Analyzing Memory Accesses in Object Code

Gogul Balakrishnan
Thomas W. Reps
University of Wisconsin - Madison

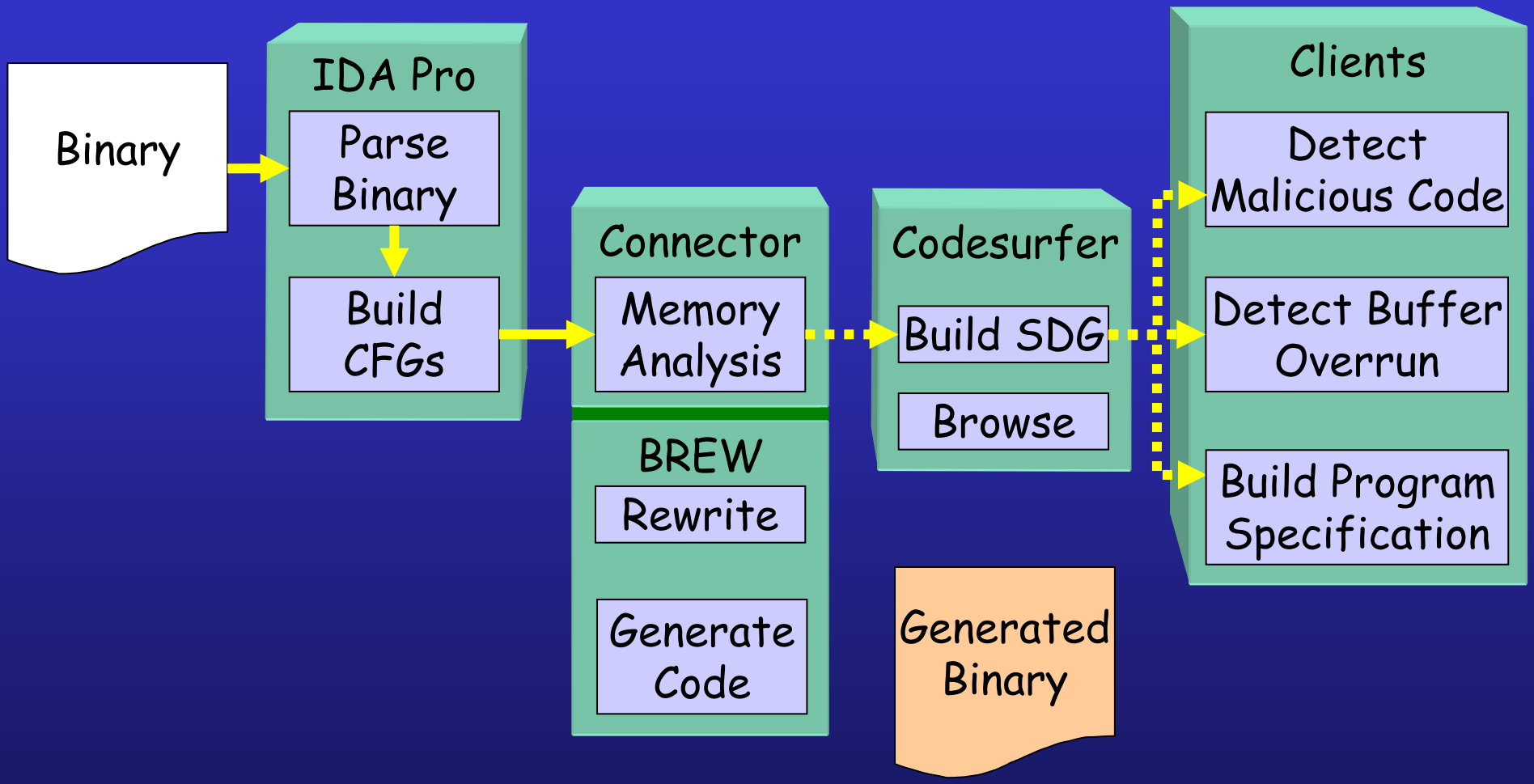
Overall Goal

- Develop an infrastructure for object code analysis
- Create an Intermediate-Representation(IR)
  - In the absence of debugging/symbol-table info
  - Similar to IR for source-code programs
    - CFG for procedures,
    - used, defined, conditionally-killed (c-killed) variables for CFG nodes
    - points-to sets
    - etc.
- Use IR for analyzing object code

