Analyzing Memory Accesses in x86 Executables

Gogul Balakrishnan
Thomas Reps
University of Wisconsin

Motivation

- Basic infrastructure for language-based security
  - buffer-overrun detection
  - information-flow vulnerabilities
  - . . .
- What if we do not have source code?
  - viruses, worms, mobile code, etc.
  - legacy code (w/o source)
- Limitations of existing tools
  - over-conservative treatment of memory accesses
    $\Rightarrow$ Many false positives
  - unsafe treatment of pointer arithmetic
    $\Rightarrow$ Many false negatives

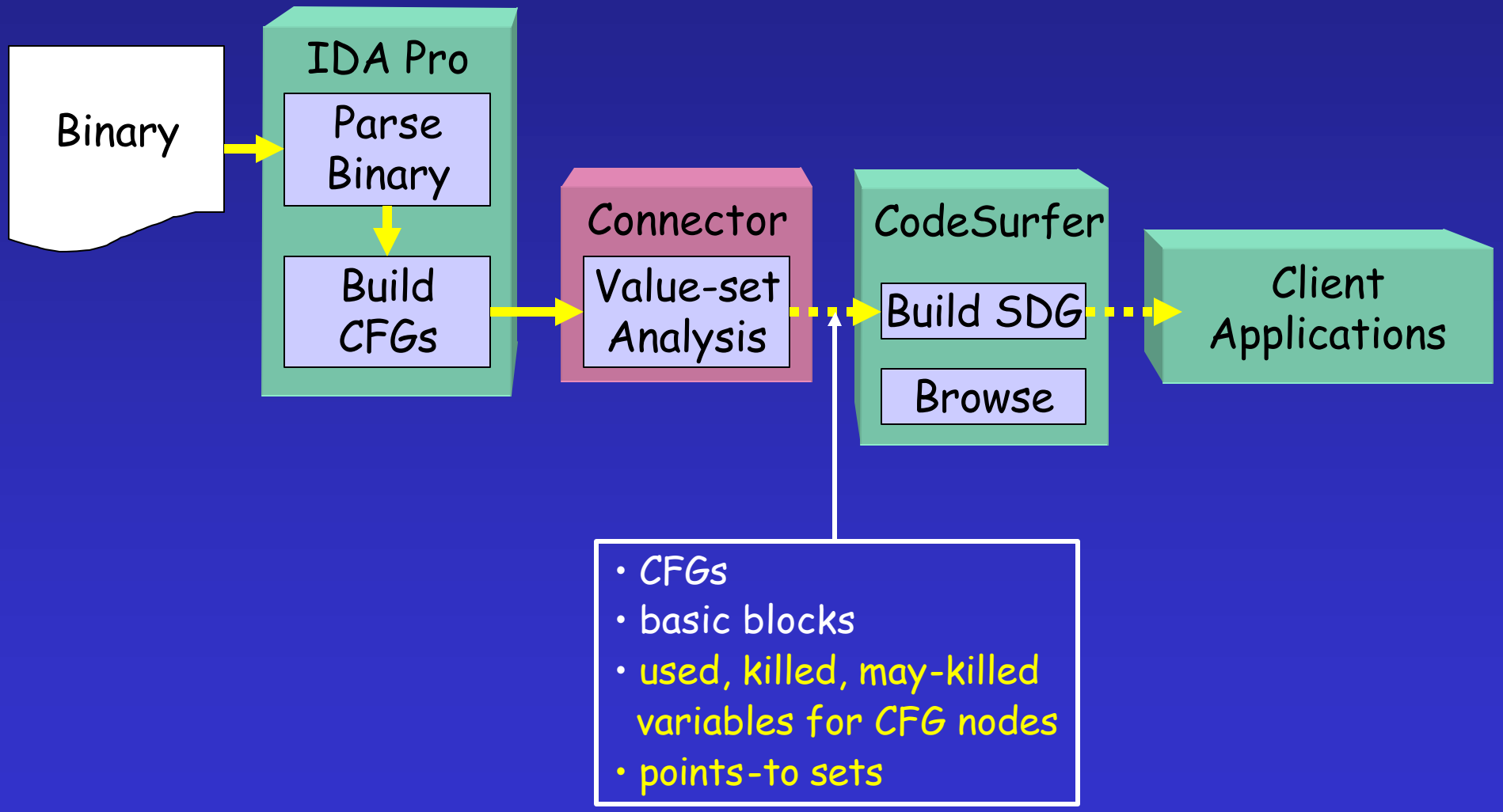
Goal (1)

- Create an intermediate representation (IR) that is similar to the IR used in a compiler
  - CFGs
  - used, killed, may-killed variables for CFG nodes
  - points-to sets
  - call-graph
- Why?
  - a tool for a security analyst
  - a general infrastructure for binary analysis

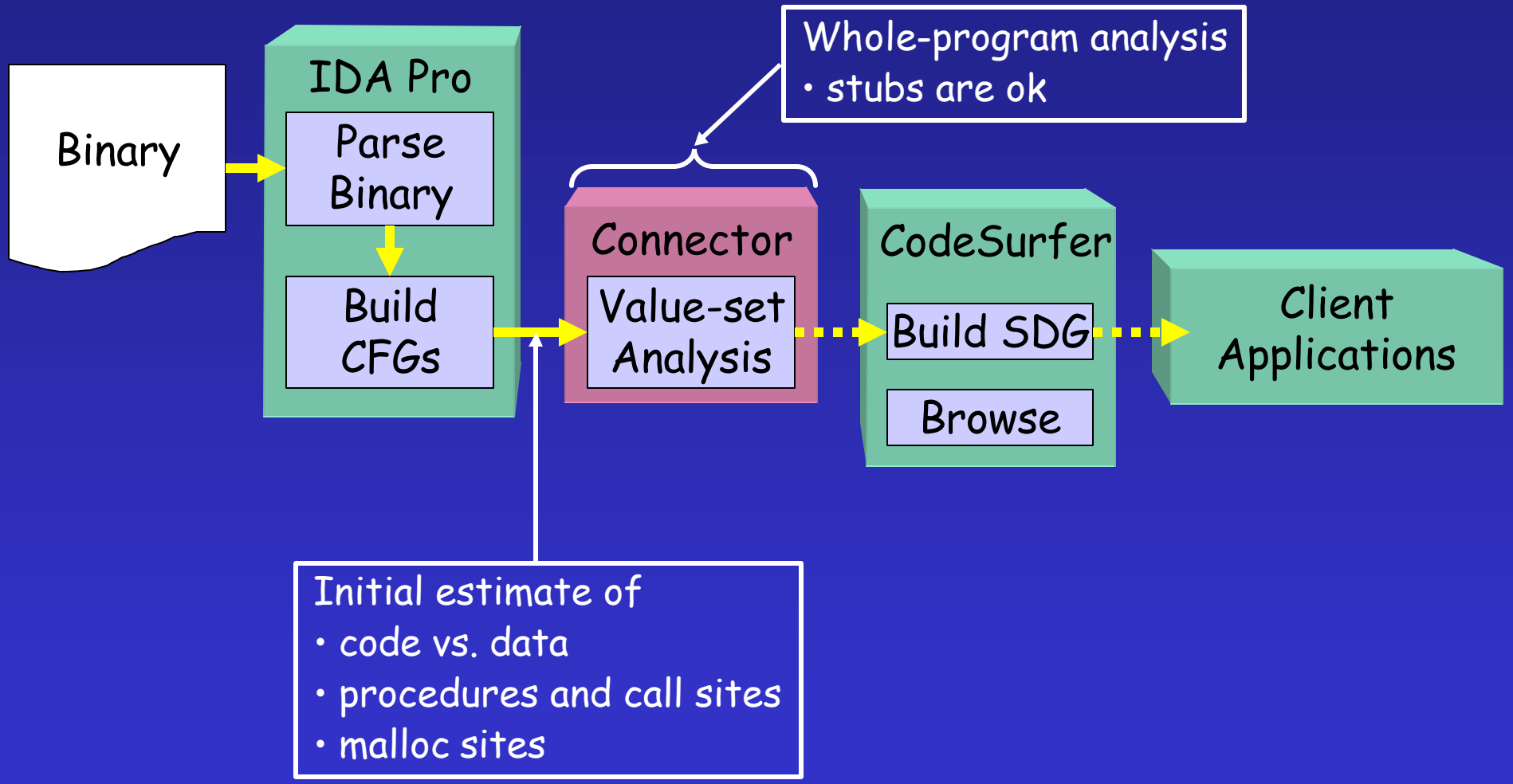
Goal (2)

