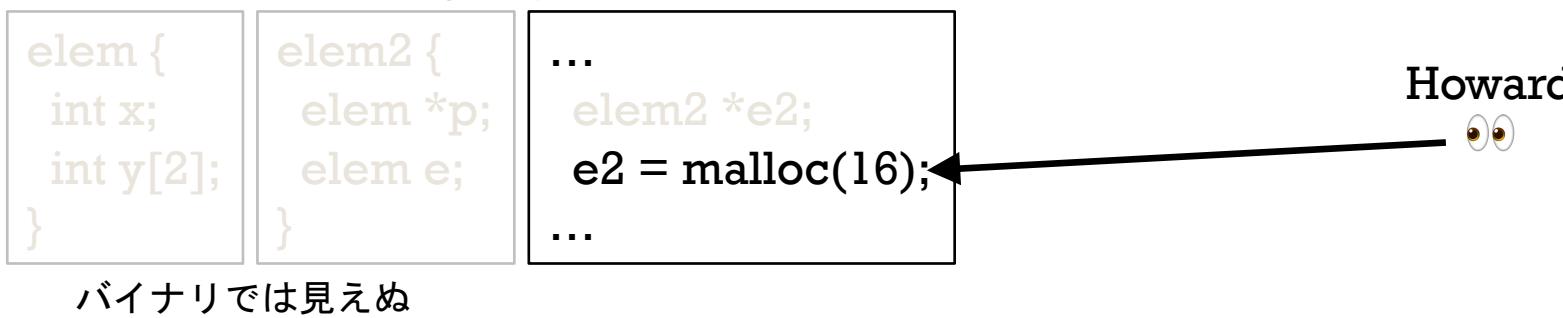
## 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



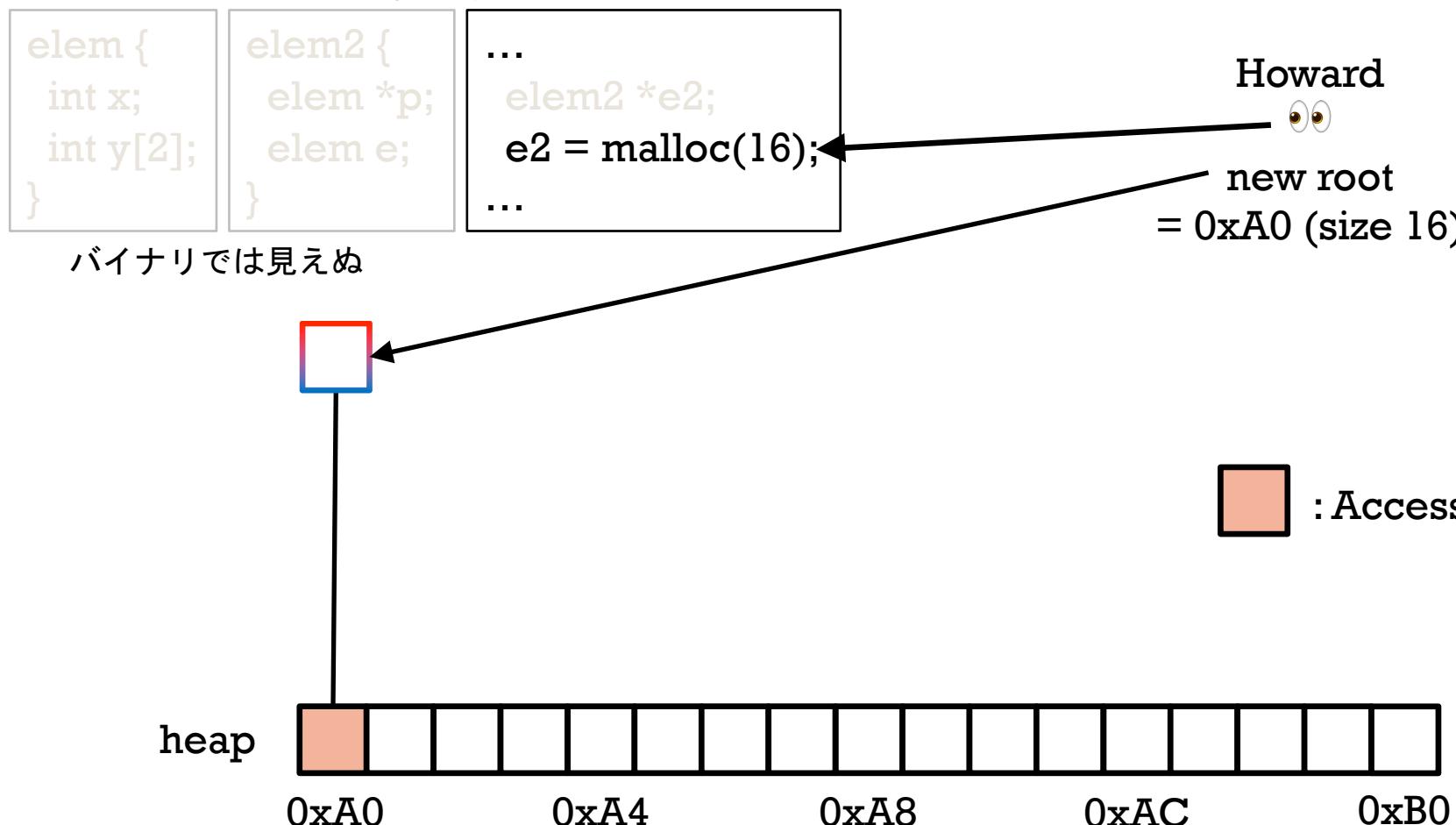
## 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



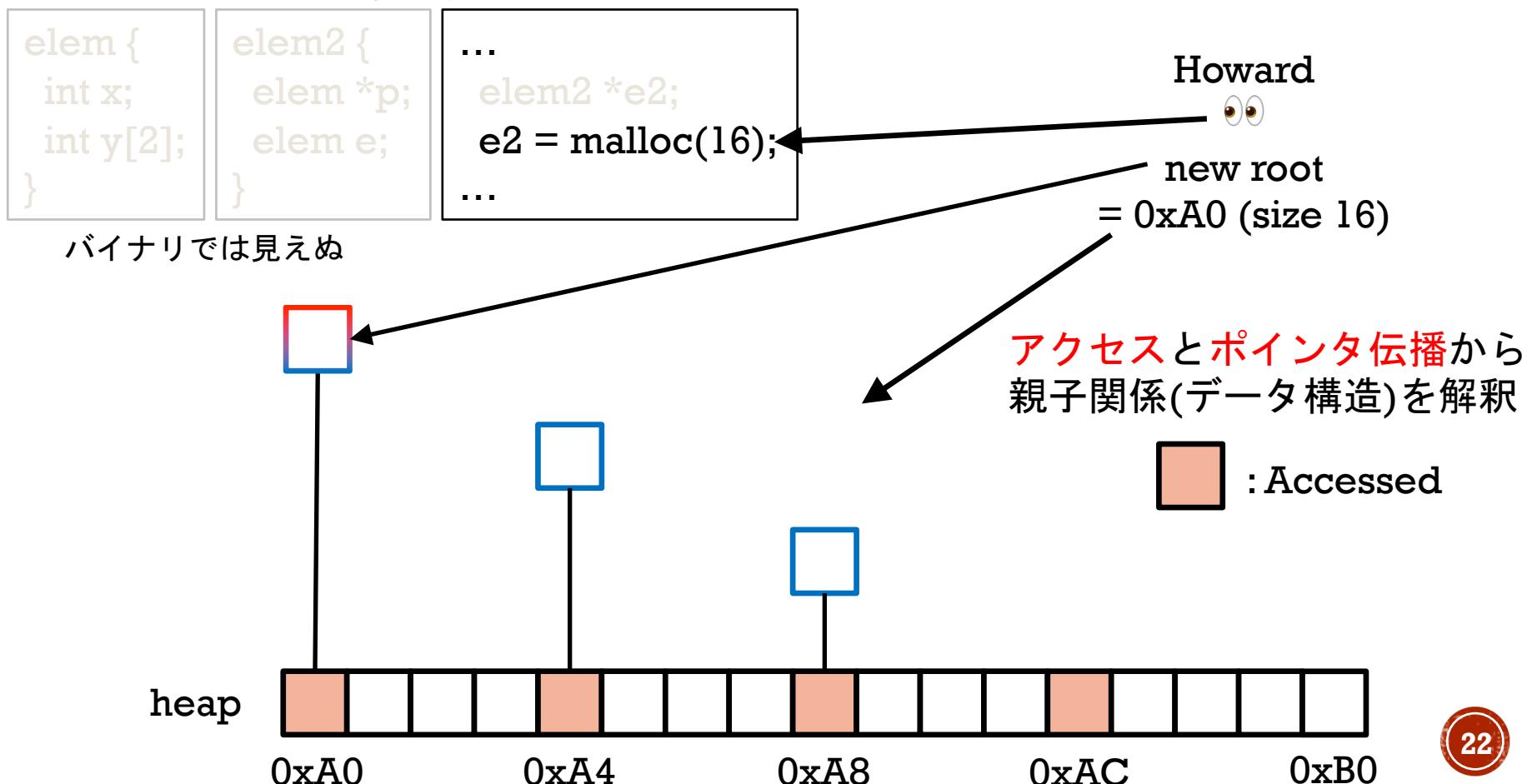
## 3.2: 提案手法(POINTER TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



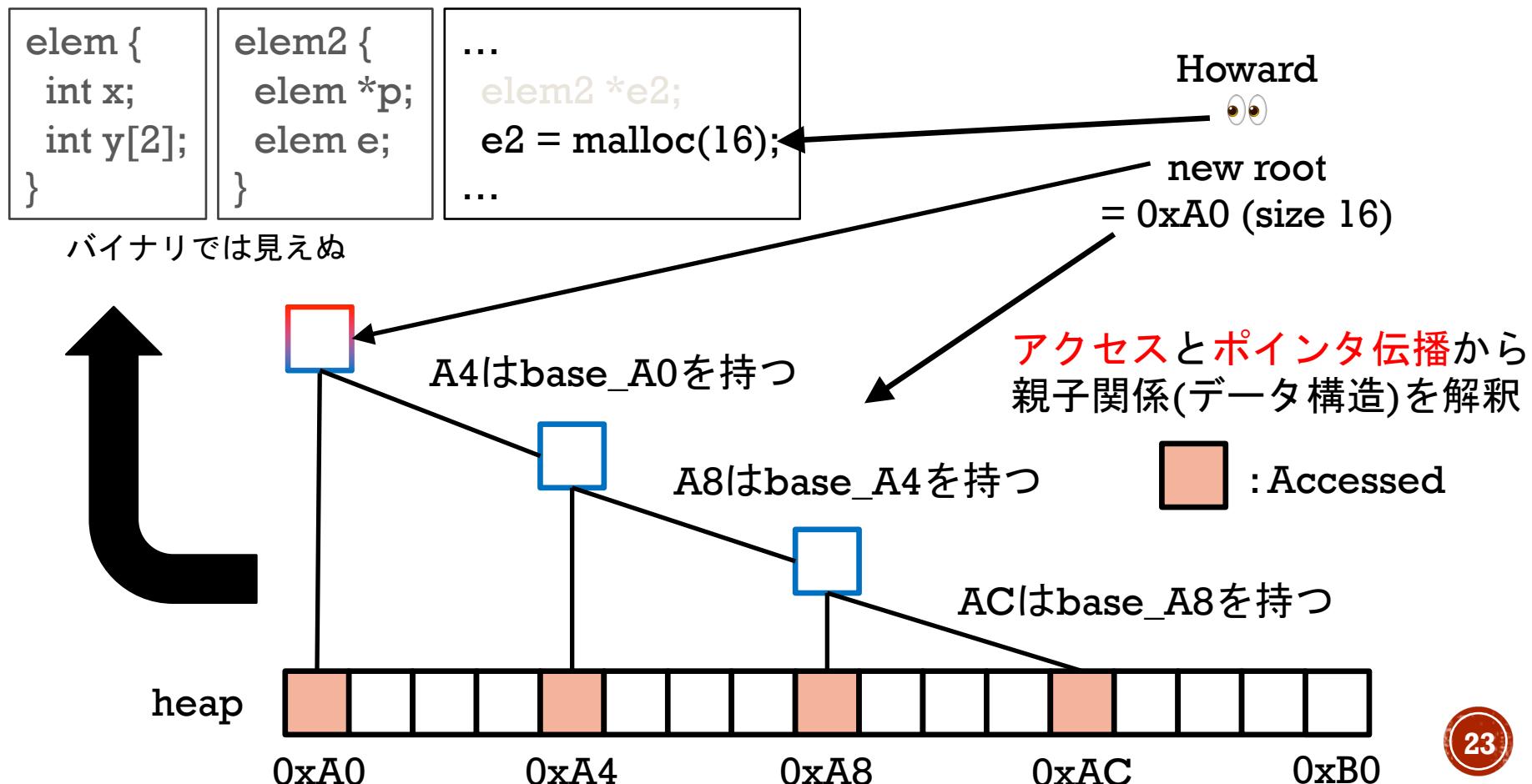
## 3.2: 提案手法(PARTICLE TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



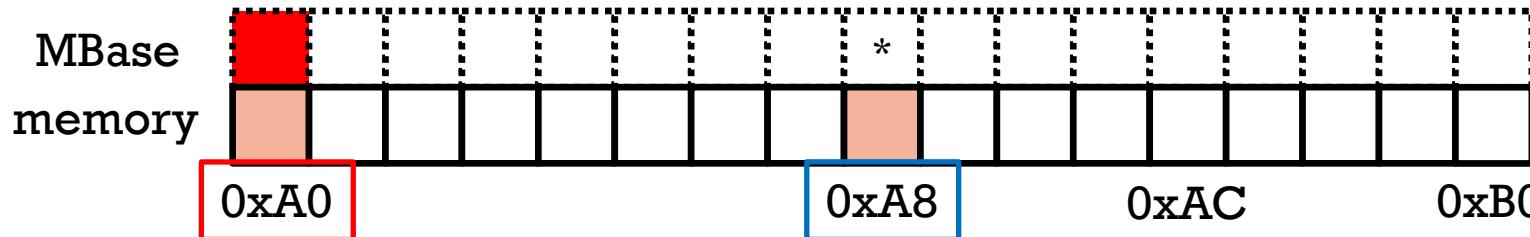
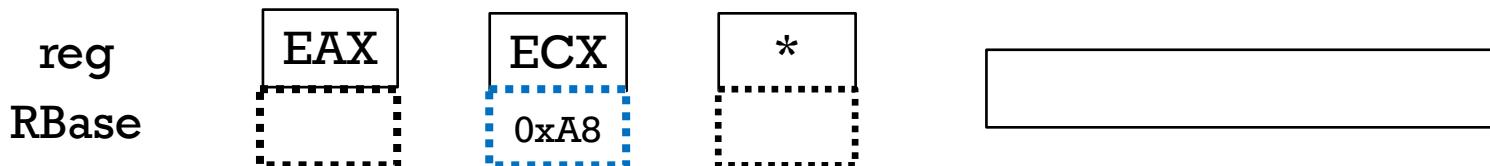
## 3.2: 提案手法(PARTICLE TRACKING) OVERVIEW