Milestones

- CodeSurfer/x86
  - Joint work with GrammaTech, Inc.
  - An tool for browsing("surfing"), inspecting, and analyzing x86 executables

Milestones - Codesurfer/x86



Creating an IR for Object Code (Difficulties)

- Explicit memory addresses to manipulate data
  - e.g. mov [4], edx
  - need a handle for contents of memory locations
  - like variables in C programs
  - our solution: a-locs
- Indirect addressing mode
  - e.g. mov [ecx], edx
  - need a pointer-analysis like algorithm
  - our solution: Value-set analysis

Overview

- Example
- A-locs
  - memory-regions
- Value-set analysis
  - value-set
- Affine-relations analysis
- Running Times
- Future work

Running Example

```
int arrVal=0, *pArray2;

int main() {
    int i, a[10], *p;
   /* Initialize pointers */
    pArray2=&a[2];
    p=&a[0];
   /* Initialize Array*/
    for(i=0; i<10; ++i) {
        *p=arrVal;
        p++;
    }
   /* Return a[2] */
    return *pArray2;
}
```

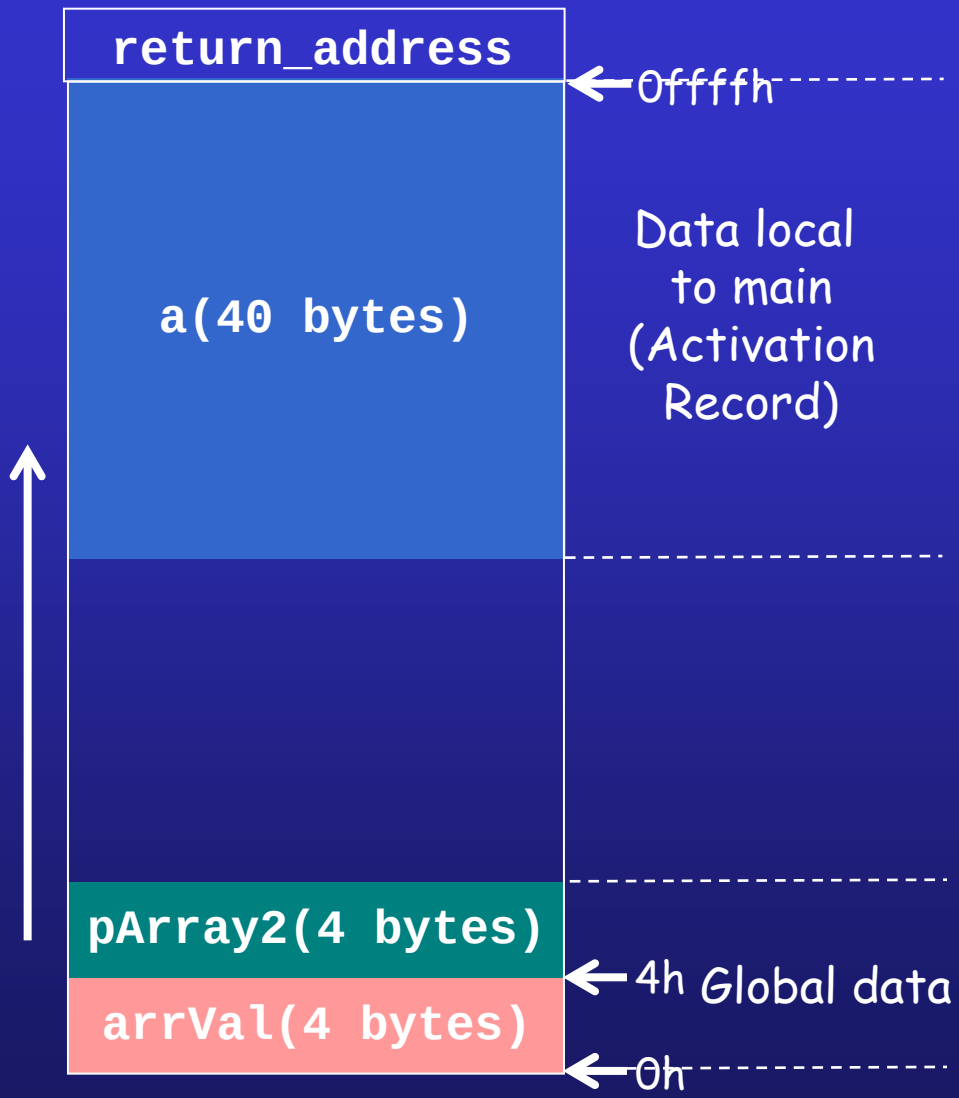
```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40         ;adjust stack
lea     edx, [esp+8]    ;
mov     [4], edx        ;pArray2=&a[2]
lea     ecx, [esp]      ;p=&a[0]
mov     edx, [0]        ;

loc_9:
    mov     [ecx], edx  ;*p=arrVal
    add     ecx, 4      ;p++;
    inc     ebx         ;i++
    cmp     ebx, 10     ;i<10?
    jl      short loc_9 ;

mov     edi, [4]    ;
mov     eax, [edi]  ;return *pArray2
add     esp, 40
retn
```