- Scope: programs that conform to a "standard compilation model"
  - data layout determined by compiler
  - some variables held in registers
  - global variables → absolute addresses
  - local variables → offsets in `esp`-based stack frame
- Report violations
  - violations of stack protocol
  - return address modified within procedure

Codesurfer/x86 Architecture



Codesurfer/x86 Architecture



Outline

- Example
- Challenges
- Value-set analysis
- Performance
- [Future work]

Running Example

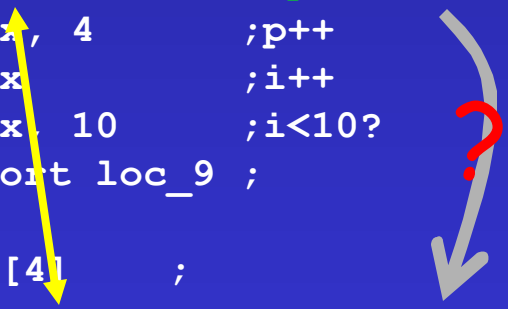
```
int arrVal=0, *pArray2;

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 Û i
; ecx Û 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
```

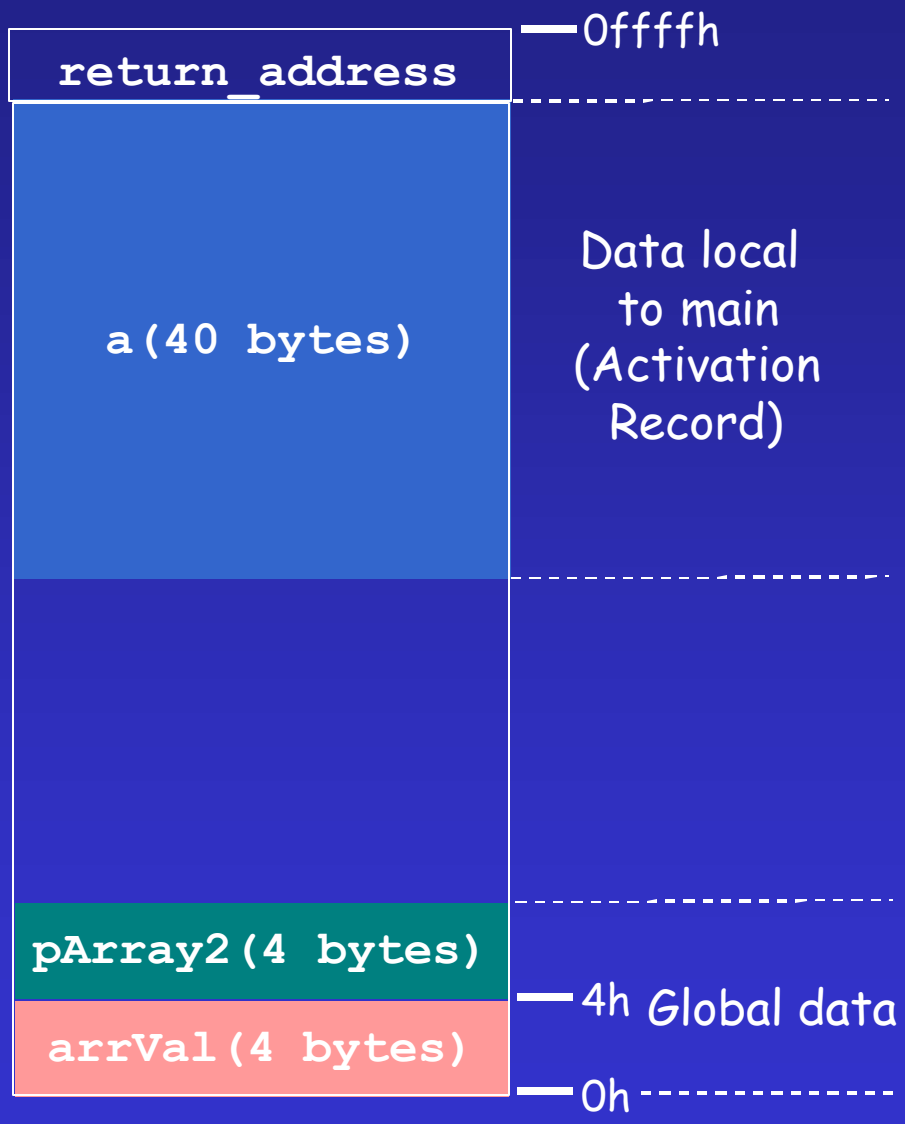
Running Example

```
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 Û i
; ecx Û 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
```