- ベースポインタ間には木構造のような親子関係が存在
  - 全ノードは親を辿ればルートポインタへ収束



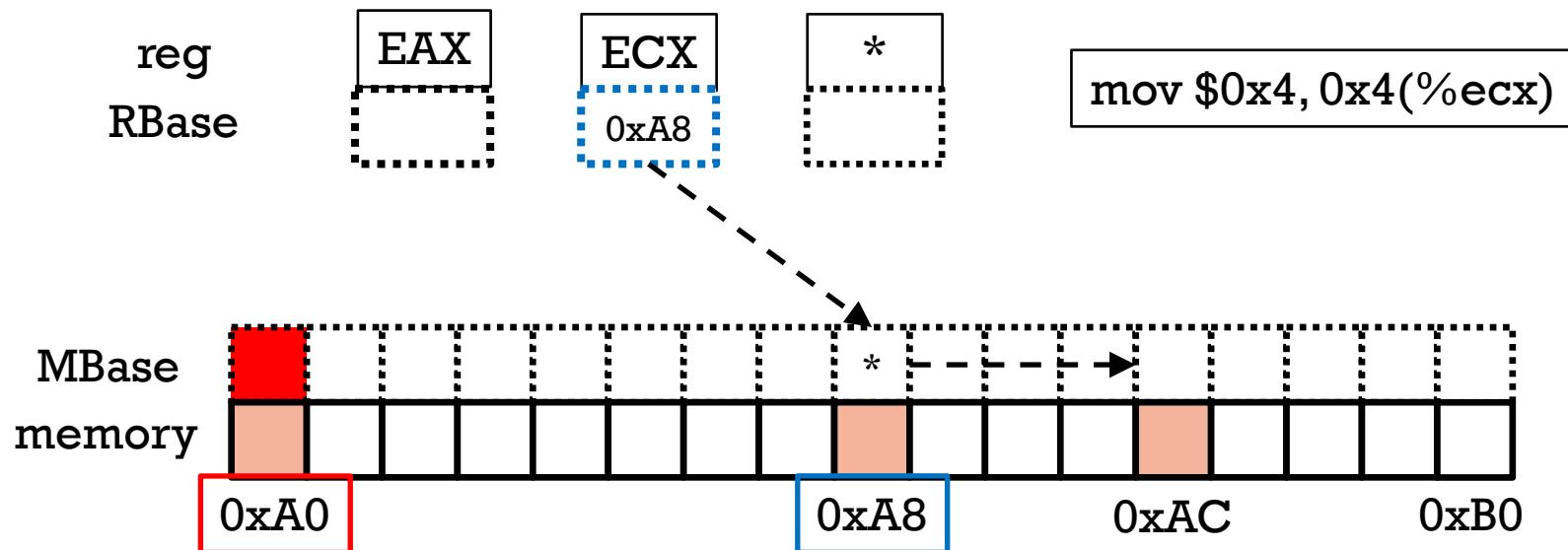
## 3.2: 提案手法(POINTER TRACKING) PROPAGATION

- 肝: メモリ参照前にはレジスタにアドレス値が格納される
  - →レジスタを介して伝播してもらう
- 用語: **MBase(addr)**: ルートポインタ以外初期値は0
  - 監視対象のメモリ領域に1byte単位で付けるタグ(ベースポインタ)
    - →各メモリのベースポインタはどこか
- 用語: **RBase(reg)**
  - 各レジスタに伝播してもらうタグ(ベースポインタ)



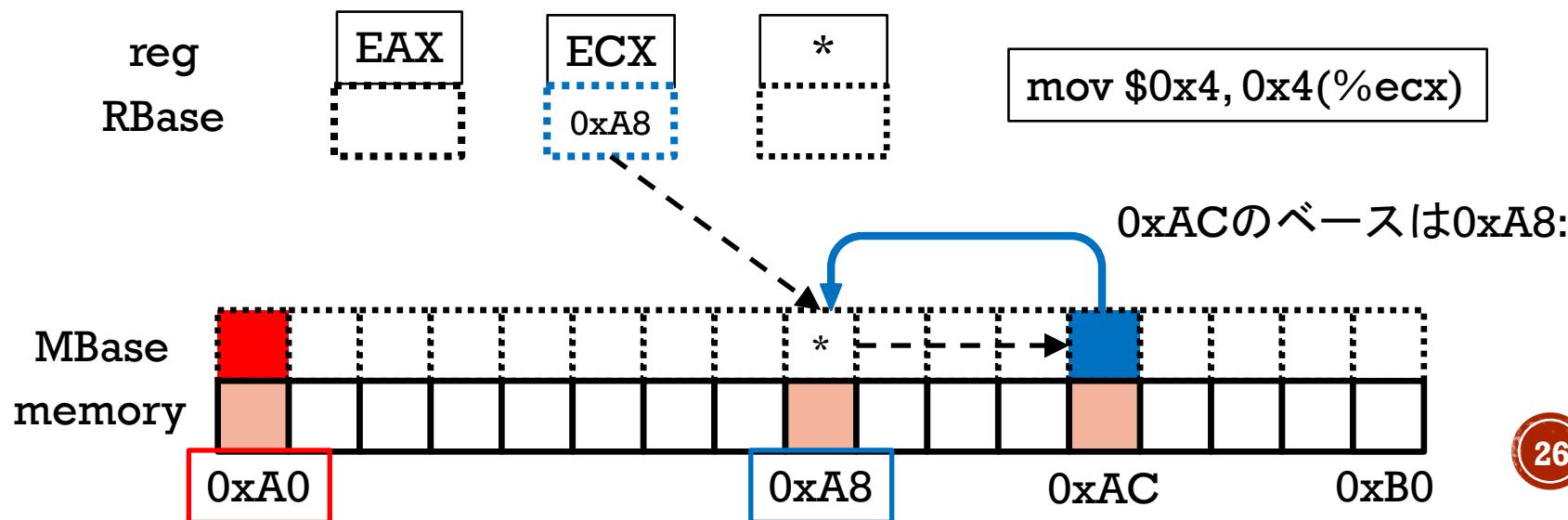
## 3.2: 提案手法(PINTER TRACKING) PROPAGATION

- 肝: メモリ参照前にはレジスタにアドレス値が格納される
  - レジスタを介して伝播してもらう
- 用語: **MBase(addr)**: ルートポインタ以外初期値は0
  - 監視対象のメモリ領域に1byte単位で付けるタグ(ベースポインタ)
    - 各メモリのベースポインタはどこか
- 用語: **RBase(reg)**
  - 各レジスタに伝播してもらうタグ(ベースポインタ)



## 3.2: 提案手法(PINTER TRACKING) PROPAGATION

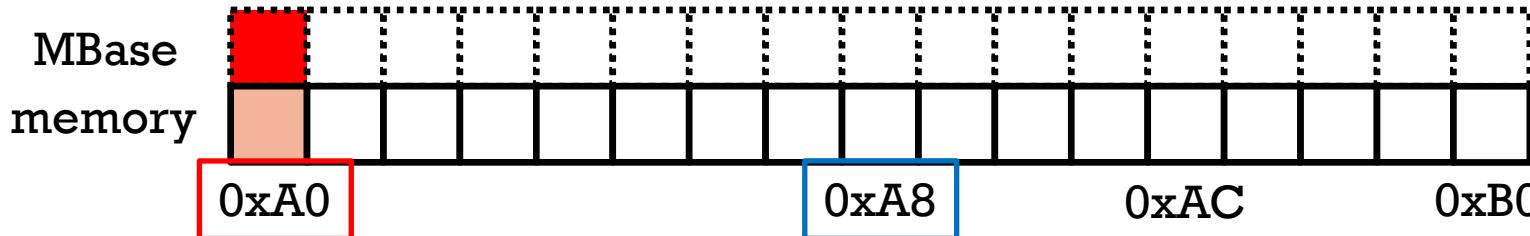
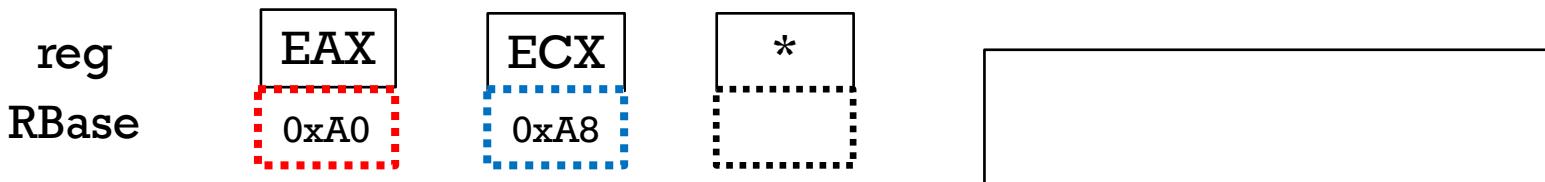
- 肝: メモリ参照前にはレジスタにアドレス値が格納される
  - レジスタを介して伝播してもらう
- 用語: **MBase(addr)**: ルートポインタ以外初期値は0
  - 監視対象のメモリ領域に1byte単位で付けるタグ(ベースポインタ)
    - 各メモリのベースポインタはどこか
- 用語: **RBase(reg)**
  - 各レジスタに伝播してもらうタグ(ベースポインタ)



## 3.2: 提案手法(POINTER TRACKING) PROPAGATION

- 基本ルール:

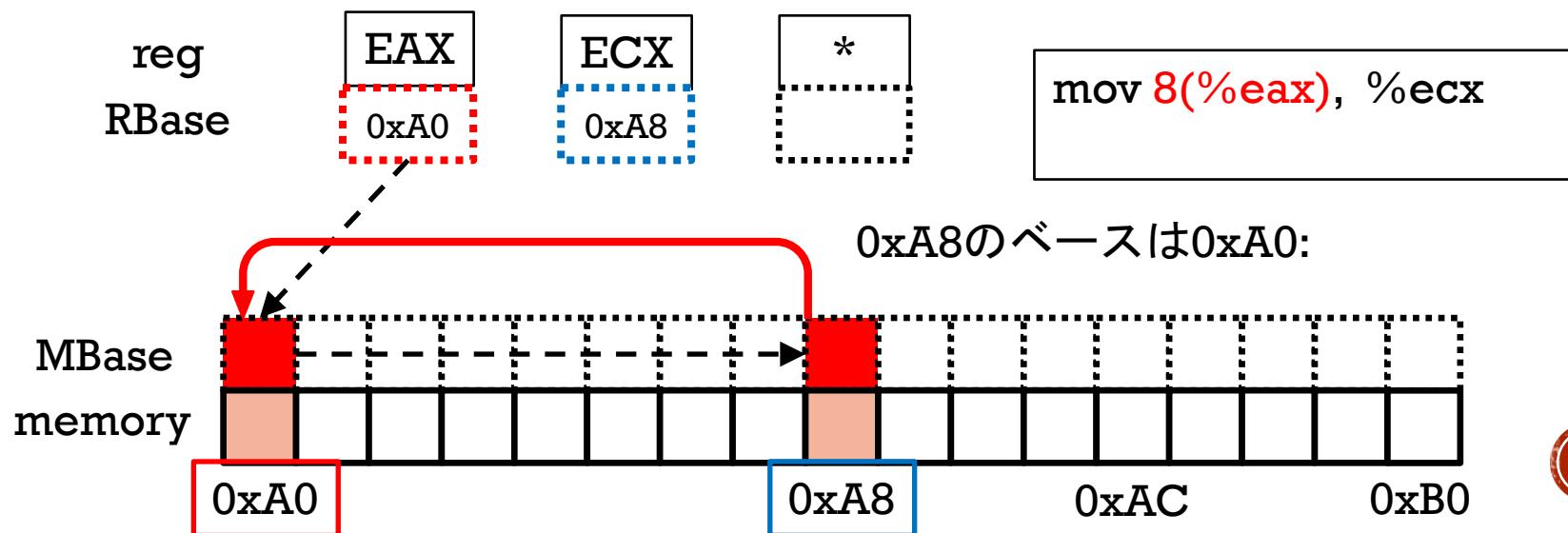
- RBaseの更新は、レジスタにアドレス値がロードされたときのみ
  - $RBase(\text{reg}) := \text{Addr}$  (iff reg loaded Addr)、（ポインタ演算でも発生）
  - スカラがストアされると  $RBase(\text{reg}) = 0$  (NULL)
  - 参照ではRBaseは更新されない
- MBaseの更新の可能性は、参照時
  - $MBase(\text{Baddr}) := RBase(\text{reg}) = \text{Addr}$  (if dereference via reg)
  - ただし、複数のベースポインタから選択をする場合も (3.3)



## 3.2: 提案手法(POINTER TRACKING) PROPAGATION

- 基本ルール:

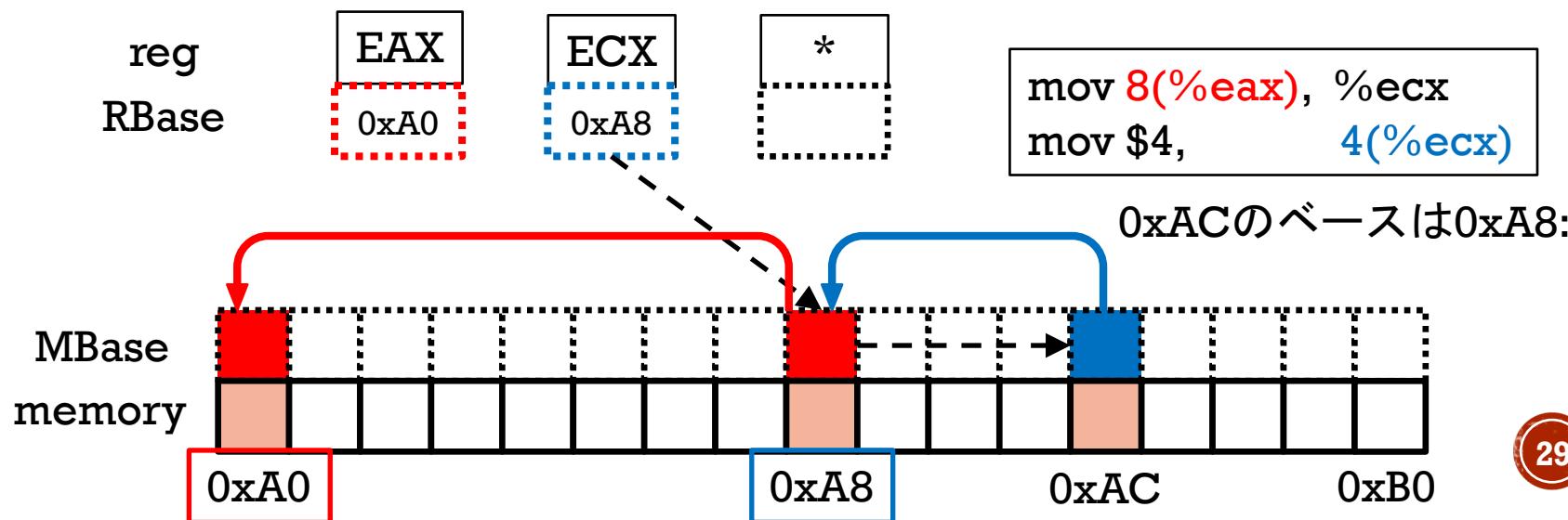
- RBaseの更新は、レジスタにアドレス値がロードされたときのみ
  - $RBase(\text{reg}) := \text{Addr}$  (iff reg loaded Addr)、（ポインタ演算でも発生）
  - スカラがストアされると  $RBase(\text{reg}) = 0$  (NULL)
  - 参照ではRBaseは更新されない
- MBaseの更新の可能性は、参照時
  - $MBase(Baddr) := RBase(\text{reg}) = \text{Addr}$  (if dereference via reg)
  - ただし、複数のベースポインタから選択をする場合も (3.3)