Running Example

```
int arrVal=0, *pArray2;

int main() {
    int i, a[10], *p;
   /* Initialize pointers */
    pArray2=&a[2];
    p=&a[0];
   /* Initialize Array*/
    for(i=0; i<10; ++i) {
        *p=arrVal;
        p++;
    }
   /* Return a[2] */
    return *pArray2;
}
```

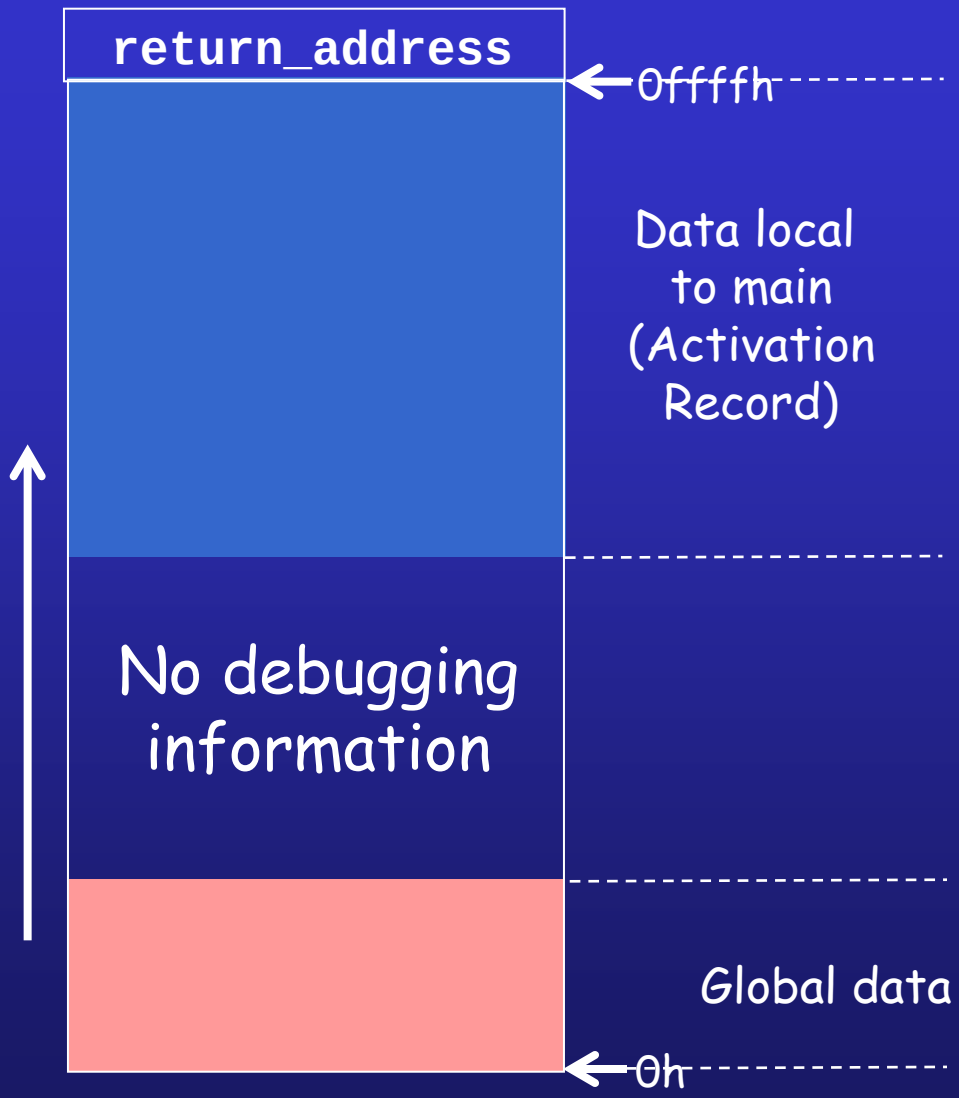
```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40         ;adjust stack
lea     edx, [esp+8]    ;
mov     [4], edx        ;pArray2=&a[2]
lea     ecx, [esp]      ;p=&a[0]
mov     edx, [0]        ;

loc_9:
    mov     [ecx], edx  ;*p=arrVal
    add     ecx, 4      ;p++;
    inc     ebx         ;i++
    cmp     ebx, 10     ;i<10?
    jl      short loc_9 ;

mov     edi, [4]    ;
mov     eax, [edi]  ;return *pArray2
add     esp, 40
retn
```