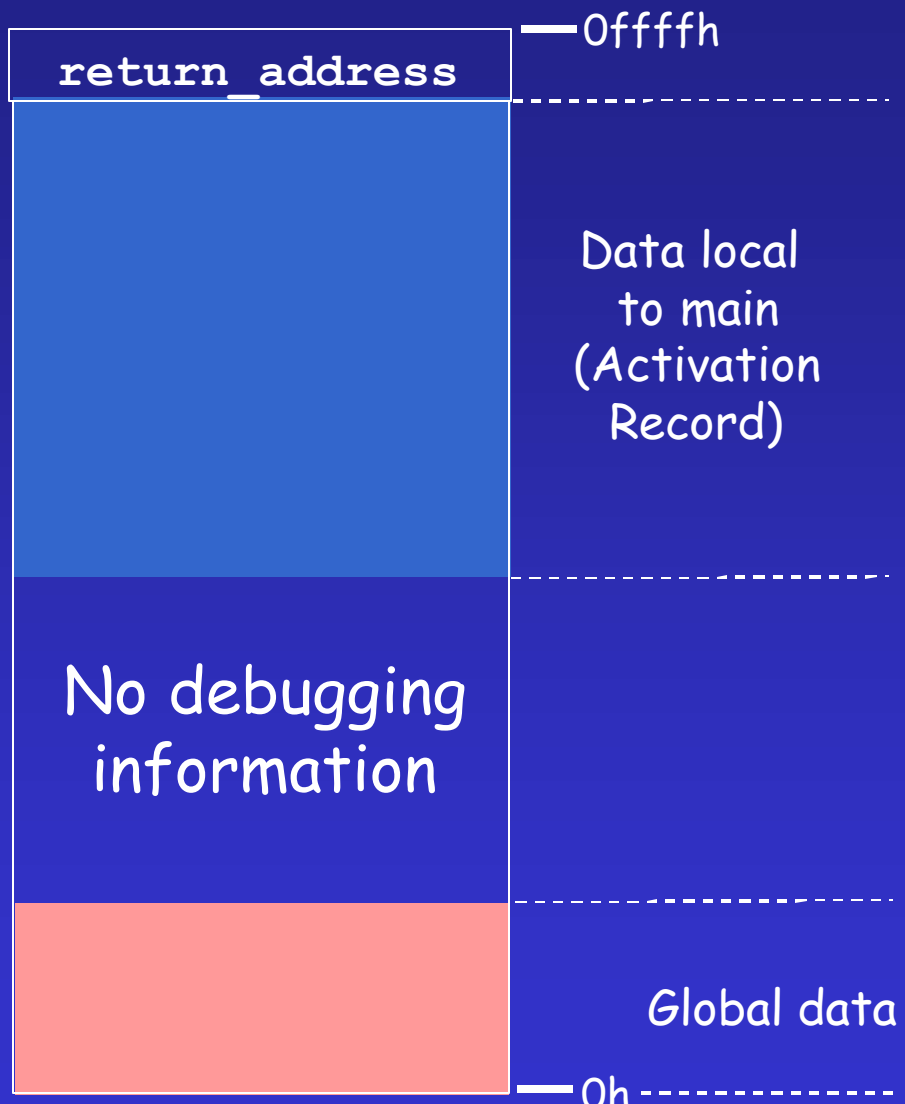


Running Example - Address Space




```
; ebx Û i
; ecx Û 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 Û i
; ecx Û 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
```

Challenges (1)

- No debugging/symbol-table information
- Explicit memory addresses
  - need something similar to C variables
  - a-locs
- Only have an initial estimate of
  - code, data, procedures, call sites, malloc sites
  - extend IR on-the-fly
    - disassemble data, add to CFG, . . .
    - similar to elaboration of CFG/call-graph in a compiler because of calls via function pointers

Challenges (2)

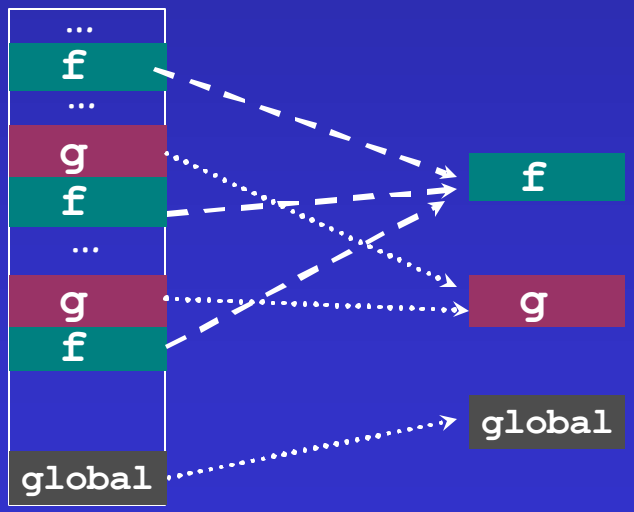
- Indirect-addressing mode
  - need "pointer analysis"
  - value-set analysis
- Pointer arithmetic
  - need numeric analysis (e.g., range analysis)
  - value-set analysis
- Checking for non-aligned accesses
  - pointer forging?
  - keep stride information in value-sets

Not Everything is Bad News !

- Multiple source languages OK
- Some optimizations make our task easier
  - optimizers try to use registers, not memory
  - deciphering memory operations is the hard part

Memory-regions

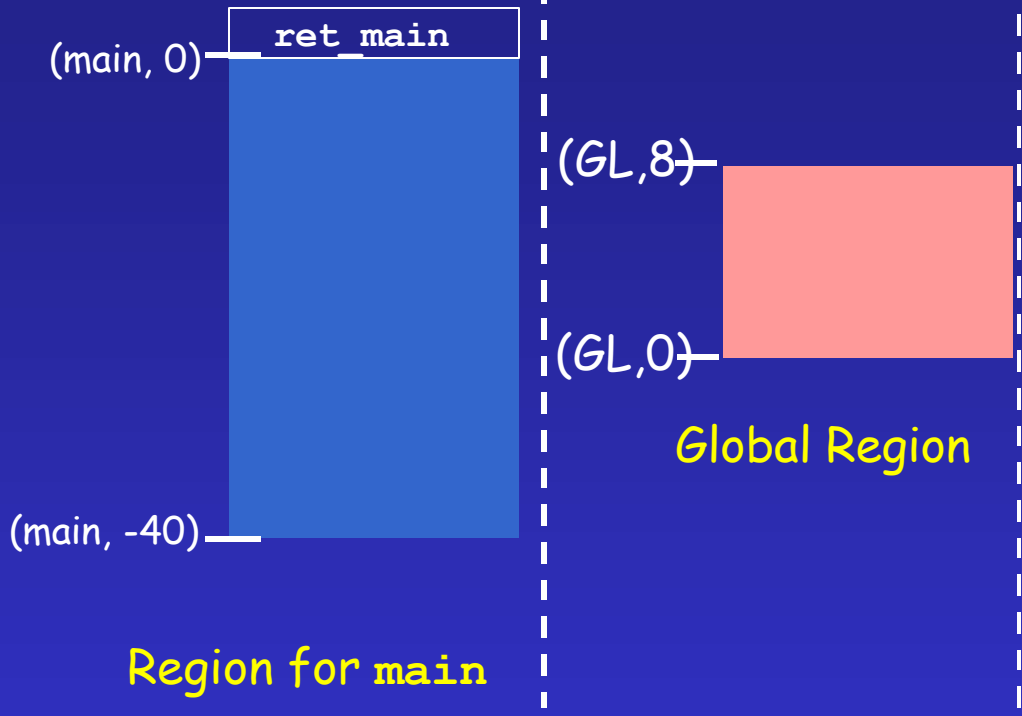
- An abstraction of the address space
- Idea: group similar runtime addresses
  - collapse the runtime ARs for each procedure