## 3.2: 提案手法(POINTER TRACKING) PROPAGATION

- 基本ルール:

- RBaseの更新は、レジスタにアドレス値がロードされたときのみ
  - $RBase(\text{reg}) := \text{Addr}$  (iff reg loaded Addr)、（ポインタ演算でも発生）
  - スカラがストアされると  $RBase(\text{reg}) = 0$  (NULL)
  - 参照ではRBaseは更新されない
- MBaseの更新の可能性は、参照時
  - $MBase(Baddr) := RBase(\text{reg}) = \text{Addr}$  (if dereference via reg)
  - ただし、複数のベースポインタから選択をする場合も (3.3)



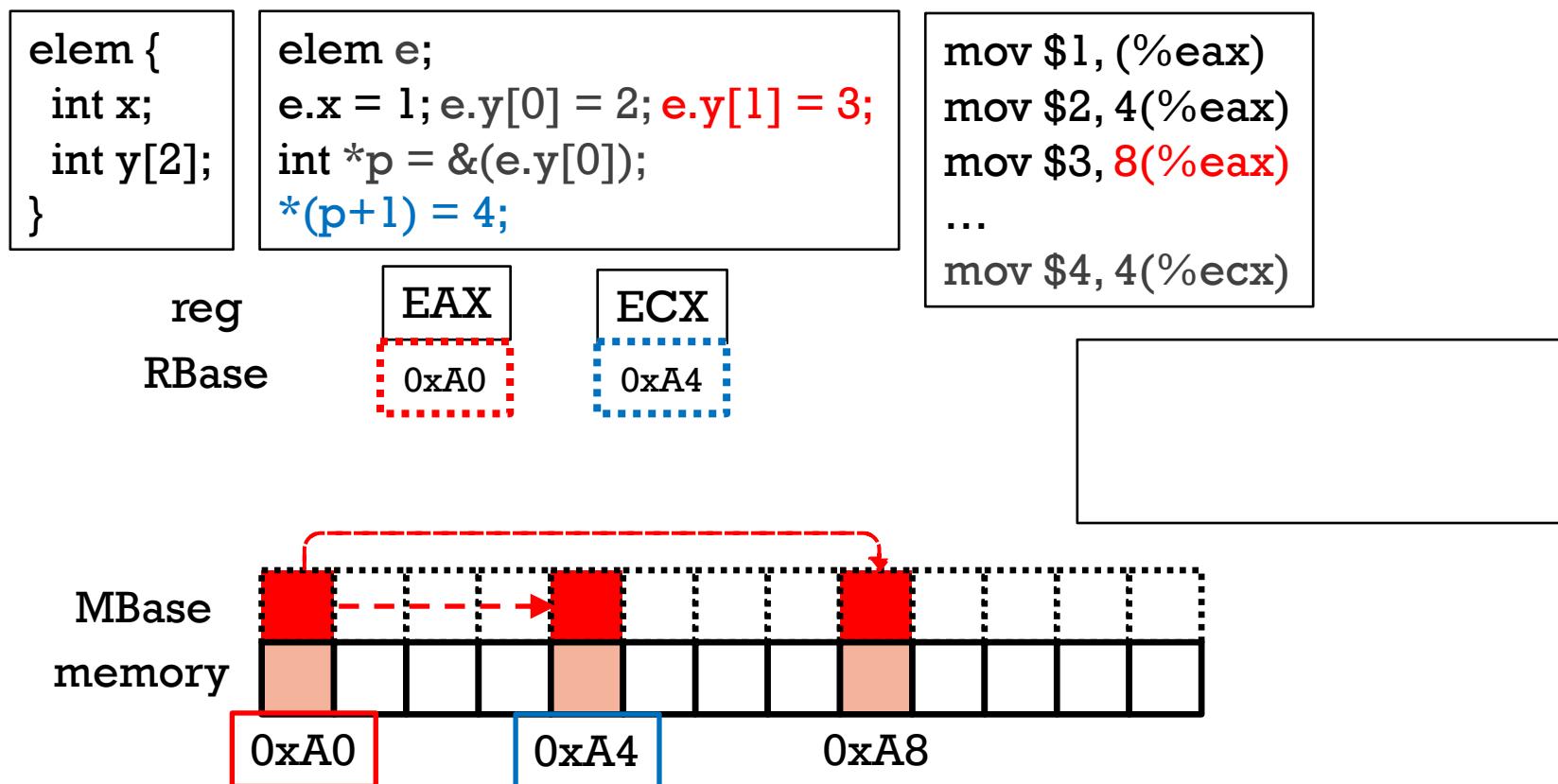
## 3.2: 補足:提案手法(POINTER TRACKING) ROOT POINTERの特定

- 動的:
  - 特定のシステムコール/ライブラリ関数をフック
    - システムコール: mmap, mmap2
    - ライブラリ: mallocなど
    - 先頭アドレスとサイズを取得
- 静的:
  - (あれば)ELF-header, relocation-tableを参照, データセクションを特定
  - あるいはスタックに属していないアドレスの発見
    - 一旦ルートと仮定して、より適切なアドレスがあれば調整を加える
- スタック:
  - 最初のローカル変数を指すアドレスを探す
    - 通常はesp-8になるが、ずれることもある
      - 可能性がある位置をすべて考慮、最終的にマージ

## 3.2: 提案手法(POINTER TRACKING)

### MULTI BP

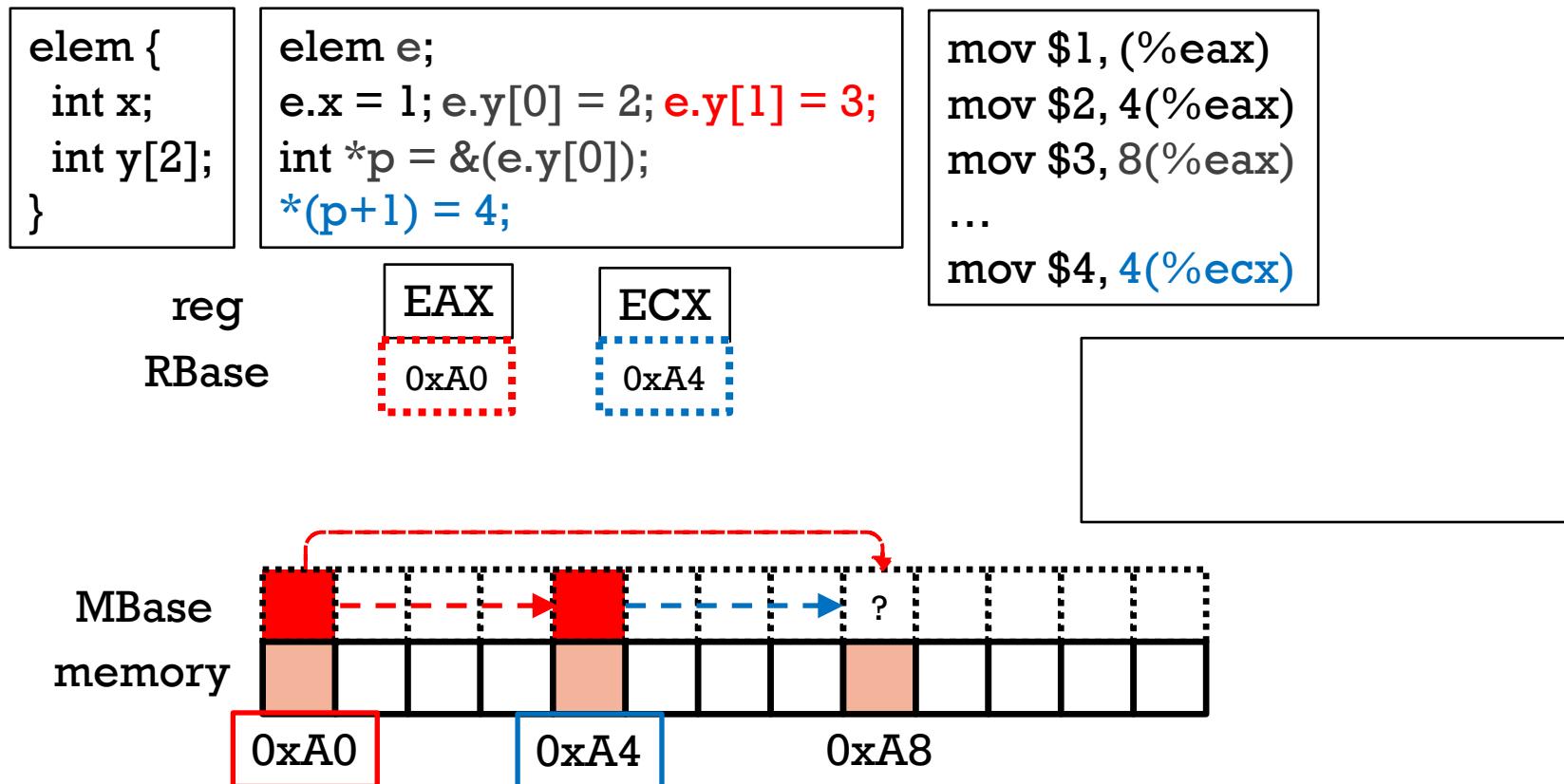
- Multiple Base Pointers
- 各メモリは複数のベースポインタから参照される可能性有り
  - Q. MBaseにはアドレス一つしか格納できませんがどうしましょう



## 3.2: 提案手法(POINTER TRACKING)

### MULTI BP

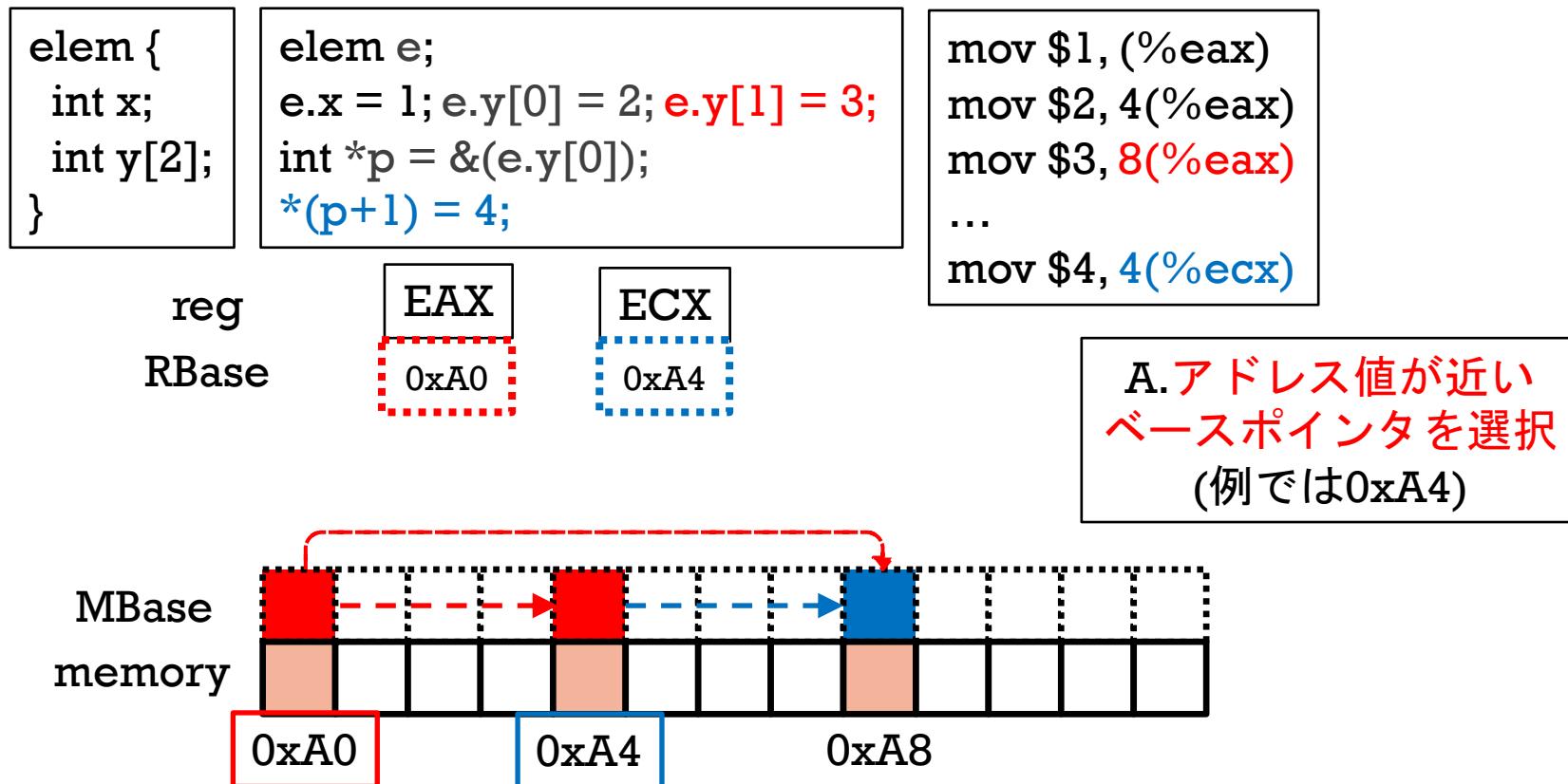
- Multiple Base Pointers
- 各メモリは複数のベースポインタから参照される可能性有り
  - Q. MBaseにはアドレス一つしか格納できませんがどうしましょう



## 3.2: 提案手法(POINTER TRACKING)

### MULTI BP

- Multiple Base Pointers
- 各メモリは複数のベースポインタから参照される可能性有り
  - Q. MBaseにはアドレス一つしか格納できませんがどうしましょう



## 3.2: 提案手法(POINTER TRACKING)

### MULTI BP

---

- Multiple Base Pointers
- 各メモリは複数のベースポインタから参照される可能性有り
  - Q. MBaseにはアドレス一つしか格納できませんがどうしましょう
  - A. アドレス値が近い方を選択します
- 近いアドレスを選択すると、ルートまでに辿るノードが増える
  - →より詳細な構造となる
    - 特に入れ子構造体に有効
  - 経験則的にもまあ妥当

## 3.2: 補足: 提案手法(POINTER TRACKING) BASE'S BASE

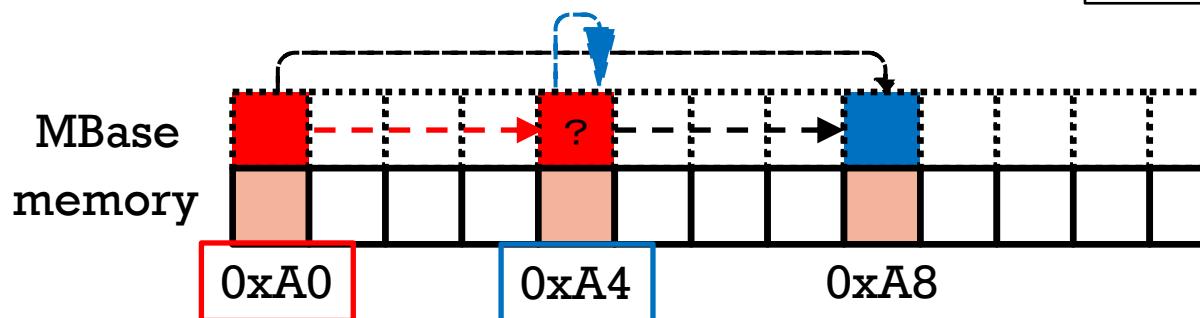
### ■ ベースポインタのベースポインタ

- MBase(base\_addr) = ?
- 以下の例だと、近さ優先ならMBase(0xA4) = **0xA4**となってしまう
- 実際にはMBase(0xA4) = **0xA0**
- ルートポインタを除き、ベースポインタは自身を指さない

```
elem {
    int x;
    int y[2];
}
```

```
elem e;
e.x = 1; e.y[0] = 2; e.y[1] = 3;
int *p = &(e.y[0]); //説明省略
*p = 4; *(p+1) = 5;
```