Running Example - Address Space




```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```

Running Example - Address Space




```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```

A-loc in Object Code

- Handle for memory contents
  - like a variable in a C program
- In object code analysis
  - keep track of information in memory
  - for e.g., used, killed, and c-killed sets in IR
    - in term of a-locs

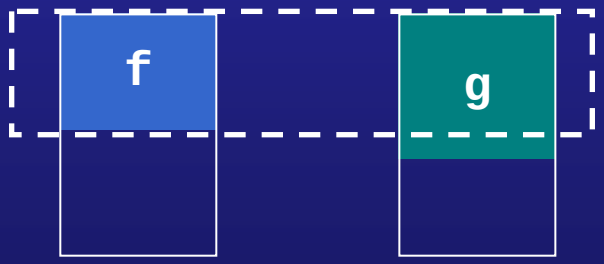
A-loc in Object Code

- One a-loc per memory address?
  - NO !
  - too many addresses
    - object code analysis too slow
  - address not known for heap allocations

A-loc in Object Code

- One a-loc per memory address? ✗
  - NO !
  - address is overloaded
    - one address may be used for different variables
    - e.g., local variables of **f** and **g** share addresses
    - imprecision in analysis

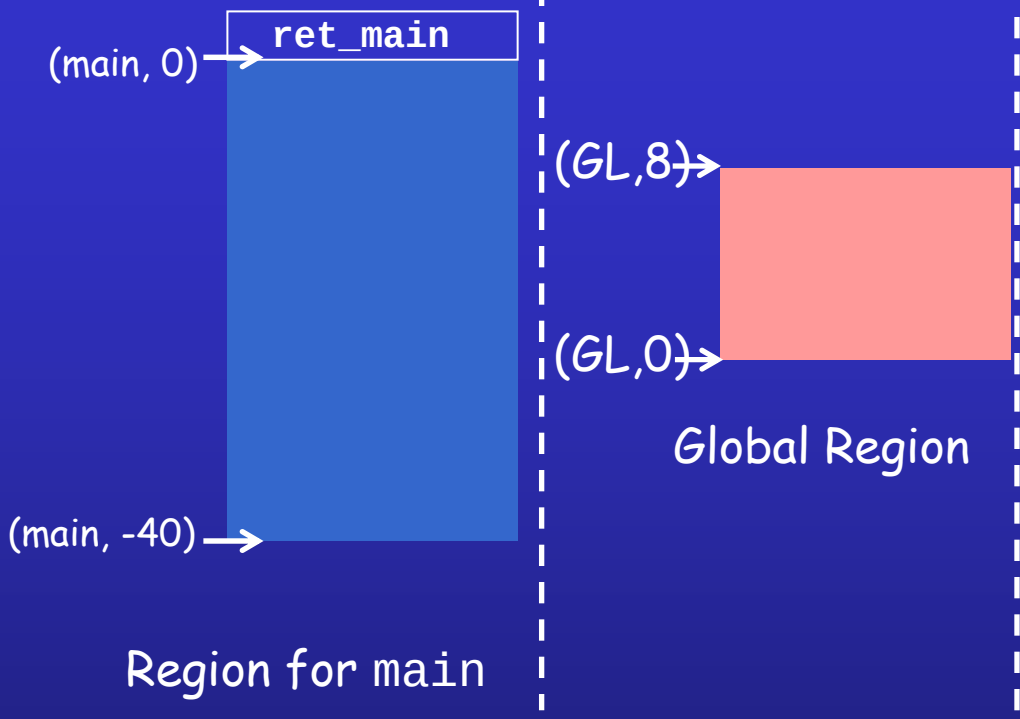
```
main() {
  f();
  …
  g();
  …
}
```