Memory-regions

- An abstraction of the address space
- Idea: group similar runtime addresses
  - collapse the runtime ARs for each procedure
- Similarly,
    - one region for all global data
    - one region for each malloc site

Example - Memory-regions

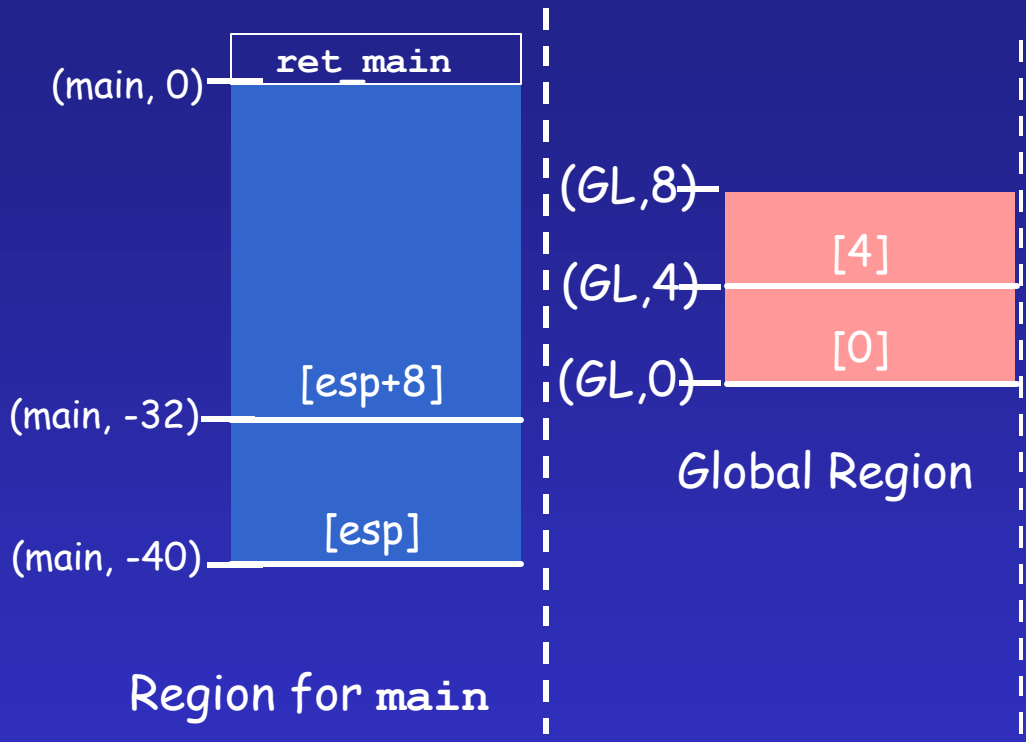



```
; ebx Û i
; ecx Û variable p

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
```

"Need Something Similar to C Variables"

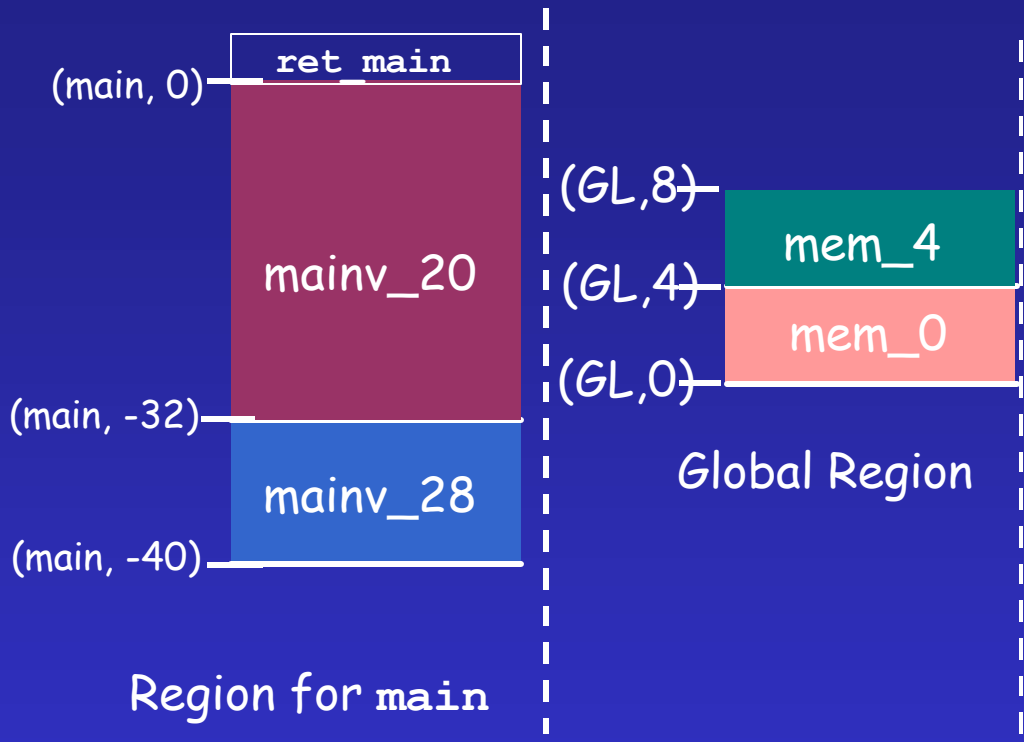
- Standard compilation model
  - some variables held in registers
  - global variables → absolute addresses
  - local variables → offsets in stack frame
- *A*-locs
  - locations between consecutive addresses
  - locations between consecutive offsets
  - registers
- Use a-locs instead of variables in static analysis
  - e.g., killed a-loc ≈ killed variable