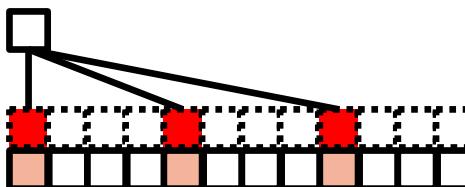
```
mov $1, (%eax)
mov $2, 4(%eax)
mov $3, 8(%eax)
...
mov $4, 0(%ecx)
mov $5, 4(%ecx)
```



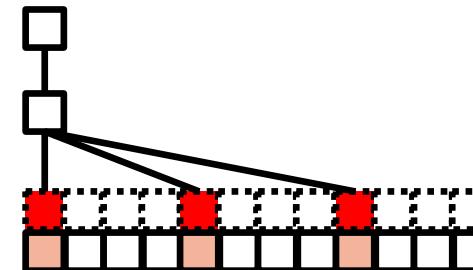
## 3.3: 提案手法(ARRAY DETECTION) OVERVIEW

- 配列の問題点
  - Ex. アクセス位置を見ただけだと次の2つのデータ構造が区別できない

```
elem {  
    int a; int b; int c;  
}
```

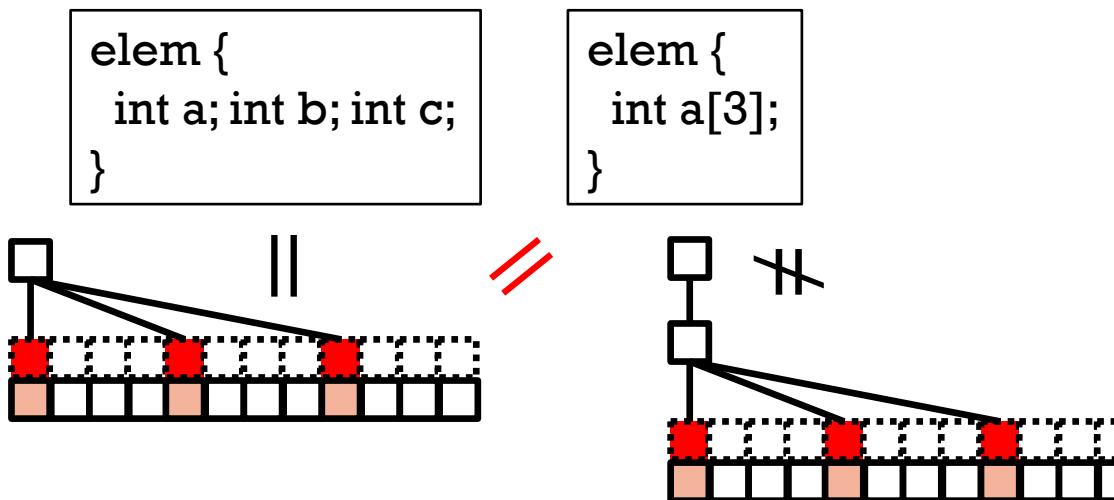


```
elem {  
    int a[3];  
}
```



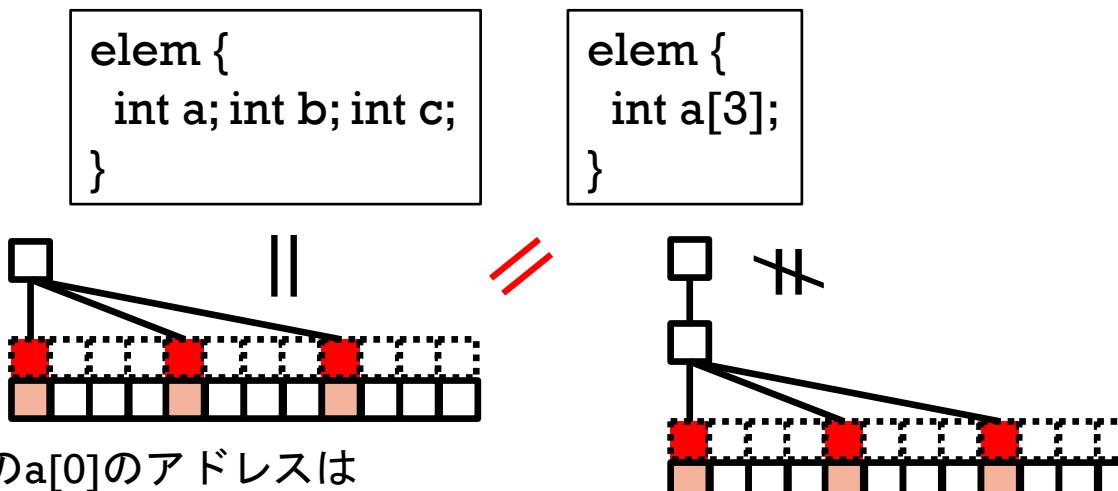
## 3.3: 提案手法(ARRAY DETECTION) OVERVIEW

- 配列の問題点
  - Ex. アクセス位置を見ただけだと次の2つのデータ構造が区別できない



## 3.3: 提案手法(ARRAY DETECTION) OVERVIEW

- 配列の問題点
  - Ex. アクセス位置を見ただけだと次の2つのデータ構造が区別できない



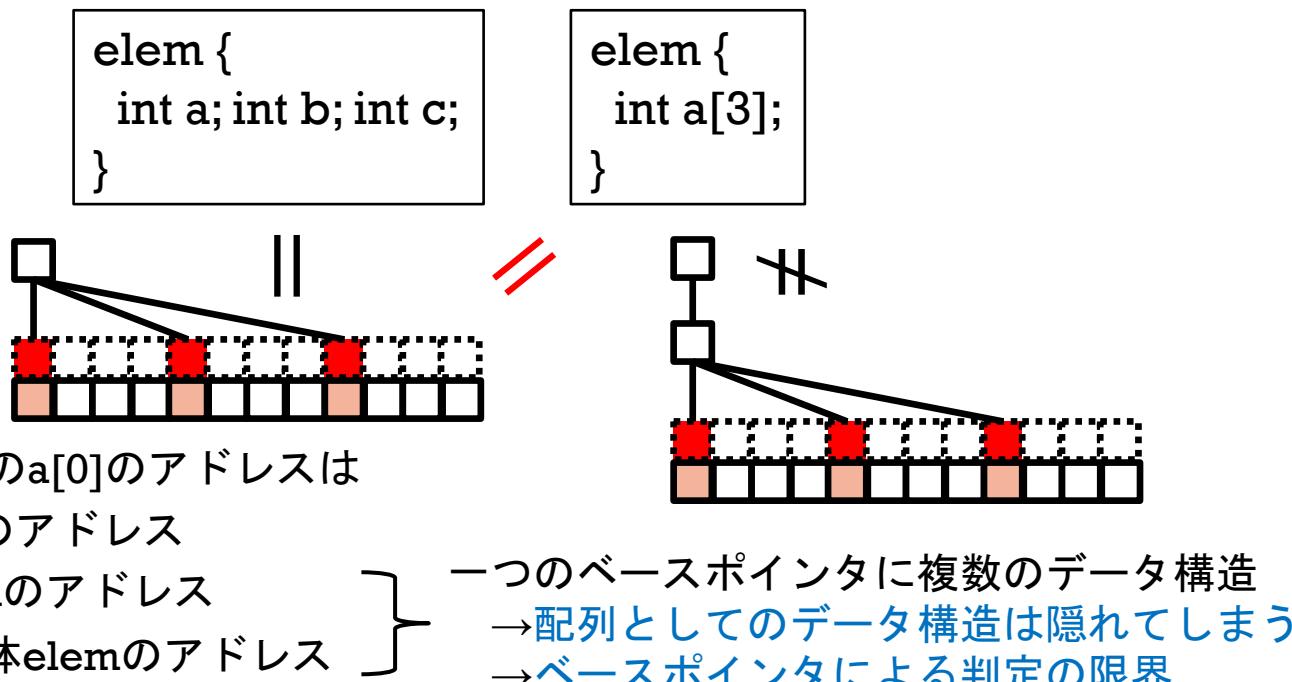
- 原因: 右のa[0]のアドレスは
  - a[0]のアドレス
  - 配列aのアドレス
  - 構造体elemのアドレス

}

一つのベースポインタに複数のデータ構造  
 →配列としてのデータ構造は隠れてしまう  
 →ベースポインタによる判定の限界

## 3.3: 提案手法(ARRAY DETECTION) OVERVIEW

- 配列の問題点
  - Ex. アクセス位置を見ただけだと次の2つのデータ構造が区別できない



- そこで配列はアクセス位置とパターンを調べる
  - Q. 配列と関係の深いパターンとは何でしょう？
  - A. そうループです

## 3.3: 提案手法(ARRAY DETECTION) LOOP DETECTION

- ループの分類

- 軸1:

- I) relative (相対的: `elem = *(prev++)`)
    - II) absolute (絶対的: `elem = array[i]`)
      - ハッシュテーブルとかもこっち

- 軸2:

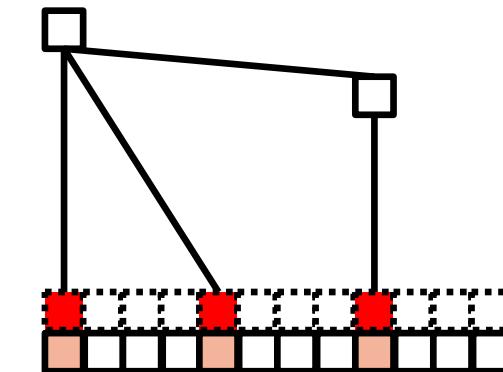
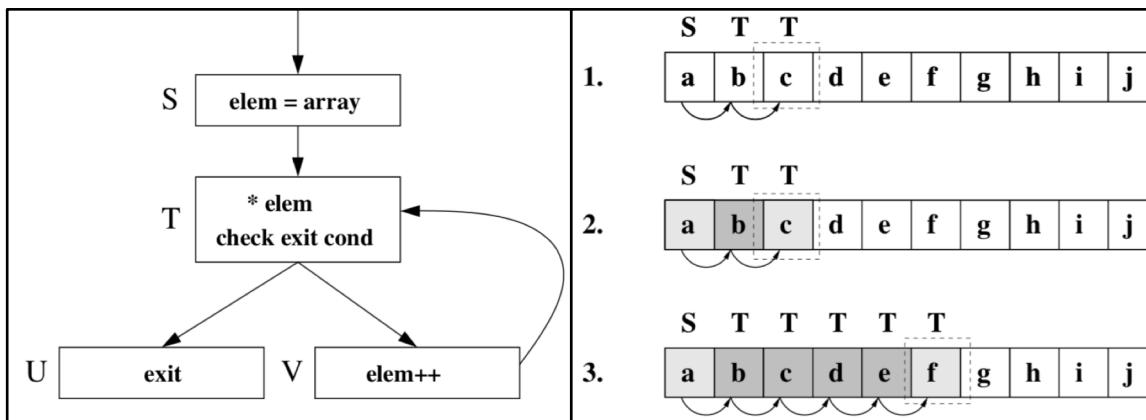
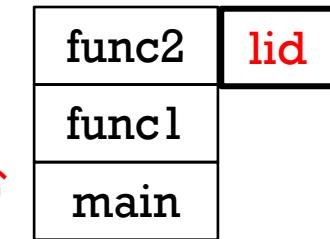
- A) 1 Loop                    1 Array
    - B) Nested Loop            1 Array
    - C) 1 Loop                    Multi Array (optmz: Loop unrollingを含む)
    - D) Nested Loop            Multi Array
    - E) Boundary

- 軸1について場合分けを行う

### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

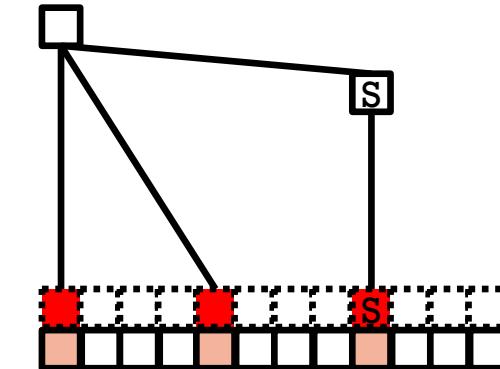
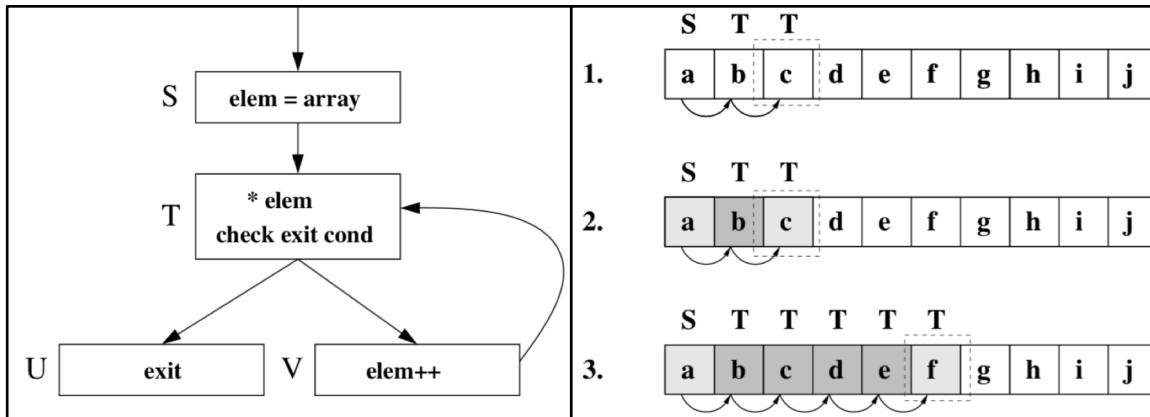
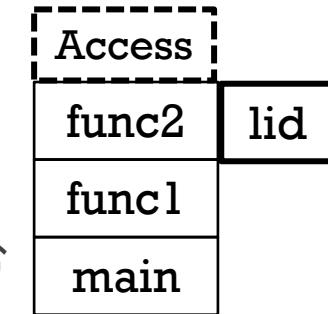
- Loop detection in relative iter
  - elem = \*(prev++)
- Case1) シンプルな1 Loop (展開なし)
  - バックエッジ(V→T)初回で、ループの情報をHStackに積む
    - ループはlid(LoopID)で識別
    - 
    - 
    - 
    -



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

- Loop detection in relative iter
  - elem = \*(prev++)
- Case1) シンプルな1 Loop (展開なし)
  - バックエッジ(V→T)初回で、ループの情報をHStackに積む
    - ループはlid(LoopID)で識別
    - 先頭要素としてループ前のMBase(addr)にSを新たにタグとして付与
  - 
  - 
  -