Memory-regions

- Ignore low-level positions of activation records
- Assume non-overlapping regions in address-space
  - One region per procedure
    - All runtime ARs
  - One region per malloc
    - All memory allocated by statement
  - One region per global
  - Memory-regions
- Memory address using memory regions
  - (memory-region, offset within region)

Example - Memory-regions

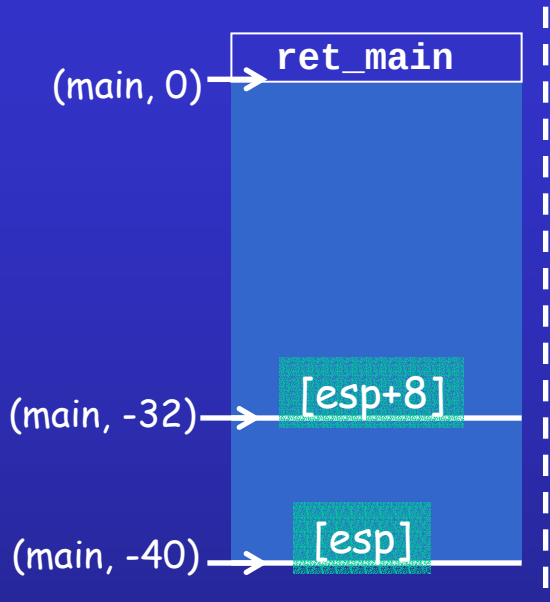


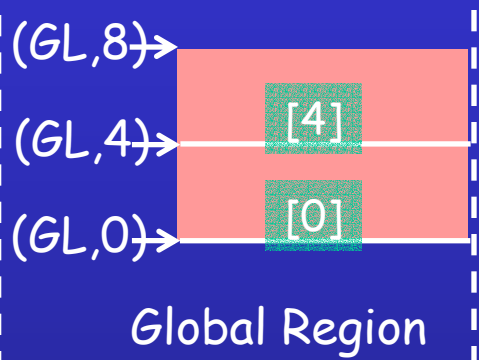

```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```

A-loc in Object Code

- An A-loc defined as:
  - Memory locations between statically known offsets in each region
  - Registers are considered a-locs
- Finding all statically known offsets
  - Global region
    - direct addressing mode operands
    - e.g., mov [4], edx
  - Region for procedure
    - esp/ebp-relative indirect operands
    - e.g., lea edx, [esp+8]
  - Heap region
    - only starting offset known (0)

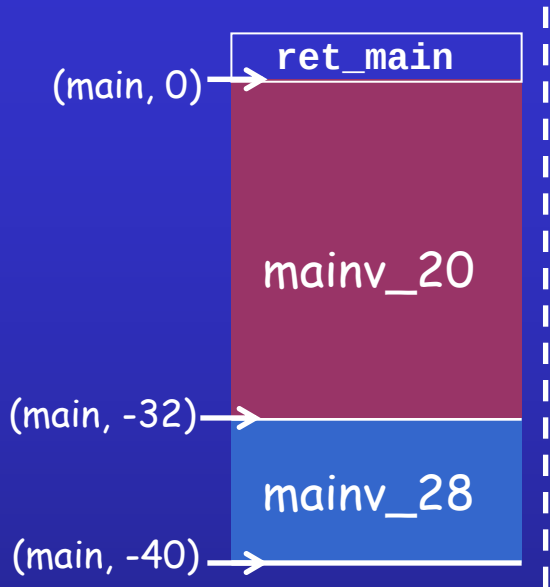
Example - A-locs



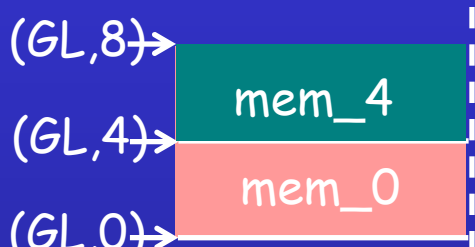