Example - A-locs



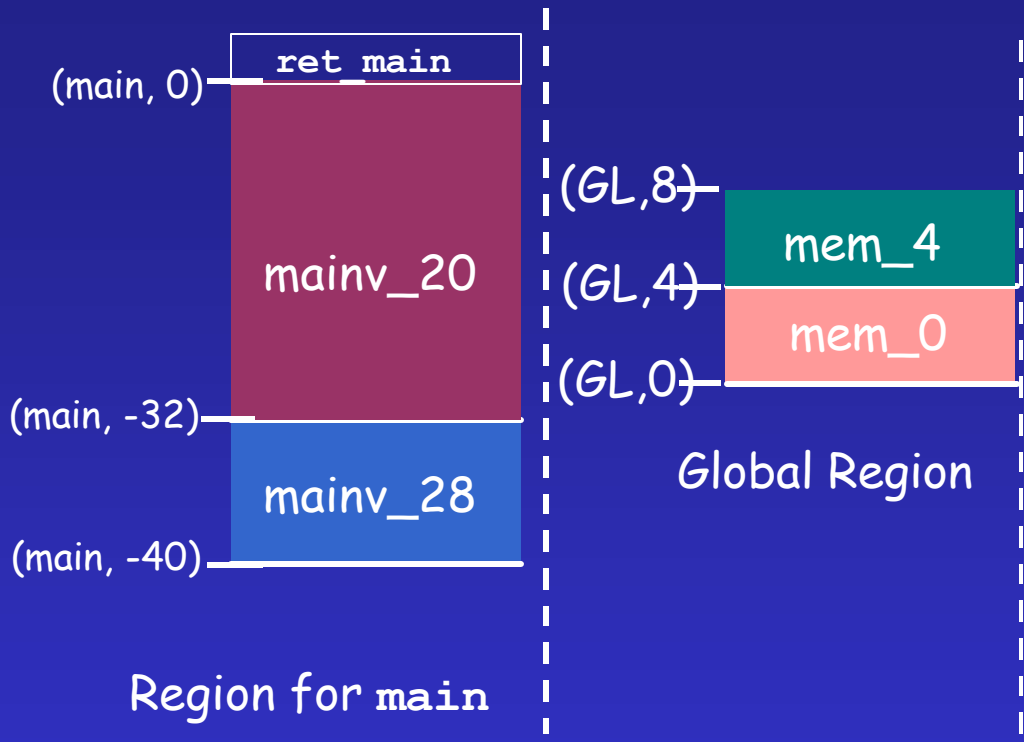

```
; ebx Û i
; ecx Û variable p

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




```
; ebx Û i
; ecx Û 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
```

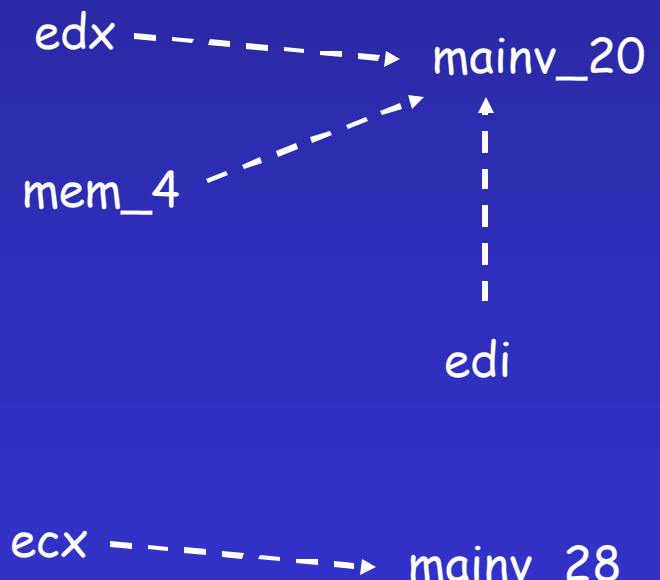
Example - A-locs




```
; ebx Û i
; ecx Û variable p

sub     esp, 40          ;adjust stack
lea     edx, &mainv_2    ;
mov     mem_4, edx       ;pArray2=&a[2]
lea     ecx, &mainv_2    ;p=&a[0]
mov     edx, mem_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, mem_4       ;
mov     eax, [edi]       ;return *pArray2
add     esp, 40
retn
```

Example - A-locs

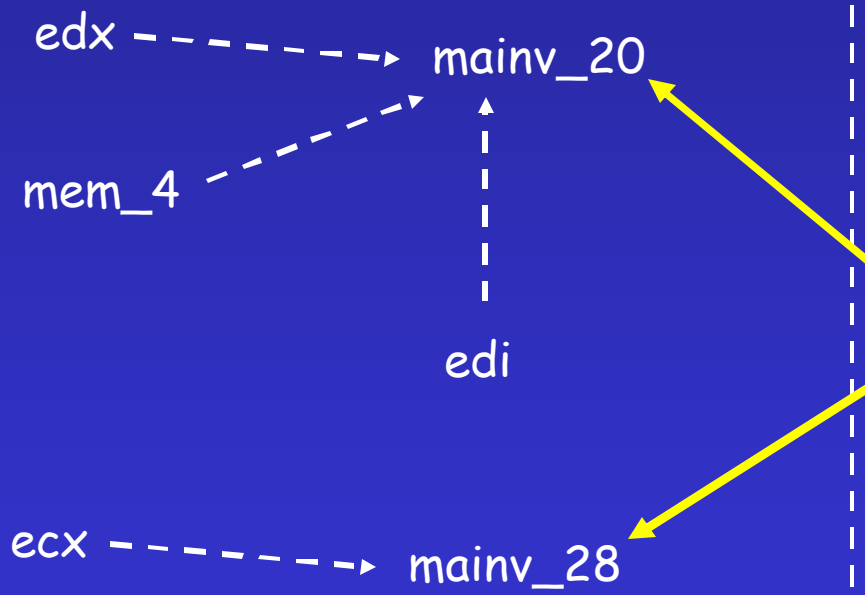
locals: mainv_28, mainv_20
{a[0], a[2]}
globals: mem_0, mem_4
{arrVal, pArray2}




```
; ebx Û i
; ecx Û variable p

sub     esp, 40         ;adjust stack
lea     edx, &mainv_20;
mov     mem_4, edx      ;pArray2=&a[2]
lea     ecx, &mainv_28;p=&a[0]
mov     edx, mem_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, mem_4      ;
mov     eax, [edi]  ;return *pArray2
add     esp, 40
retn
```

Example – A-locs

locals: mainv_28, mainv_20
{a[0], a[2]}
globals: mem_0, mem_4
{arrVal, pArray2}