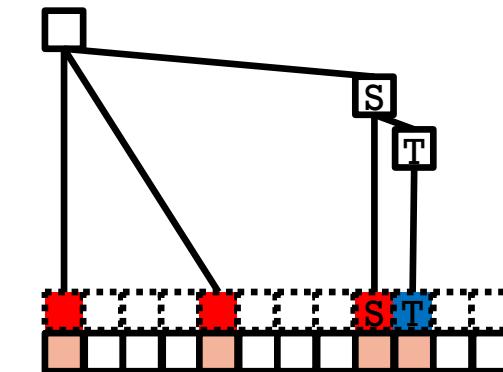
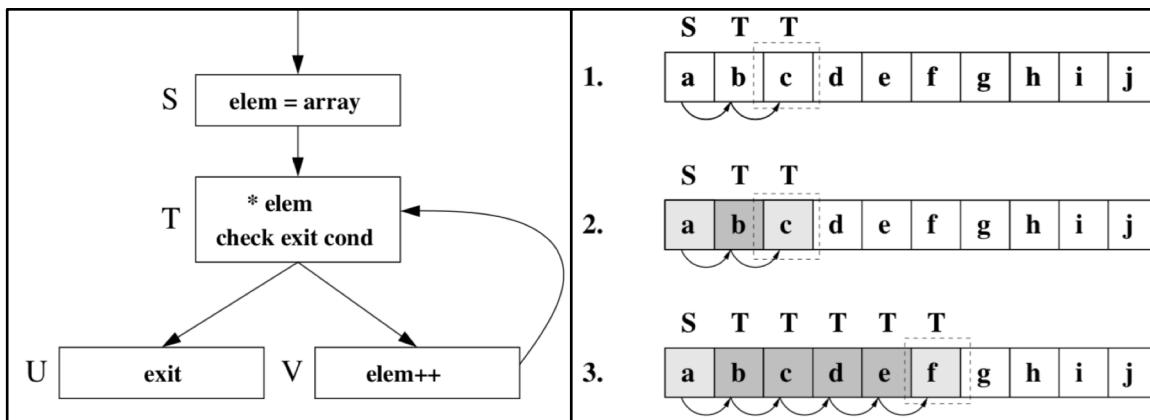


### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

- Loop detection in relative iter
  - elem = \*(prev++)
- Case1) シンプルな1 Loop (展開なし)
  - バックエッジ(V→T)初回で、ループの情報をHStackに積む
    - ループはlid(LoopID)で識別
    - 先頭要素としてループ前のMBase(addr)にSを新たにタグとして付与
    - ループして前要素をベースにした参照→Tをタグとして付与
    - 
    -

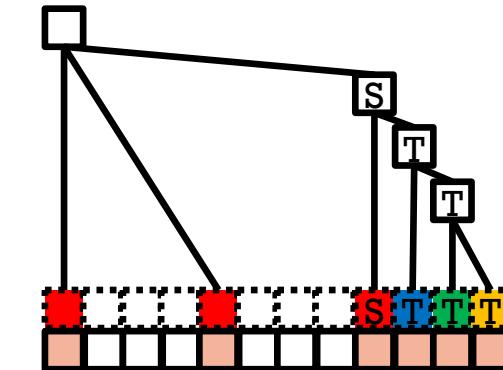
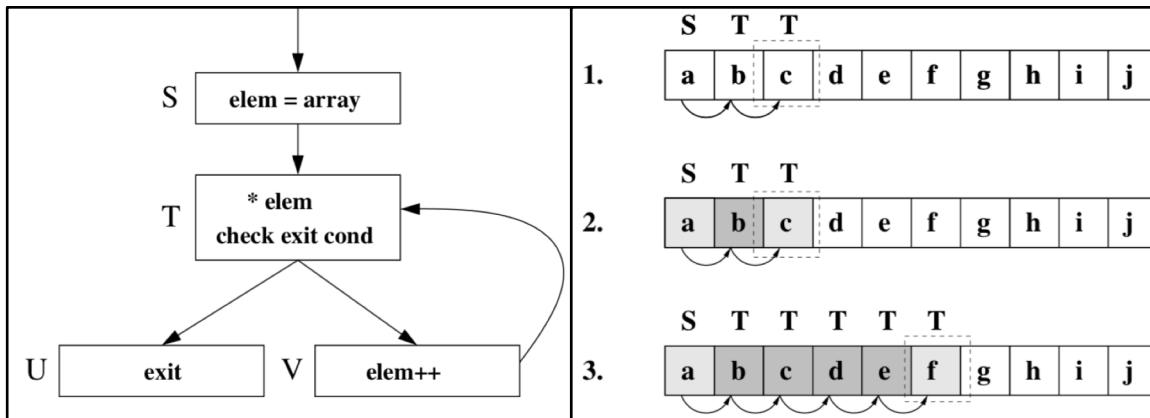
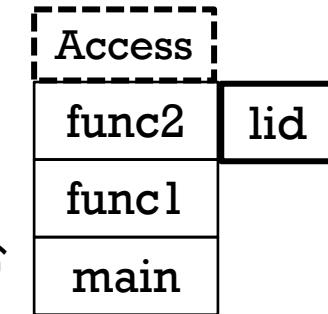
Access	
func2	lid
func1	
main	



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

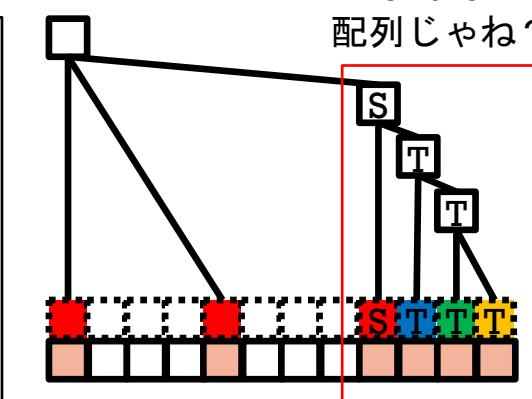
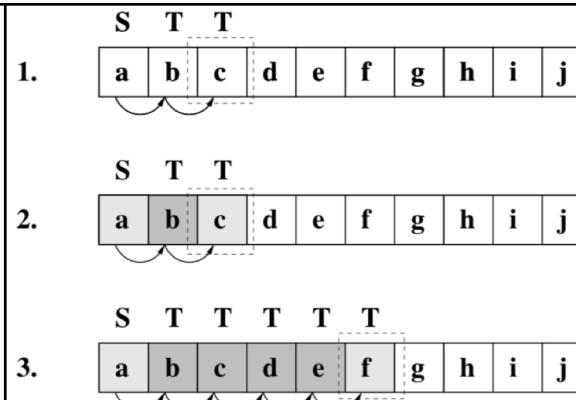
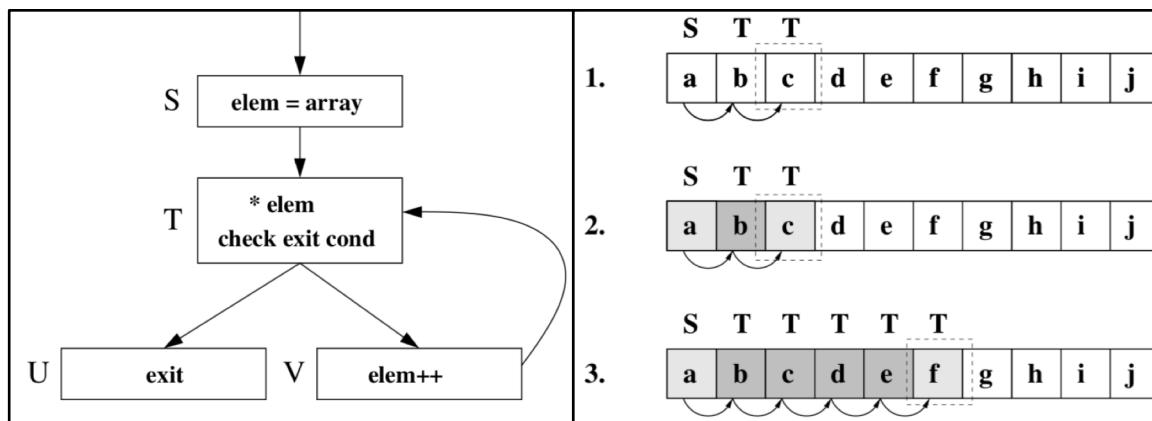
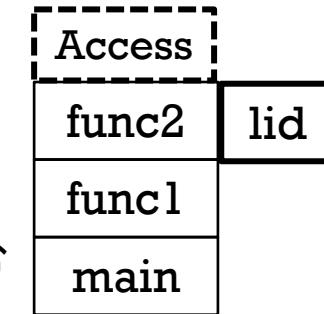
- Loop detection in relative iter
  - elem = \*(prev++)
- Case1) シンプルな1 Loop (展開なし)
  - バックエッジ(V→T)初回で、ループの情報をHStackに積む
    - ループはlid(LoopID)で識別
    - 先頭要素としてループ前のMBase(addr)にSを新たにタグとして付与
    - ループして前要素をベースにした参照→Tをタグとして付与
    - Tが連続すると、HowardはSを先頭とする配列と判断する
      - ただしSが配列に入るかどうかは判断が付きにくい



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

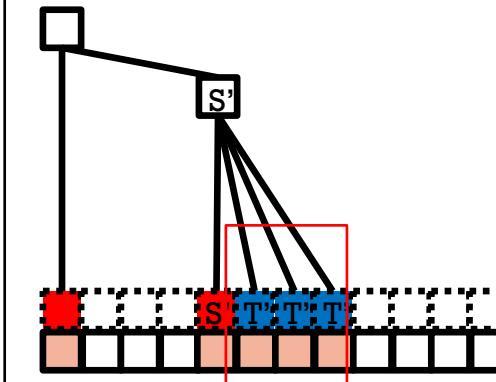
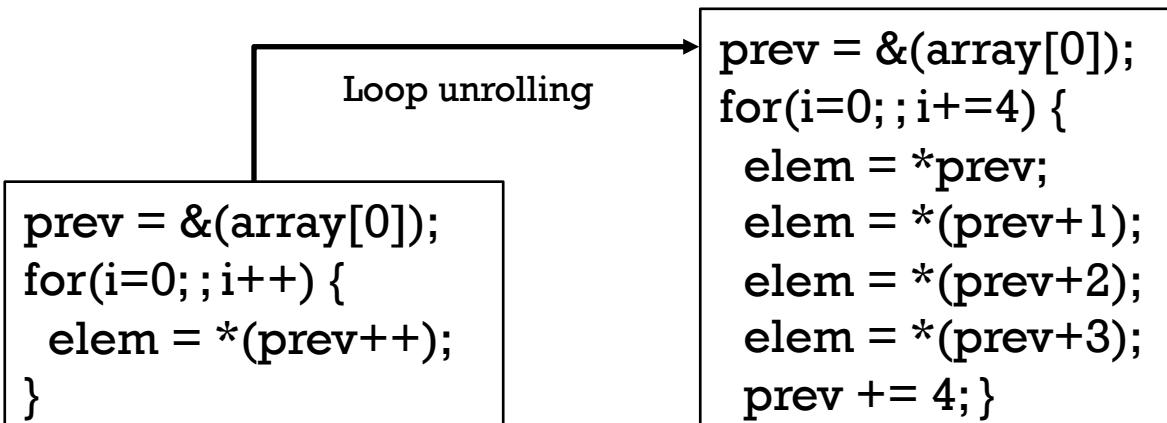
- Loop detection in relative iter
  - elem = \*(prev++)
- Case1) シンプルな1 Loop (展開なし)
  - バックエッジ(V→T)初回で、ループの情報をHStackに積む
    - ループはlid(LoopID)で識別
    - 先頭要素としてループ前のMBase(addr)にSを新たにタグとして付与
    - ループして前要素をベースにした参照→Tをタグとして付与
    - Tが連続すると、HowardはSを先頭とする配列と判断する
      - ただしSが配列に入るかどうかは判断が付きにくい



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

- Case2) 展開済みループ (Loop unrolling)
  - ここではunrollingされても最低1回バックエッジがあると仮定
    - →lidが存在
  - 次の条件を満たす( $S'$ , $T''$ )を持つデータ構造群を基準に判定
    - $S' \geq S \Leftrightarrow S'$ を持つ領域は、先頭要素開始以降にアクセスがあった領域
    - $T'' \geq T \Leftrightarrow T''$ を持つ領域は、ループ内部で $S'$ あるいは $T''$ をベースとした領域
    - この条件を満たすT'を持つ領域を配列要素とみなす
      - ただし、最低でも1回 $T'$ をベースとした領域が必要
  - 
  -



### 3.3: 提案手法(ARRAY DETECTION)

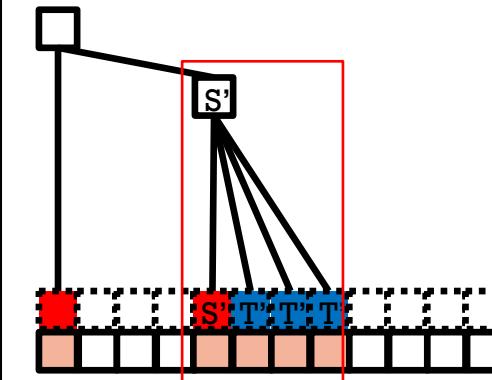
#### LOOP DETECTION (RELATIVE)

- Case2) 展開済みループ (Loop unrolling)
  - ここではunrollingされても最低1回バックエッジがあると仮定
    - →lidが存在
  - 次の条件を満たす( $S'$ , $T''$ )を持つデータ構造群を基準に判定
    - $S' \geq S \Leftrightarrow S'$ を持つ領域は、先頭要素開始以降にアクセスがあった領域
    - $T'' \geq T \Leftrightarrow T''$ を持つ領域は、ループ内部で $S'$ あるいは $T''$ をベースとした領域
    - この条件を満たす**T'を持つ領域を配列要素とみなす**
      - ただし、最低でも1回 $T'$ をベースとした領域が必要
  - 一回のループ内で $S'$ がベースとして複数参照された場合
    - $S'$ も配列要素にみなす(境界チェック)

Loop unrolling

```
prev = &(array[0]);
for(i=0; ; i++) {
    elem = *(prev++);
}
```

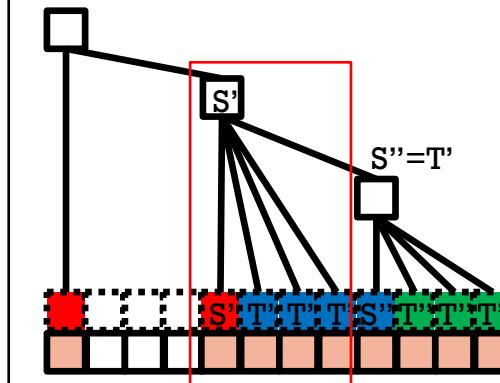
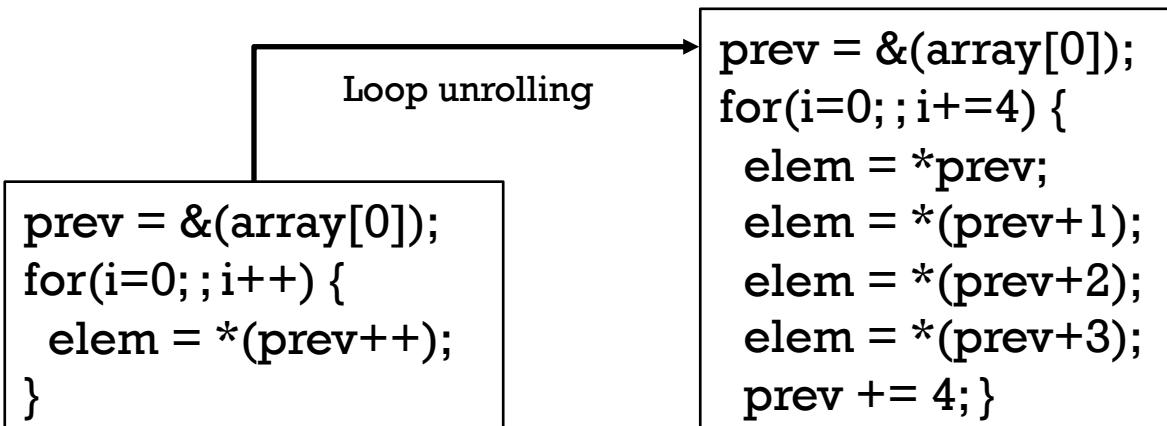
```
prev = &(array[0]);
for(i=0; ; i+=4) {
    elem = *prev;
    elem = *(prev+1);
    elem = *(prev+2);
    elem = *(prev+3);
    prev += 4; }
```