```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]         ;
mov     eax, [edi]       ;return *pArray2
add     esp, 40
retn
```

Example - A-locs



Region for main



Global Region

```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]         ;
mov     eax, [edi]       ;return *pArray2
add     esp, 40
retn
```

Value-set Analysis

- Combined pointer and numeric analysis
- Flow-sensitive
- Context-insensitive
- Abstract interpretation to determine
  - possible addresses and values in a-locs at each program point
  - Abstract Domain – Value-set

Value-set

- A concise representation
  - for a set of addresses and values
- A set of addresses in memory-region
  - (rgn, $\{o_1, o_2, \ldots, o_n\}$)
- $\{o_1, o_2, \ldots, o_n\}$
  - set of numbers
  - ∴ use numerical domains
  - e.g., {1, 3, 5, 7} – [1,7] in interval domain
- Reduced Interval Congruences(RIC)

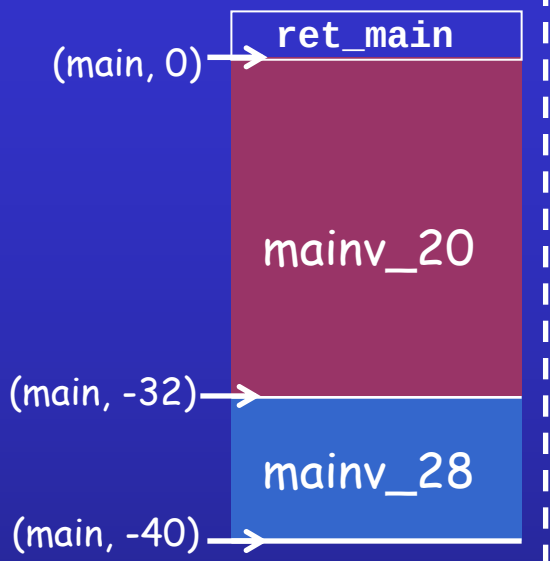
Reduced Interval Congruence (RIC)

- Approximates set of numbers by:
  - common stride
  - lower and upper bounds
  - offset from 0
- {1, 3, 5, 7} represented as $2[0,3]+1$
- Formally,
  - Interval Domain $\cap$ Congruence Domain
  - $\{1, 3, 5, 7\} \Leftrightarrow [1,7] \cap 2\mathbb{Z}+1$
- Approximation is always safe
  - {Approximate set} $\supseteq$ {Original Set}

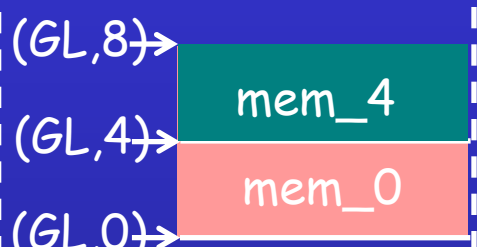
Value-set

- Set of addresses
  - $\{(rgn_1, ric_1), (rgn_2, ric_2), \ldots, (rgn_r, ric_r)\}$
  - "r" – number of regions in program
- Also represented as an r-tuple
  - $(ric_1, ric_2, \ldots, ric_r)$
  - ric_1 – always offsets in global region
- Global addresses also numbers
  - Set of numbers -- $(ric_1, \perp, \ldots, \perp)$
- Such r-tuples are called value-sets

Example - Value-set analysis



Region for main



Global Region

1

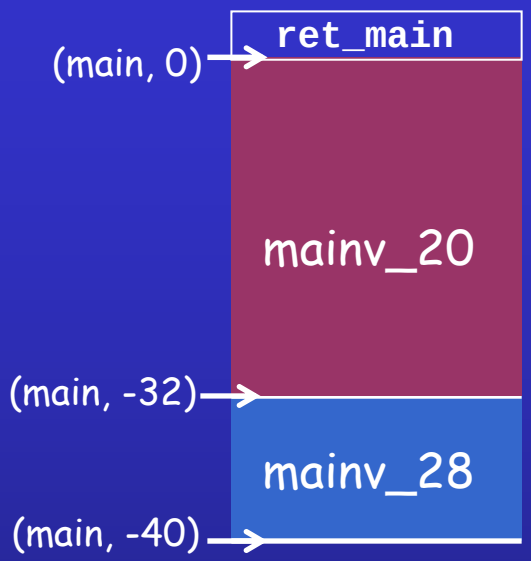
ecx → (⊥, 4[0,∞]-40)
ebx → (1[0,9], ⊥)
esp → (⊥, -40)

2

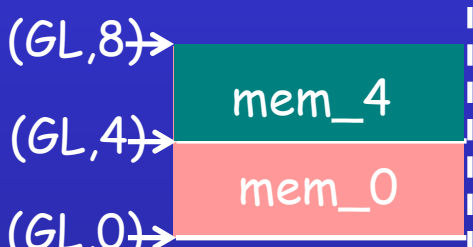
edi → (⊥, -32)
esp → (⊥, -40)