```
; ebx Û i
; ecx Û variable p

sub     esp, 40         ;adjust stack
lea     edx, &mainv_20;
mov     mem_4, edx      ;pArray2=&a[2]
lea     ecx, &mainv_28;p=&a[0]
mov     edx, mem_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, mem_4      ;
mov     eax, [edi]   ;return *pArray2
add     esp, 40
retn
```

Value-Set Analysis

- Resembles a pointer-analysis algorithm
  - interprets pointer-manipulation operations
  - pointer arithmetic, too
- Resembles a numeric-analysis algorithm
  - over-approximate the set of values/addresses held by an a-loc
    - range information
    - stride information
  - interprets arithmetic operations on sets of values/addresses

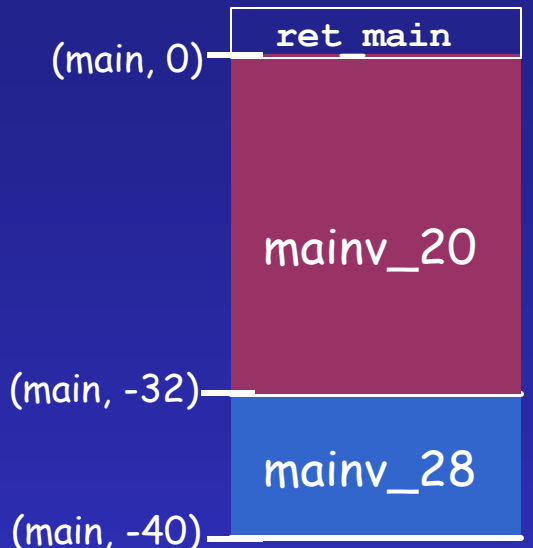
Value-set

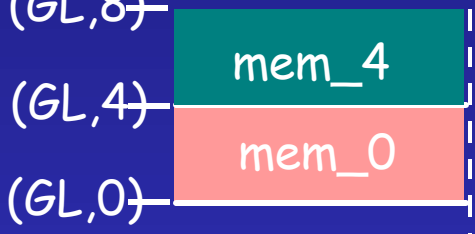
- An a-loc ≈ a variable
  - the address of an a-loc
    (memory-region, offset within the region)
- An a-loc ≈ an aggregate variable
  - addresses of elements of an a-loc
    (rgn, $\{o_1, o_2, \ldots, o_n\}$)
- Value-set = a set of such addresses
  $\{(rgn_1, \{o_1, o_2, \ldots, o_n\}), \ldots, (rgn_r, \{o_1, o_2, \ldots, o_m\})\}$
  "r" – number of regions in the program

Value-set

- Set of addresses: $\{(rgn_1, \{o_1, \ldots, o_n\}), \ldots, (rgn_r, \{o_1, \ldots, o_m\})\}$
- Idea: approximate $\{o_1, \ldots, o_k\}$ with a numeric domain
  - {1, 3, 5, 9} represented as 2[0,4]+1
  - Reduced Interval Congruence (RIC)
    - common stride
    - lower and upper bounds
    - displacement
- Set of addresses is an r-tuple: $(ric_1, \ldots, ric_r)$
  - ric_1: offsets in global region
  - a set of numbers: $(ric_1, \wedge, \ldots, \wedge)$

Example - Value-set analysis





Global Region

Region for **main**

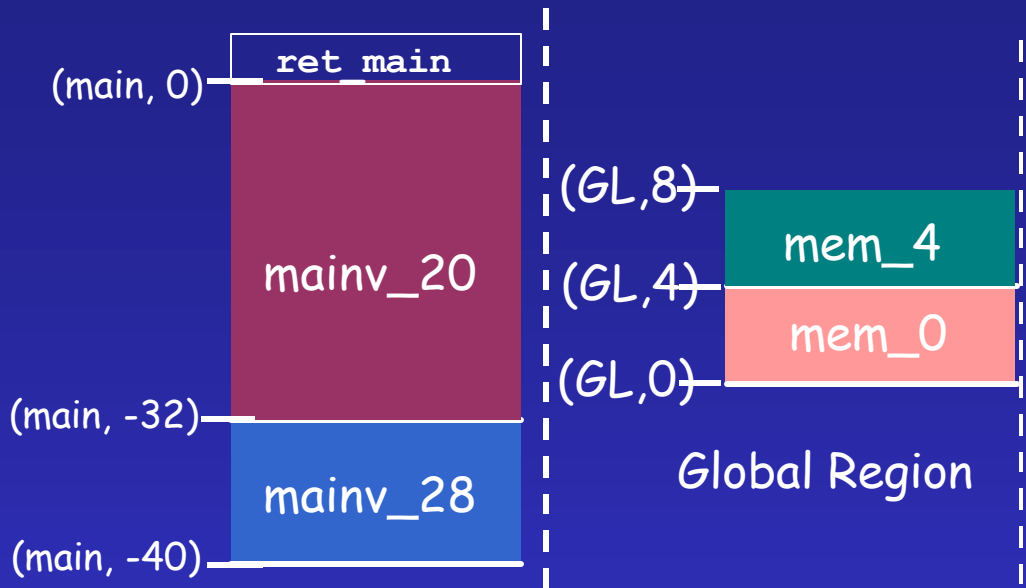
1 **ecx** → (^, 4[0,**8**]-40)
ebx → (1[0,9], ^)
esp → (^, -40)

2 **edi** → (^, -32)
esp → (^, -40)

```
; ebx Û i
; ecx Û 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   1
    add     ecx, 4      ;p++
    inc     ebx         ;i++
    cmp     ebx, 10     ;i<10?
    jl      short loc_9 ;

mov     edi, [4]    ;
mov     eax, [edi]  ;return *pArr    2
add     esp, 40
retn
```

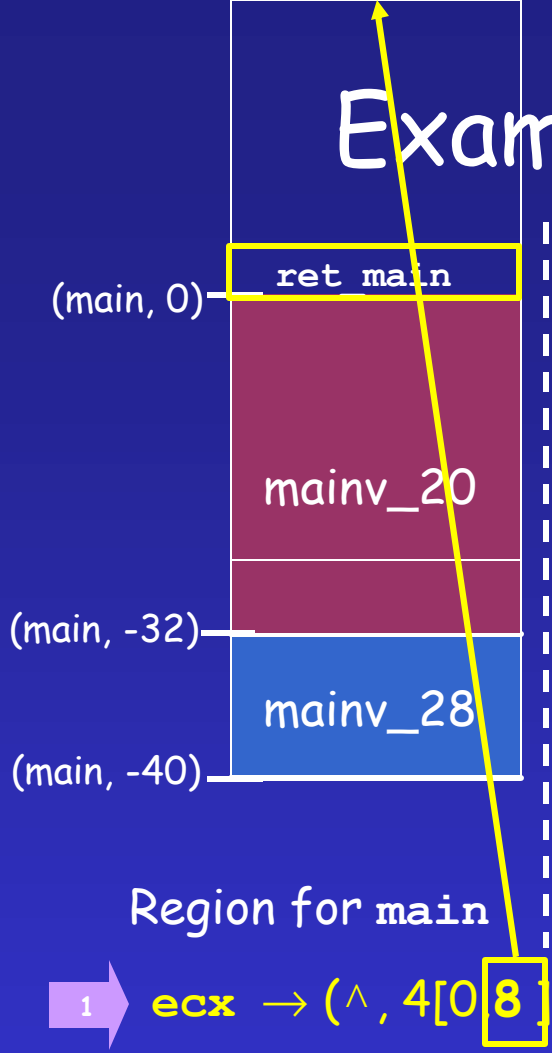
?

Example - Value-set analysis

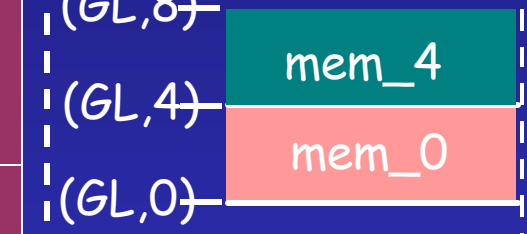