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

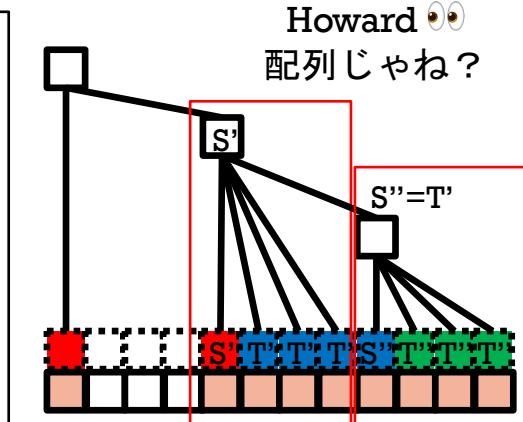
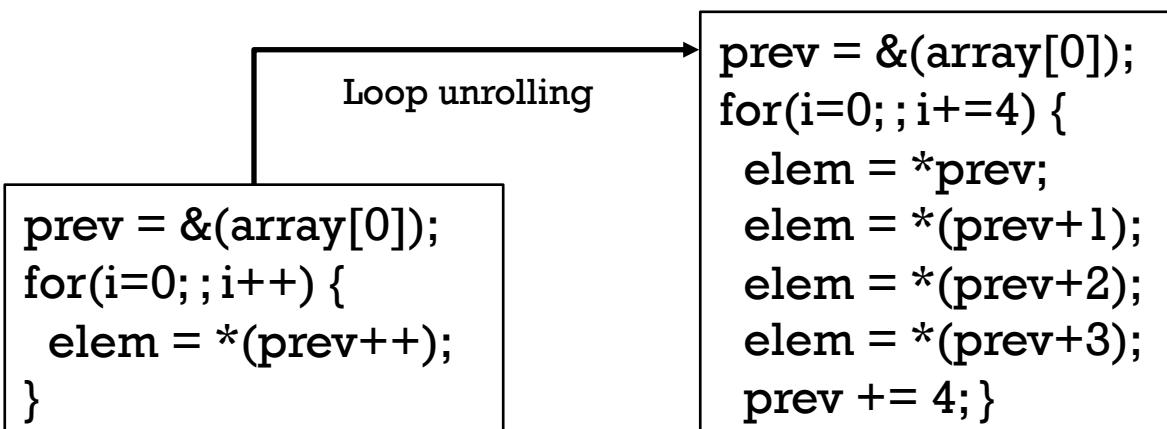
- Case2) 展開済みループ (Loop unrolling)
  - ここではunrollingされても最低1回バックエッジがあると仮定
    - →lidが存在
  - 次の条件を満たす( $S'$ , $T''$ )を持つデータ構造群を基準に判定
    - $S' \geq S \Leftrightarrow S'$ を持つ領域は、先頭要素開始以降にアクセスがあった領域
    - $T'' \geq T \Leftrightarrow T''$ を持つ領域は、ループ内部で $S'$ あるいは $T''$ をベースとした領域
    - この条件を満たす **$T'$ を持つ領域を配列要素とみなす**
      - ただし、最低でも1回 $T'$ をベースとした領域が必要
  - 一回のループ内で $S'$ がベースとして複数参照された場合
    - $S'$ も配列要素にみなす(境界チェック)



### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (RELATIVE)

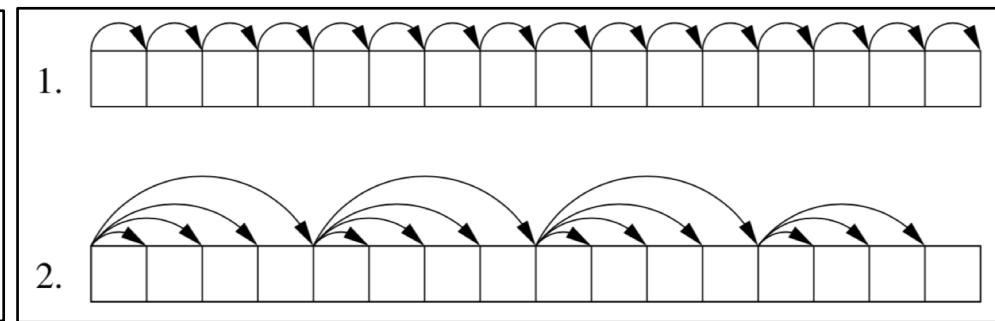
- Case2) 展開済みループ (Loop unrolling)
  - ここではunrollingされても最低1回バックエッジがあると仮定
    - →lidが存在
  - 次の条件を満たす( $S'$ , $T''$ )を持つデータ構造群を基準に判定
    - $S' \geq S \Leftrightarrow S'$ を持つ領域は、先頭要素開始以降にアクセスがあった領域
    - $T'' \geq T \Leftrightarrow T''$ を持つ領域は、ループ内部で $S'$ あるいは $T''$ をベースとした領域
    - この条件を満たす**T'を持つ領域を配列要素とみなす**
      - ただし、最低でも1回 $T'$ をベースとした領域が必要
  - 一回のループ内で $S'$ がベースとして複数参照された場合
    - $S'$ も配列要素にみなす(境界チェック)



### 3.3: 補足: 提案手法(ARRAY DETECTION) LOOP DETECTION (RELATIVE)

- 実は先のunrollingの例だと、正しく構造を解析できていない
  - 配列の要素に配列があるようなデータ構造(図2)として認識される

```
char a[4];
a[0] = 'a';
a[1] = 'b';
a[2] = 'c'
a[3] = char b[4]; ←(0w0 )!?
```



- 解消法
  - 現状、他のアクセスパターン(図1など)を待つしかない

### 3.3: 提案手法(ARRAY DETECTION) LOOP DETECTION (RELATIVE)

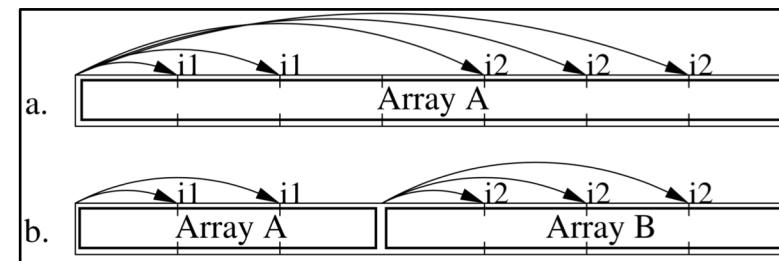
- Case3) 入れ子ループ (Nested Loop)
  - 詳しい説明ができません...
    - この論文の基礎になったTechReport[42]に書いてあるっぽい
    - まだ読めてません...

```
for(){  
    for(){  
    }  
}
```

### 3.3: 提案手法(ARRAY DETECTION)

#### LOOP DETECTION (ABSOLUTE)

- Loop Detection in absolute iter
  - 基本的な直感:
    - 同一ベースポインタを持つデータ構造について
      - 線形的(同一幅: 4bytes幅とか)にマッピング
      - ループ上でマッピング
      - →配列では?
    - ベースポインタから同じデータ構造と判別しやすい
      - 複数の命令からの参照もマッピングできる(図a)
      - ベースポインタが異なる配列 or 構造体(図b)



- 結果論的には:
  - ハッシュテーブルのようなランダムアクセスも区別可
    - ホント?
  - Unrollingの有無もほぼ無視可
    - ホント?

### 3.3: 提案手法(ARRAY DETECTION)

#### まとめ

---

- 相対的判定法:
  - ベースポインタの伝播とループのタイミングから配列を判定
- 絶対的判定法:
  - 同一ベースポインタと線形性から配列を判定
- Howardの強み: 相対的と絶対的の組み合わせ
  - 光と闇が両方そなわり最強に見える
  - 既存技術より正確に配列を識別可能
- 所感: 基本的にはループの存在に依存
  - ループなしで配列操作されると区別がつかないのが現状
    - 完全にループを展開した場合など

## 3.4: 提案手法(FINAL MAPPING)

---

- 最終的にマップしたMBaseからデータ構造を報告
  - 各データ構造の監視開始と監視終了のタイミング
    - ヒープオブジェクト
      - 開始: アロケーションされた時点
      - 終了: 解放時
    - 静的変数/グローバル変数
      - 開始: プログラム開始時 (各変数は発見次第)
      - 終了: プログラム終了時
    - 関数フレーム
      - 開始: コール時
      - 終了: リターン時
      - メタデータはおそらく常に保持(明記なし)

## 4. デモ

---

- Howardの機能
  - シンボルテーブルの再構築
    - ちょっとデモる
  - レガシーコードの保護
    - バッファーオーバーフローとかから保護
    - 今回は紹介省略

## 4. デモ(シンボルテーブル構築)

---

- 対象: wget (stripped)
  - シンボル情報を削いたwgetに対してHowardを適用
  - 実行 (&コアダンプしたという設定)
  - 新たにシンボルテーブルを付与されている
    - デバッガ的にはダンプファイルはstrippedのまま
- GDBにダンプファイル(wget.gdb)を食わせる
  - あら不思議、 strippedなのにシンボル情報が
- おことわり
  - シンボル情報欠落前の**関数名などを復元できるわけではない**
  - 解析したデータ構造に沿った名称を構築
    - 関数名→function0など
    - 変数名→field\_4\_bytes\_0, pointer\_struct\_1\_0など
  - 一部変数の型は有名関数の引数から復元
    - →その他 (タイプシンク)

## 4. デモ(シンボルテーブル構築)

- L.2: ダンプファイルはstripped(シンボル情報欠落)
- L.5-6: ブレークポイント設定

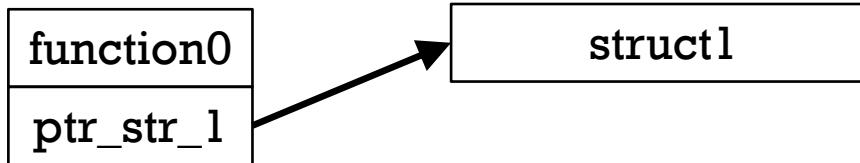
function0

コマンドライン

```
1 # file wget.gdb
2 wget.gdb: ... ,stripped
3 # gdb -q wget.gdb
4 Reading symbols from /.../wget.gdb ...done
5 (gdb) b *0x805adb0
6 Breakpoint 1 at 0x805adb0
7 (gdb) run www.google.com
8 Start program: ...
```

# 4. デモ(シンボルテーブル構築)

- L.13: function0のローカル変数を表示
- L.14-16: ローカル変数の内訳
  - L.15: hostnet構造体へのポインタがある（参照済み）
  - L.16: 空の構造体へのポインタや、inetaddr\_string構造体がある



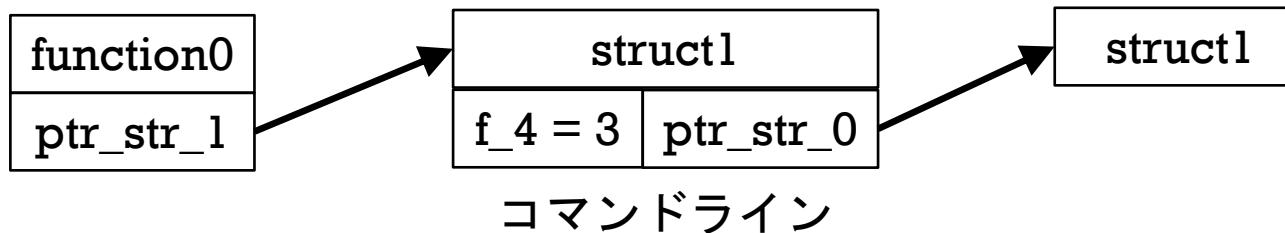
コマンドライン

```

9 Breakpoint 1, 0x805adb0 in function0 ()
10 (gdb) info scope function0
11 Scope for function0:
12 Symbol. Variables_function0 is .... length 152
13 (gdb) print variables function0
14 $1 = {   field_4_bytes_0 = 0, field_4_bytes_1 = 0, ... ,
15           ptr_str_hostnet_0 = 0xbffffec90, ... ,
16           ptr_str_1_0 = 0x0, ... , inetaddr_string_0, ... }
  
```

# 4. デモ(シンボルテーブル構築)

- L.13: 空の構造体ポインタを監視
- L.22-24: 新たに割り当てられた構造体のフィールド
  - L.22: 整数値3が入っている変数と、別の構造体へのポインタ



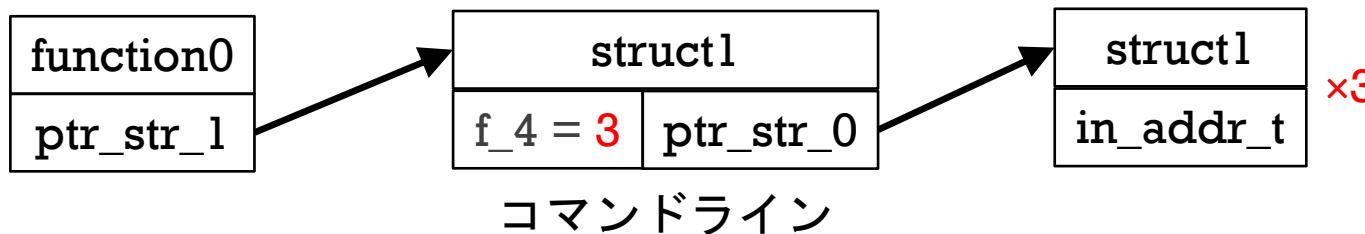
```

17 (gdb) watch var_function0.ptr_str_1_0
18 (gdb) c
19 Old value = {struct str_1 *} 0x0
20 New value = {struct str_1 *} 0x80b2678
21 (gdb) print /x *var_function0.ptr_str_1_0
22 $2 = { field_4_bytes_0 = 0x3, ptr_str_0_0 = 0x80b2690,
23           field_int_0 = 0x0, filed_1_byte_0 = 0x0
24           field_4_bytes_1 = 0x0 }

```

# 4. デモ(シンボルテーブル構築)

- L.26: 構造体内部のポインタが指す構造体のフィールド
- L.30: 配列長と先ほどのptr\_str\_1\_0.field\_4\_bytes\_0が一致
  - ptr\_str\_1\_0.field\_4\_bytes\_0はIPアドレスの個数では？
  - ( 0w0) ウエッ!?



```

25 (gdb) print /x var_function0.ptr_str_1_0.ptr_str_0_0
26 $3 = {   field_4_bytes_0, field_in_addr_t_0 = 0x934d7d4a }
27 (gdb) in_addr_tフィールドをIPアドレスに変換
28 $4 = 0xb7fe46a0 "74.125.77.147"
29 (gdb) ptr_str_1_0.ptr_str_0_0の配列長は？(人間が配列と類推)
30 $5 = 3
31 (gdb) ptr_str_1_0.ptr_str_0_0[1].field_in_addr_t_0をIPに変換
32 $6 = 0xb7fe46a0 "74.125.77.90"

```

## 4. デモ(シンボルテーブル構築)

---

- デモの結論:
  - シンボル情報がなくなっててもデータ構造を回復できたでしょ?
- 注意点:
  - もちろんアクセスが少なかつたりすると正確さは損なわれる
- 小泉's 所感
  - それでも人が類推しないといけないことは多い印象
  - やっぱり関数名や変数名って偉大なんやなって...