```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray
add     esp, 40
retn
```

Example - Value-set analysis



Region for main



Global Region

```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
    mov     [ecx], edx   ;*p=arrVal
    add     ecx, 4       ;p++;
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray
add     esp, 40
retn
```

1

ecx → (⊥, 4[0,∞]-40)

⇒[ecx] c-kills {mainv_28, mainv_20, ret_main}

2

edi → (⊥, -32)

⇒ [edi] accesses {mainv_20}

Affine-relations Analysis

- Value-set domain is non-relational
  - Cannot capture relationships among a-locs
- Imprecise results
  - e.g. no upper bound for **ecx** at loc_9
    - **ecx** $\rightarrow (\perp, 4[0,\infty]-40)$

```
……

loc_9:
   mov    [ecx], edx   ;*p=arrVal
   add    ecx, 4       ;p++;
   inc    ebx          ;i++
   cmp    ebx, 10      ;i<10?
   jl     short loc_9 ;
……
……
```

Affine-relations Analysis

- Obtain relationships from other analyses
- Use relationships to improve precision
  - e.g., at loc_9:
    - **ecx=esp+(4×ebx), ebx=([0,9],⊥), esp=(⊥,-40)**
    - **⇒ ecx=(⊥,-40)+4([0,9])**
    - **⇒ ecx=(⊥,4[0,9]-40)**
    - Upper bound for **ecx** at loc_9