```
; ebx Û i
; ecx Û variable p

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   1
    add     ecx, 4      ;p++
    inc     ebx         ;i++
    cmp     ebx, 10     ;i<10?
    jl      short loc_9 ;

mov     edi, [4]    ;
mov     eax, [edi]  ;return *pArr    2
add     esp, 40
retn
```

Example - Value-set analysis



A stack-smashing attack?




```
; ebx Û i
; ecx Û 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   1
    add     ecx, 4       ;p++
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArr   2
add     esp, 40
retn
```

Affine-Relation 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** $\rightarrow$ ($\wedge$, 4[0,**8**]-40)

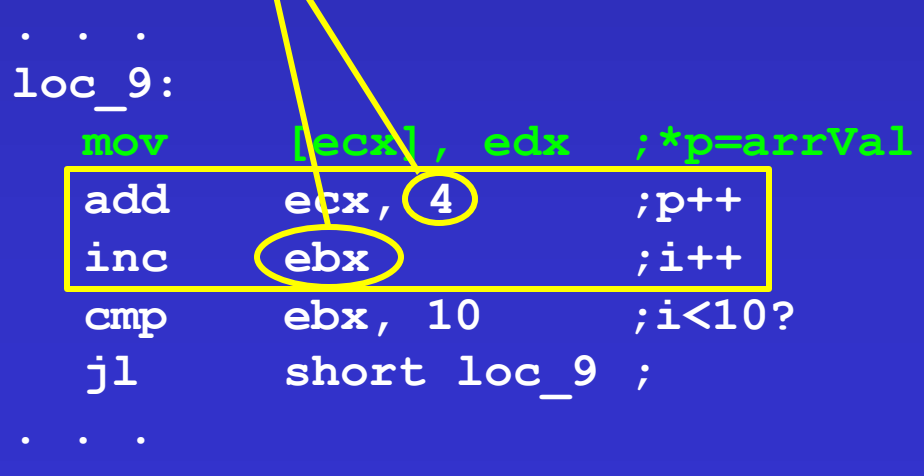
```
. . .
loc_9:
   mov     [ecx], edx   ;*p=arrVal
   add     ecx, 4       ;p++
   inc     ebx          ;i++
   cmp     ebx, 10      ;i<10?
   jl      short loc_9  ;
. . .
```

Affine-Relation Analysis

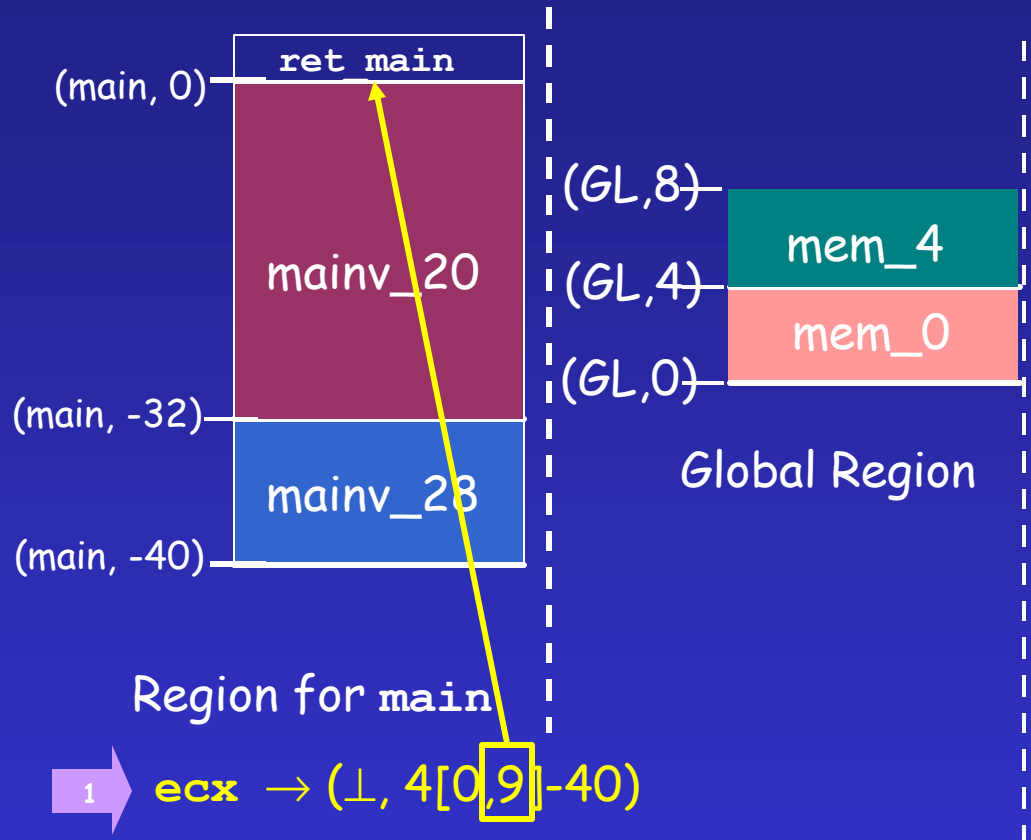
- Obtain affine relations via static analysis
- Use affine relations to improve precision
  - e.g., at loc_9

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



Example - Value-set analysis



No stack-smashing attack reported