# HOWRAD [SLOWINSKA'11] 論文掲載のスクショ

```
# file wget.gdb <---- The binary is stripped
wget.gdb: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
# gdb -q wget.gdb
Reading symbols from /home/          /dynamit_instrumented_binaries/wget/wget.gdb...done.
(gdb) b *0x805adb0 <---- Set breakpoint
Breakpoint 1 at 0x805adb0
(gdb) run www.google.com
Starting program: /home/          /dynamit_instrumented_binaries/wget/wget.gdb www.google.com
[Thread debugging using libthread_db enabled]
--2010-08-09 16:24:00-- http://www.google.com/
①
Breakpoint 1, 0x0805adb0 in function0 ()
(gdb) info scope function0 <---- Display function variables
Scope for function0:
Symbol `variables_function0' is a variable with complex or multiple locations (DWARF2), length 152.
(gdb) print variables function0
$1 = {field_4_bytes_0 = 0, field_4_bytes_1 = 0, pointer_struct_hostent_0 = 0xbffffec90, field_8_bytes_0_unused = 579558798248313200,
pointer_char_0 = 0x30bb14 "\274\t", field_in_addr_t_0 = -1073744880, pointer_struct_l_0 = 0x0, field_l_byte_0_unused = 0 '\000',
field_l_byte_0 = 0 '\000', field_l_byte_1 = 0 '\000', field_8_bytes_1_unused = -4611705105257579776,
inetaddr_string_0 = 0x80b0170 "www.google.com", field_4_bytes_2 = 0}
②
(gdb) watch variables_function0.pointer_struct_l_0
Hardware watchpoint 2: variables_function0.pointer_struct_l_0
(gdb) c
Continuing.
Resolving www.google.com... Hardware watchpoint 2: variables_function0.pointer_struct_l_0
③
Old value = (struct struct_l *) 0x0
New value = (struct struct_l *) 0x80b2678
0x0805af5f in function0 ()
(gdb) print /x *variables_function0.pointer_struct_l_0 <---- Display a wget structure
$2 = {field_4_bytes_0 = 0x3, pointer_struct_0_0 = 0x80b2690, field_int_0 = 0x0, field_l_byte_0 = 0x0, field_4_bytes_1 = 0x0}
(gdb) print /x *variables_function0.pointer_struct_l_0.pointer_struct_0_0 <---- Display the structure's fields
$3 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x934d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_l_0.pointer_struct_0_0.field_in_addr_t_0)
$4 = 0xb7fe46a0 "74.125.77.147"
(gdb) print malloc_usable_size(variables_function0.pointer_struct_l_0.pointer_struct_0_0) / sizeof(*variables_function0.pointer_struct_l_0.
pointer_struct_0_0)
$5 = 3
(gdb) print /x variables_function0.pointer_struct_l_0.pointer_struct_0_0[1]
$6 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x634d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_l_0.pointer_struct_0_0[1].field_in_addr_t_0)
$7 = 0xb7fe46a0 "74.125.77.99"
(gdb) print /x variables_function0.pointer_struct_l_0.pointer_struct_0_0[2]
$8 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x684d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_l_0.pointer_struct_0_0[2].field_in_addr_t_0)
$9 = 0xb7fe46a0 "74.125.77.104"
(gdb) 
```

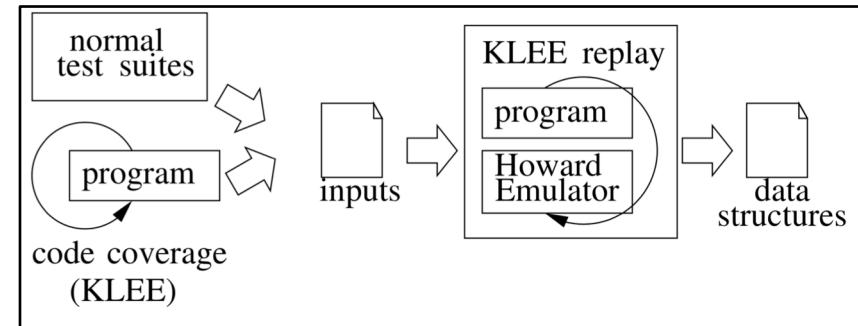
# 5. 評価実験

- 実験環境:

- Linux 2.6.31-19 on QEMU
- with KLEE on LLVM 2.6

- 条件設定:

- KLEEを使ってカバレッジを確保



Prog	LoC	Size	Funcs %	Vars %	How tested?	KLEE %
wget	46K	200 KB	298/576 (51%)	1620/2905 (56%)	KLEE + test suite	24%
fortune	2K	15 KB	20/28 (71%)	87/113 (77%)	test suite	N/A
grep	24K	100 KB	89/179 (50%)	609/1082 (56%)	KLEE	46%
gzip	21K	40 KB	74/105 (70%)	352/436 (81%)	KLEE	54%
lighttpd	21K	130 KB	199/360 (55%)	883/1418 (62%)	test suite	N/A

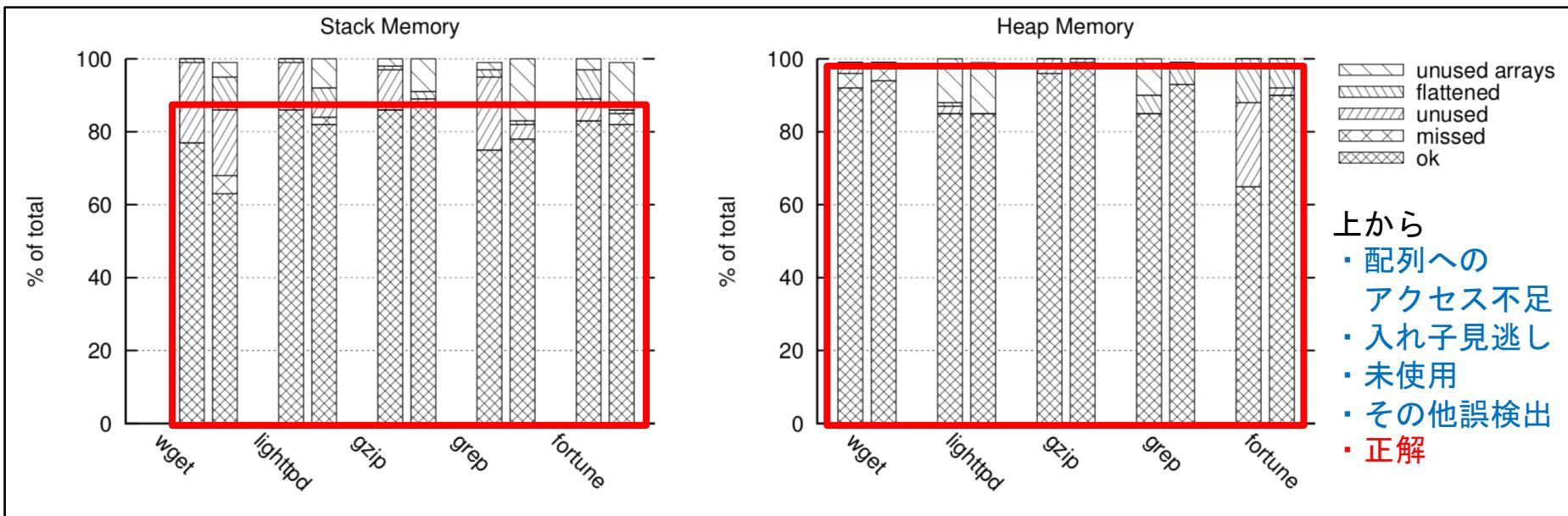
表1: カバレッジ  
 Funcs: 実行関数の割合、Vars: 実行変数の割合  
 KLEE: KLEEが追加でテストした割合

## 5. 評価実験 精度評価

---

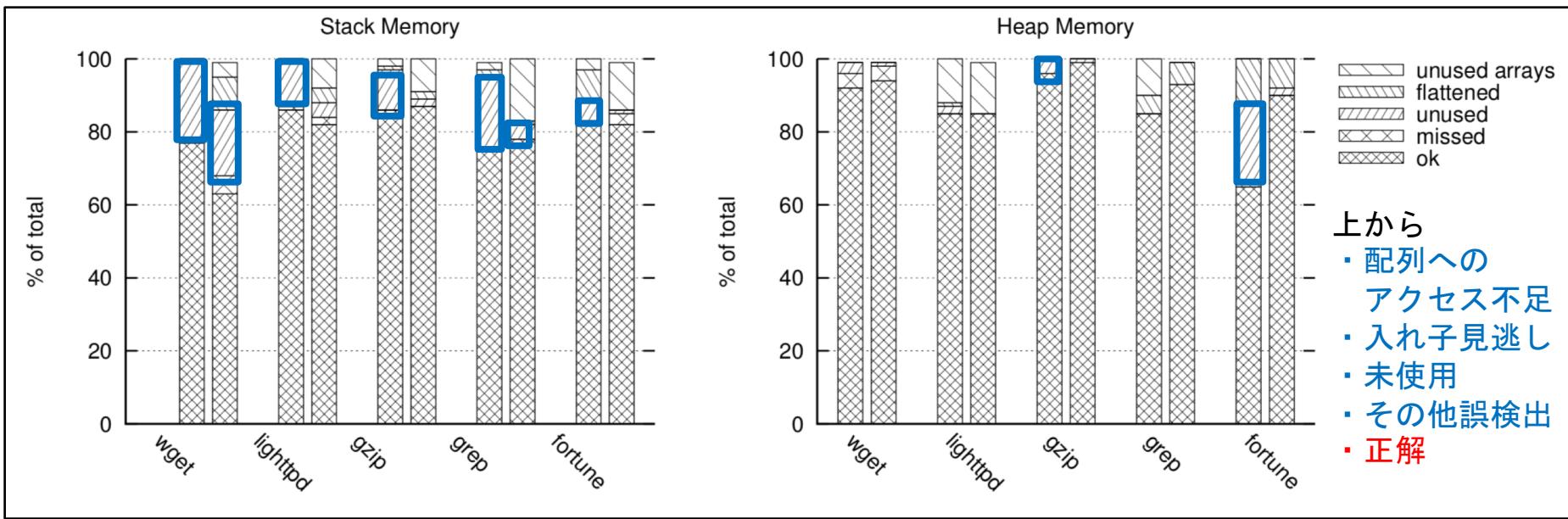
- 精度指標について:
  - 実際のソースコードでのデータ構造を正解とする
    - コンパイル最適化などで正解とどうしても一致しない場合もある
    - What you see is not what you execute.[]
  - テスト全体で使用された変数、配列、構造体を最小単位とする
  - 未使用関数のフレームはカウントから除外
  - 各データ構造の**数とバイト数**によるPrecisionを計算
    - 次ページ以降のグラフの各プログラムについて
      - 左のバー: データ構造数から導出したPrecision
      - 右のバー: データ構造バイト数から導出したPrecision

# 5: 評価実験 精度結果 ①



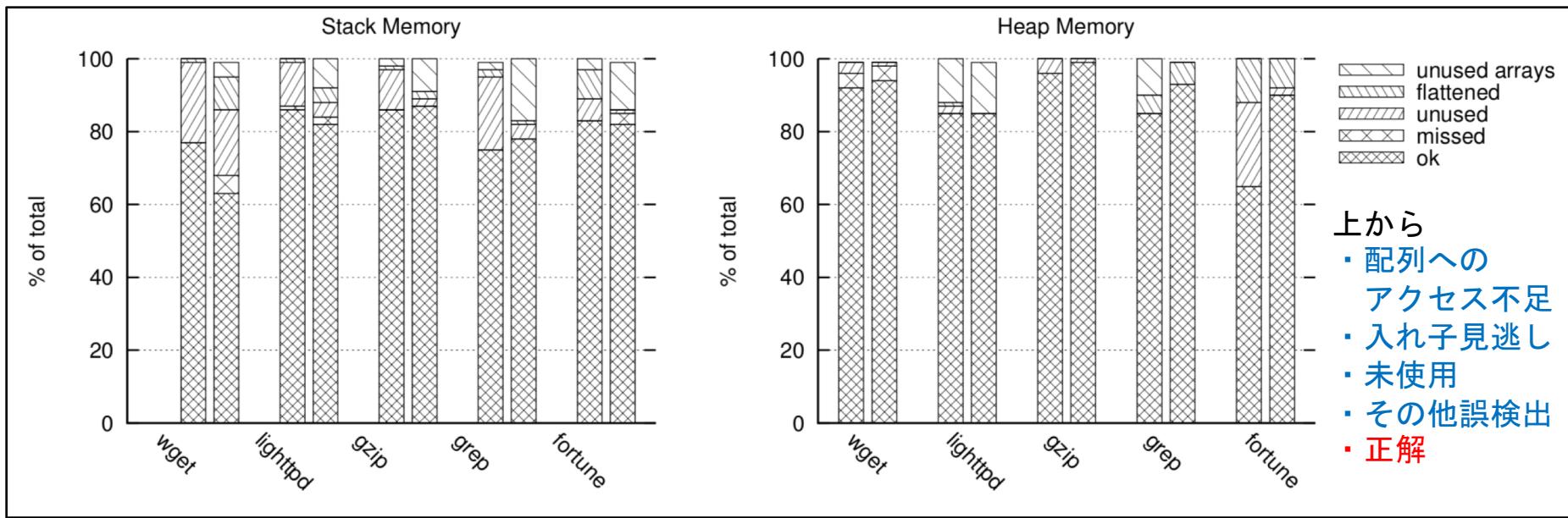
- 大体は完全に識別(ok) or アクセスなしによる見逃し(unused)
  - 区割りの誤検出はあれど、事実上すべての構造の長さは正解
  - 
  - 
  -