```
…

loc_9:
   mov    [ecx], edx  ;*p=arrVal
   add    ecx, 4      ;p++;
   inc    ebx         ;i++
   cmp    ebx, 10     ;i<10?
   jl     short loc_9 ;
…
…
```

Affine-relations Analysis

- We use Affine-relations analysis
- Affine-relation:
  - $r_1, r_2, \ldots, r_n$ - a-locs, $a_0, a_1, \ldots, a_n$ - integer constants
  - $a_0 + \sum_{i=1..n}(a_i r_i) = 0$
- Müller-Olm and Seidl algorithm
  - Flow-sensitive and context-sensitive
  - To determine affine relations among global variables
- Use Müller-Olm and Seidl algorithm
  - To find affine relations that hold among registers
- Use such relations to improve precision

Running Times

Program	nProc	nInsts	Value-set Analysis(s)	Affine-relations	Coverage (%)
cat(2.0.14)	123	3814	0.48	5.21	49.14
cut(2.0.14)	129	4246	0.72	18.07	56.76
grep(2.4.2)	245	16682	1.89	83.65	27.43
gcc(2.96)	252	22842	6.18	2229.09	97.05
flex(2.5.4)	239	23373	8.61	389.20	97.62

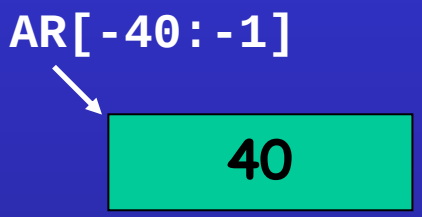
Future Work

- Safe handling of indirect calls and jumps
- Context-sensitivity
  - using call-strings approach
  - improves precision

Future Work

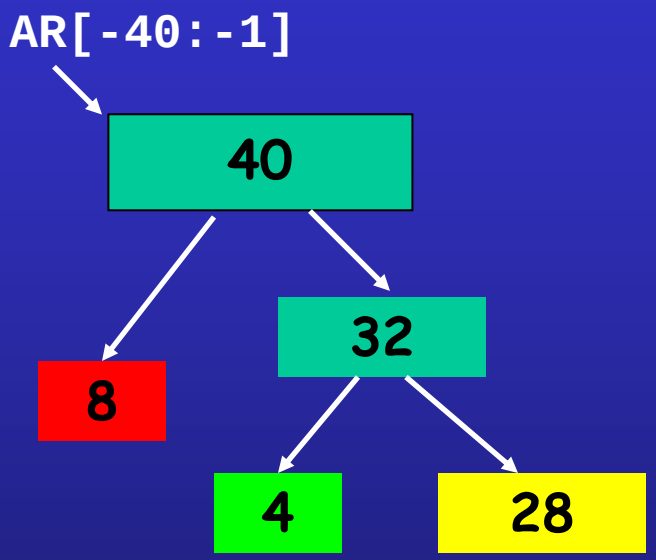
- Aggregate Structure Identification
  - Ramalingam et al. [POPL 99]
  - Ignore declarative information
  - Identify fields from the access patterns
  - Uses:
    - improve a-loc abstraction
    - discover type information

Future Work



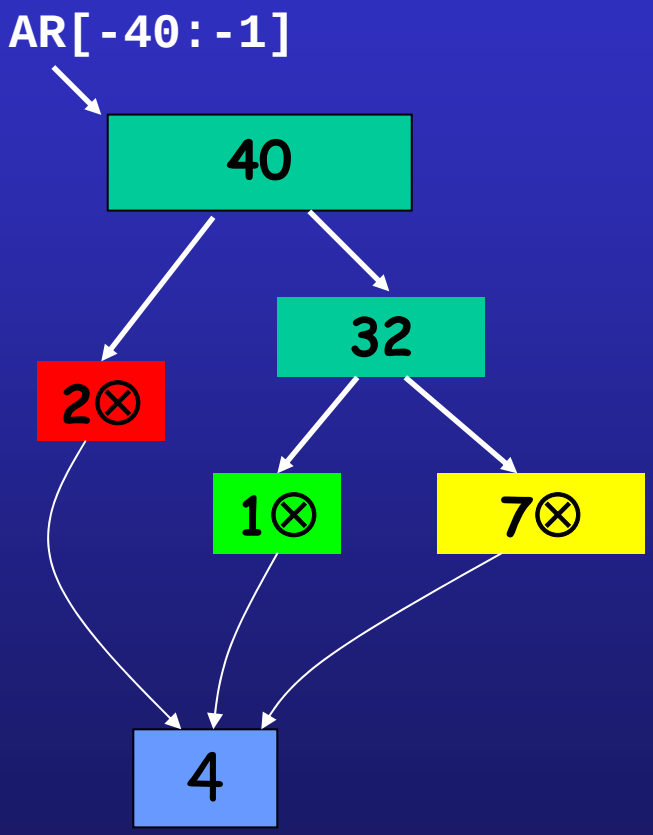

```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
   mov     [ecx], edx    ;*p=arrVal
   add     ecx, 4        ;p++;
   inc     ebx           ;i++
   cmp     ebx, 10       ;i<10?
   jl      short loc_9   ;

mov     edi, [4]      ;
mov     eax, [edi]    ;return *pArray2
add     esp, 40
retn
```

Future Work




```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40         ;adjust stack
lea     edx, [esp+8]    ;
mov     [4], edx        ;pArray2=&a[2]
lea     ecx, [esp]      ;p=&a[0]
mov     edx, [0]        ;

loc_9:
   mov     [ecx], edx   ;*p=arrVal
   add     ecx, 4       ;p++;
   inc     ebx          ;i++
   cmp     ebx, 10      ;i<10?
   jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```

Future Work




```
; ebx ⇔ i
; ecx ⇔ variable p

sub     esp, 40          ;adjust stack
lea     edx, [esp+8]     ;
mov     [4], edx         ;pArray2=&a[2]
lea     ecx, [esp]       ;p=&a[0]
mov     edx, [0]         ;

loc_9:
   mov     [ecx], edx    ;*p=arrVal
   add     ecx, 4        ;p++;
   inc     ebx           ;i++
   cmp     ebx, 10       ;i<10?
   jl      short loc_9   ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```