```
; ebx Û i
; ecx Û 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   1
    add     ecx, 4       ;p++
    inc     ebx          ;i++
    cmp     ebx, 10      ;i<10?
    jl      short loc_9  ;

mov     edi, [4]     ;
mov     eax, [edi]   ;return *pArr    2
add     esp, 40
retn
```

Affine-Relation Analysis

- Affine relation
  - $x_1, x_2, \ldots, x_n$ - a-locs
  - $a_0, a_1, \ldots, a_n$ - integer constants
  - $a_0 + \sum_{i=1..n}(a_i x_i) = 0$
- Idea: determine affine relations on registers
  - use such relations to improve precision
- Implemented using WPDS++

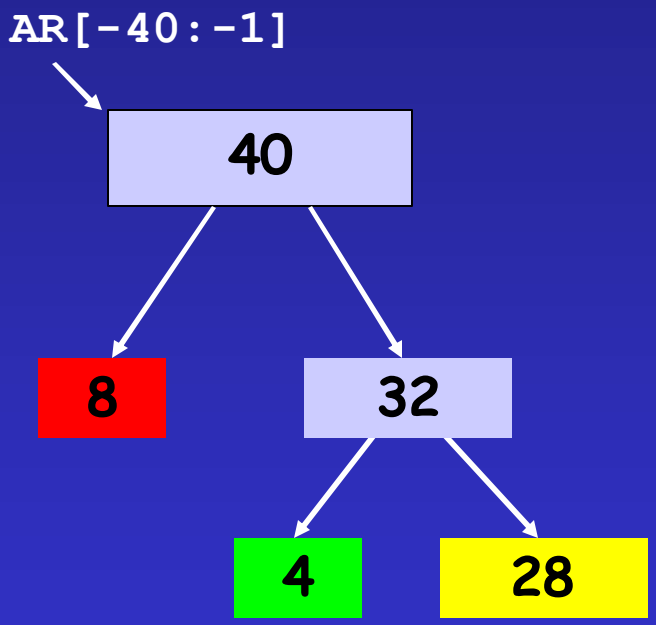
Performance

Program	nProc	nInsts	Value-set analysis (seconds)	Affine-relations (seconds)
javac	36	3,555	42	36
cat(2.0.14)	123	3,892	51	32
cut(2.0.14)	129	4,329	28	50
grep(2.4.2)	245	16,808	85	78
flex(2.5.4)	239	23,373	200	376
tar(1.13.19)	587	50,347		210
awk(3.1.0)	595	69,927		1,507
winhlp32 (5.00.2195.2014)	1,018	108,380		2,002

Future Work

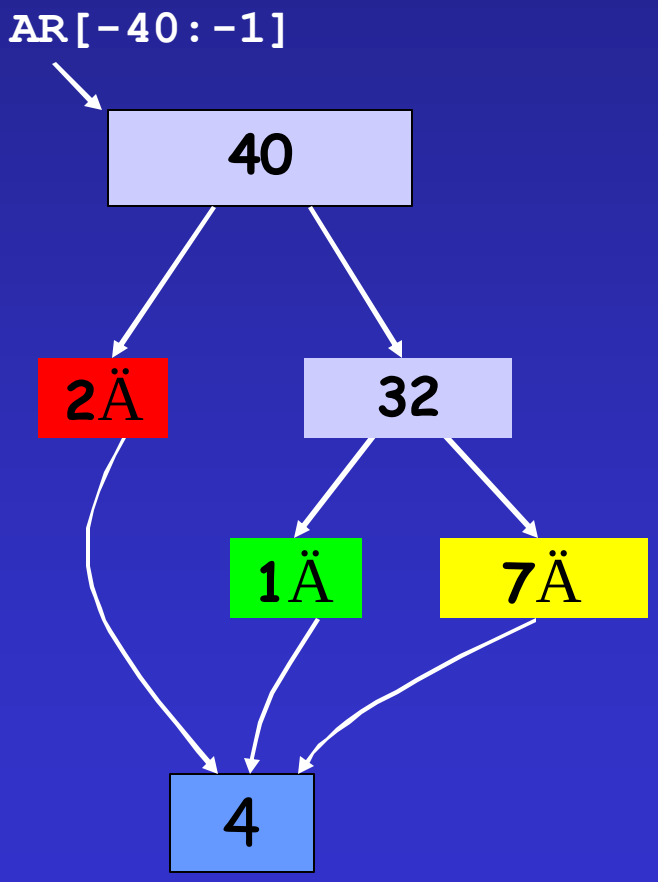
- Aggregate Structure Identification
  - Ramalingam et al. [POPL 99]
  - Ignore declarative information
  - Identify fields from the access patterns
  - Useful for
    - improving the a-loc abstraction
    - discovering type information

Future Work



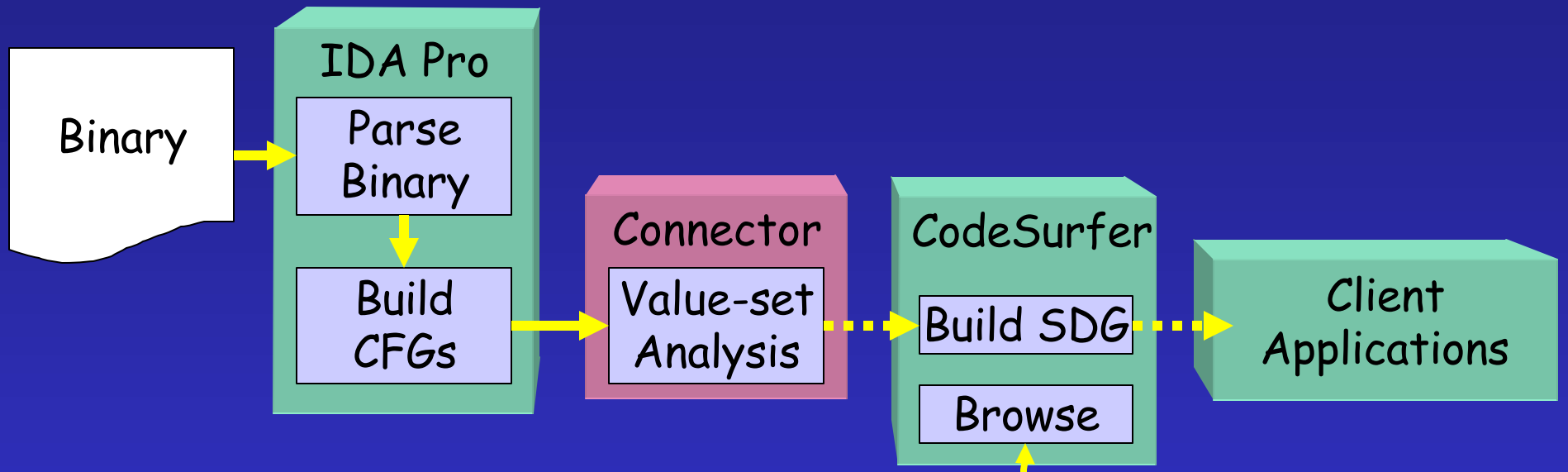

```
; ebx Û i
; ecx Û variable p

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 Û i
; ecx Û 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
```

Codesurfer/x86 Architecture



For more details

- Gogul Balakrishnan's demo
- Gogul Balakrishnan's poster
- Consult UW-TR 1486 [http://www.cs.wisc.edu/~reps/#tr1486]

Analyzing Memory Accesses in x86 Executables

Gogul Balakrishnan
Thomas Reps
University of Wisconsin