# 5: 評価実験 精度結果 ①



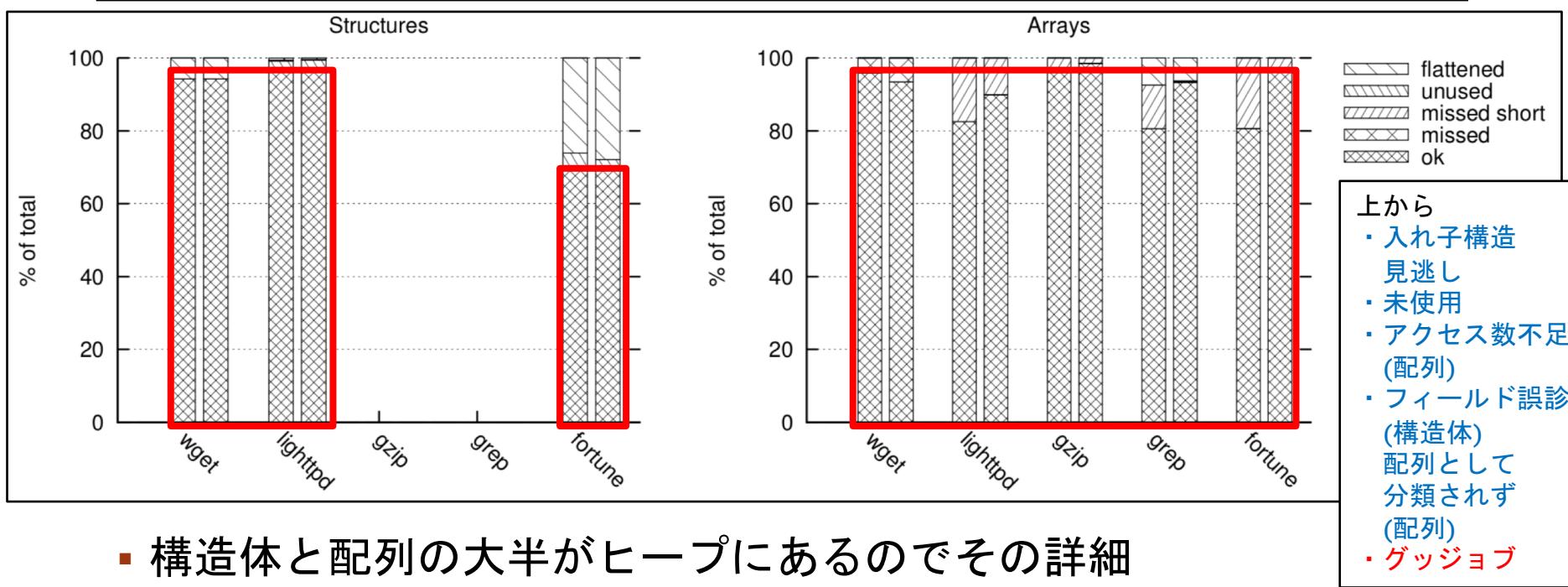
- 大体は完全に識別(ok) or アクセスなしによる見逃し(unused)
  - 区割りの誤検出はあれど、事実上すべての構造の長さは正解
- 入れ子になってる構造体を見逃す場合がある(**flattened**)
  - 
  -

# 5: 評価実験 精度結果 ①



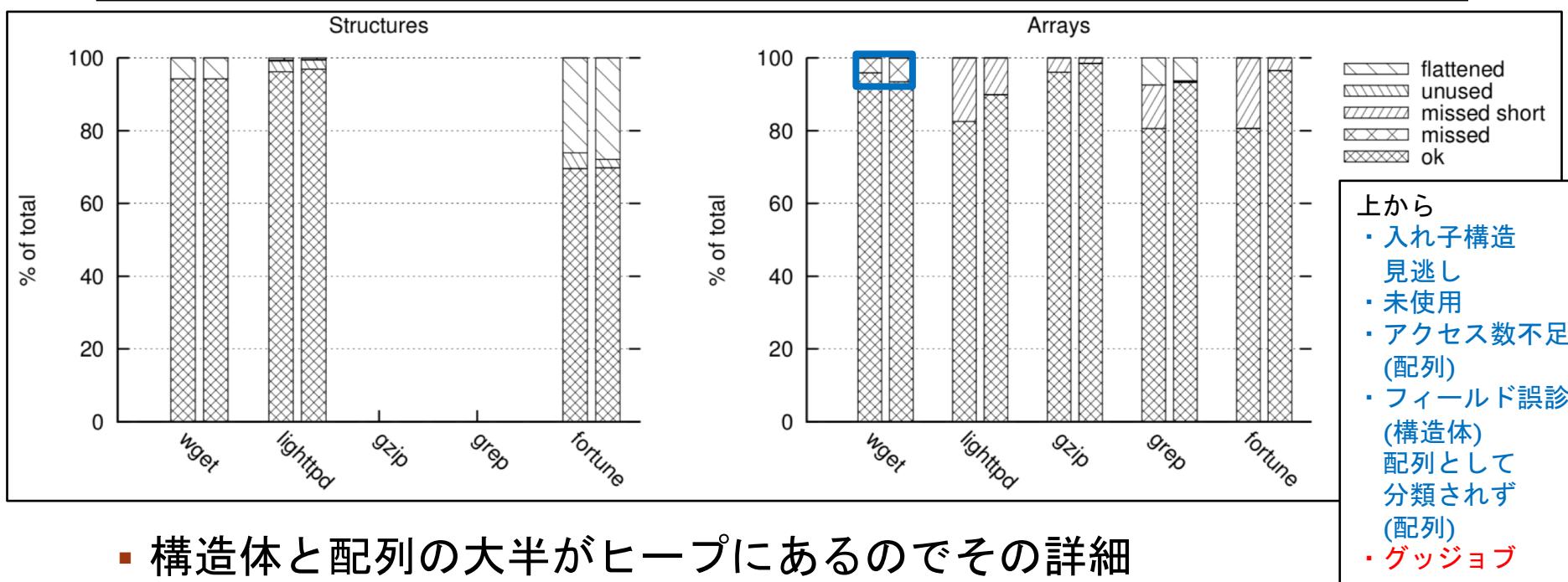
- 大体は完全に識別(ok) or アクセスなしによる見逃し(unused)
  - 区割りの誤検出はあれど、事実上すべての構造の長さは正解
- 入れ子になってる構造体を見逃す場合がある(flattened)
- 最大の誤検出要因は4bytes幅の配列と判断すること
  - 配列を否定できるパターンが少ない

## 5: 評価実験 精度結果② (ヒープ)



- 構造体と配列の大半がヒープにあるのでその詳細
- 大半は正確に識別できている(ok)
- 
- 
-

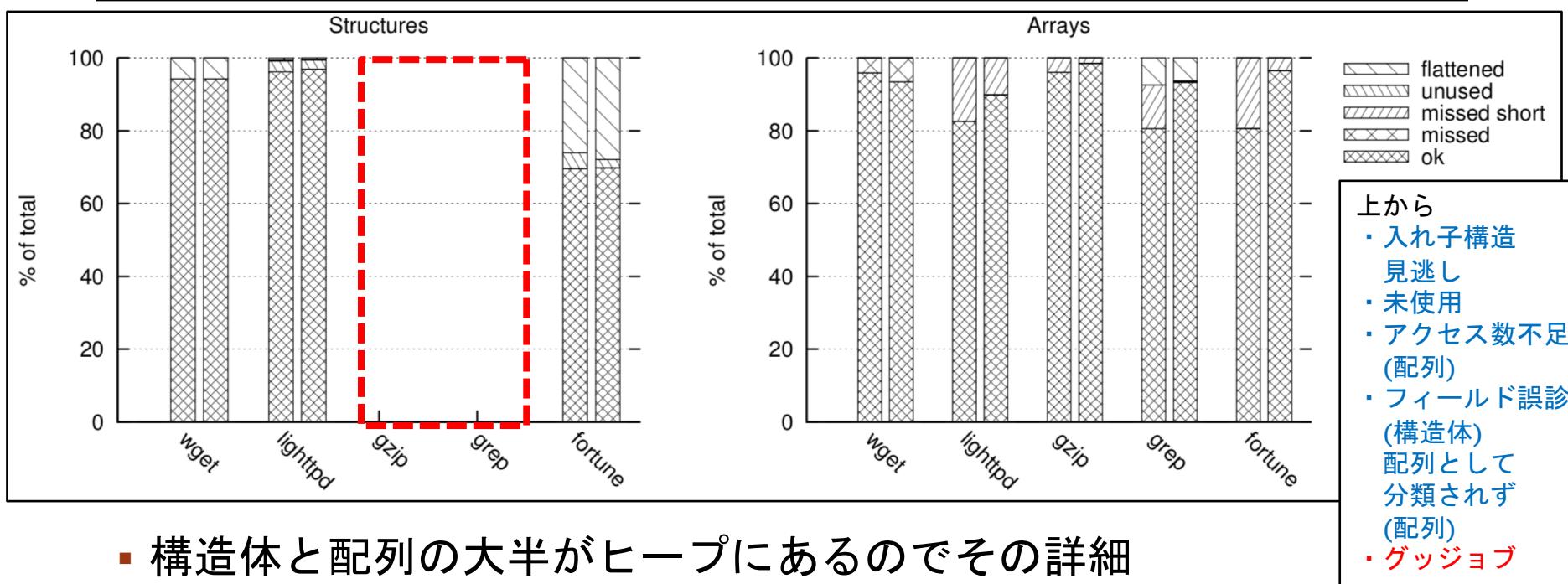
## 5: 評価実験 精度結果② (ヒープ)



- 構造体と配列の大半がヒープにあるのでその詳細
- 大半は正確に識別できている(ok)
- 完全に誤ったのはwget内の配列
  - 再割り当てされたchar[]をint[]と誤判定
- 

上から  
 • 入れ子構造  
 見逃し  
 • 未使用  
 • アクセス数不足  
 (配列)  
 • フィールド誤診  
 (構造体)  
 配列として  
 分類されず  
 (配列)  
 • グッジョブ

## 5: 評価実験 精度結果② (ヒープ)



- 構造体と配列の大半がヒープにあるのでその詳細
- 大半は正確に識別できている(ok)
- 完全に誤ったのはwget内の配列
  - 再割り当てされたchar[]をint[]と誤判定
- gzipとgrepはヒープ内に構造体が生成されず

## 5.1: 評価実験(オーバーヘッド)

---

- 軽く聞き流してください
  
- オーバーヘッドの大半はカバレッジ確保のための実行
  - それでもマニュアルでやるよかずつと早い
    - 早いのはKLEEのおかげでは...(小泉)
- 最長はgrepで15分
  - どういう入力で実行したのかは分からぬけども...(小泉)
- 全然最適化してない割には現実的な数値に収まった、とのこと

## 6. まとめ

---

- Howardはデータ構造解析器である
  - 新規性:
    - 対象がCバイナリ（デバッグ情報なし）
    - ベースポインタとルートポインタを用いた階層構造の識別
    - ループ検知による配列識別
  - 結果:
    - 大半の一般的なデータ構造を正しく識別した
    - 入れ子構造、配列は誤判定しやすい傾向がある
    - カスタムアロケータ、obfuscationなどまだ対応できない技術も

# 感想

---

- 大半が経験則なので、特殊な例では解析できない、という印象
  - マルウェアやウイルスの解析という点では弱いかも？
- Cバイナリでは初、という触れ込みのせいか実験が乏しい印象
  - Rewardsなどとの比較も欲しかった
  - 一方で、考慮すべき要素をめっちゃ羅列してくれた
  - あと伝播のところもう少し詳しく書いてくれ
- アクセスを全部拾うので、IntelPinだとオーバヘッドが大きそう
  - サンプリング適用でも動きそうなら実験してみたい
  - アクセスパターンから型の一致は類推できるかもしれない

# その他(制限): ALLOCA

---

- `alloca`
  - 動的アロケーション
  - 割り付け位置は関数スタック (not heap)
- `alloca`によってフレームのデータ構造が変化する
  - 一時変数とローカル変数/引数の区切りが分からなくなる
- Howard論文内
  - 関数アドレスからフレーム構造が特定できると仮定（非CF依存）
  - 実際には...
    - 関数フレームにはダミータグの伝搬
    - ダミータグの差異で`alloca`による構造変化を識別
  - タグが伝播すればローカル変数or引数、そうでなければ一時変数

# その他(制限): JUMP INSTEAD OF CALL

- 機能的にはcall/ret以外にもjmpでcallを再現可
- だがっ...！それはレアケース...！起こりやしない...！

ざわ...

ざわ...

ざわ...

ざわ...

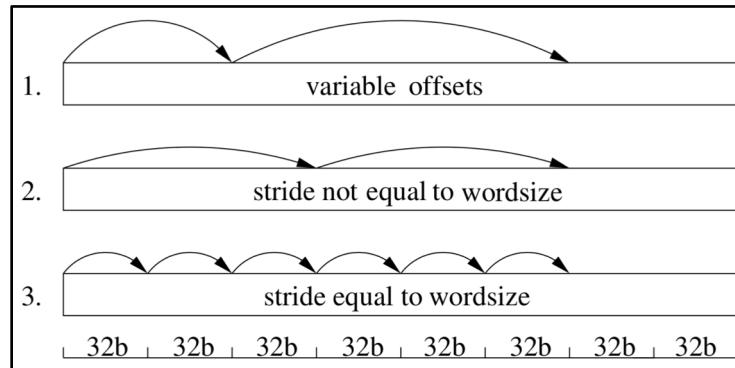
- 一応espのタグを検査することで他の関数へ遷移したかチェック
  - もし遷移していれば解析対象から除外
- →基本的に想定外の処理をされるとコールスタックは再現不可

# その他(制限): MEMSET/MEMCPY

---

- memset: memcpy:

- 対象の構造体や配列の初期化、コピーに使用
- 問題なのは、**対象構造体の全byteにアクセスすること**
  - memcpyならコピー元、コピー先両方
  - →アクセスパターンとしては文字列(1byte幅の配列)に見える
  - 経験的に、1byte幅のアクセス以外のパターンがあればそちらを優先
- ①不規則なアクセスパターン（最優先）
- ②1byte長以外の規則的なアクセスパターン
- ③1byte長の規則的なアクセスパターン（他のパターンを疑う）



# その他(制限): CUSTOM ALLOCATOR

---

- 結論：ムリです
  
- カスタムアロケータ/my\_malloc/俺malloc:
  - まとめてヒープ領域を確保、自身で管理する手法
  - 利点: malloc回数が減少→高速化
- ルートポインタを特定できない
  - →せいぜい4byte配列に特定するのが関の山状態

# その他(制限): COMPILER OPTMZ

---

- コンパイラ最適化 (関係ある箇所)
  - レイアウト関係
    - 構造体フィールドの並び替え
    - 並び替え後の構造を計測することは可能
      - 元のソースコードとは異なる可能性は高い
  - ループ関連
    - 展開
      - アクセスパターンから配列の判定をミスるかも(3.2)
      - ピーリング: 最初/最後のループ内操作をループ外に出すこと
        - 境界チェックに影響
      - ブロック化: キャッシュに乗るようにループ内の処理を調整すること
        - 一種のunrolling? → 配列構造がうまく検知できなくなるかも
  - フレーム関連
    - インライン化
      - 関数フレームが変化
    - 引数のすり抜け
      - 引数がespを介さない → ベースポインタの構築ミス

# その他(制限): OBFUSCATING

---

- Obfuscating(難化):
  - リバースエンジニアリングされないようにバイナリを複雑にすること
  - 配列の難化
    - 配列を分割/マージ、次元の変更(二次元を一次元など)
    - →ムリポ<sup>°</sup>
  - 変数の難化
    - 重要変数の隠蔽(代わりに動的な値などを使用)
    - 変数を分割/マージ
    - →ムリポ<sup>°</sup>
  - 解析封じ
    - 仮想空間では動かないように設計する
      - ネイティブでしか働きたくないでござる
  - →現状難化前には復元できない

## その他(制限): UNION/タイプシンク

---

- unionをフィールドに持つてると、データ構造に複数の可能性
  - Howardは全可能性を報告
    - unionとして報告するわけではない?
    - もちろん誤判定にもつながる
- タイプシンク:
  - 有名関数は引数の型が分かっている
    - 有名関数: システムコールやライブラリ
    - **有名関数の引数から型を類推する**
  - Rewardなどはタイプシンクを使って型を類推している
  - 今回の紹介ではデモのところでちょっと出てきた
    - Howard自体はタイプシンクの機能を持つが、実験では不使用