A Next-Generation Platform for Analyzing Executables

Thomas Reps

University of Wisconsin
GrammaTech, Inc.

Joint work with

Univ. of Wisconsin: G. Balakrishnan, J. Lim, A. Lal, N. Kidd

GrammaTech: T. Teitelbaum, R. Gruian. D. Melski, S. Yong, C. Chen

The Context

Larry Wagoner

"25% of IT jobs outsourced to developing countries by 2010."

Daniel Wolf, IA Director of NSA

"60–80% of all current US coding jobs are expected to be exported to China, Israel, Russia, the E.U., and India in the next decade."

Bill Scherlis (re offshore development)

"Acceptance evaluation is a huge problem."

The Vision

- Code-inspection tools for security analysts
- Analyses for identifying
  - security vulnerabilities and bugs
  - malicious code
  - commonalities and differences
- Platform for
  - code obfuscation and de-obfuscation
  - de-compilation
  - installation of protection mechanisms
  - remediation of security vulnerabilities

Why Executables?

- Reveals platform-specific choices
  - memory layout
    - padding between fields of a struct
    - which variables are adjacent?
  - register usage
  - execution order
  - optimizations performed
- Reveals other artifacts
  - compiler bugs
  - Thompson-style attack
- Allows analysis of library code

What's Wrong with Source Code?

- May not have source code
- Program may be written in multiple languages
- Must model library code with stubs
- Source-code analyses typically make unsafe assumptions (e.g., "program is ANSI-C compliant")
- Source-code constructs hide
  - memory layout
  - register usage
  - execution order (e.g., of actual parameters)
  - optimizations performed
  - compiler bugs
  - Lack of fidelity can allow vulnerabilities to escape notice

Minimizing Data Lifetime?

- Windows
  - Login process keeps a user's password in the heap after a successful login
- Should minimize data lifetime by
  - clearing memory
  - calling free()
- But . . .
  - the compiler might "optimize" away the memory-clearing code ("dead-code" elimination)

```
memset(buffer, '\0', len);
free(buffer);
```

```
free(buffer);
```

Puzzle

```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Answer: 1
(for the Microsoft compiler)

Tutorial on x86 (Intel Syntax)

```
 p = q;
 p = *q;
*p = q;
 p = &a[2];
```

Tutorial on x86 (Intel Syntax)

```
mov     ecx, edx
mov     ecx, [edx]
mov     [ecx], edx
lea     ecx, [esp+8]
```

```
ecx = edx;
ecx = *edx;
*ecx = edx;
ecx = &a[2];
```

Puzzle

```
int callee(int a, int b)
  int local;
  if (local == 5) retur
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

Standard prolog		Prolog for 1 local	
push	ebp	push	ebp
mov	ebp, esp	mov	ebp, esp
sub	esp, 4	push	ecx

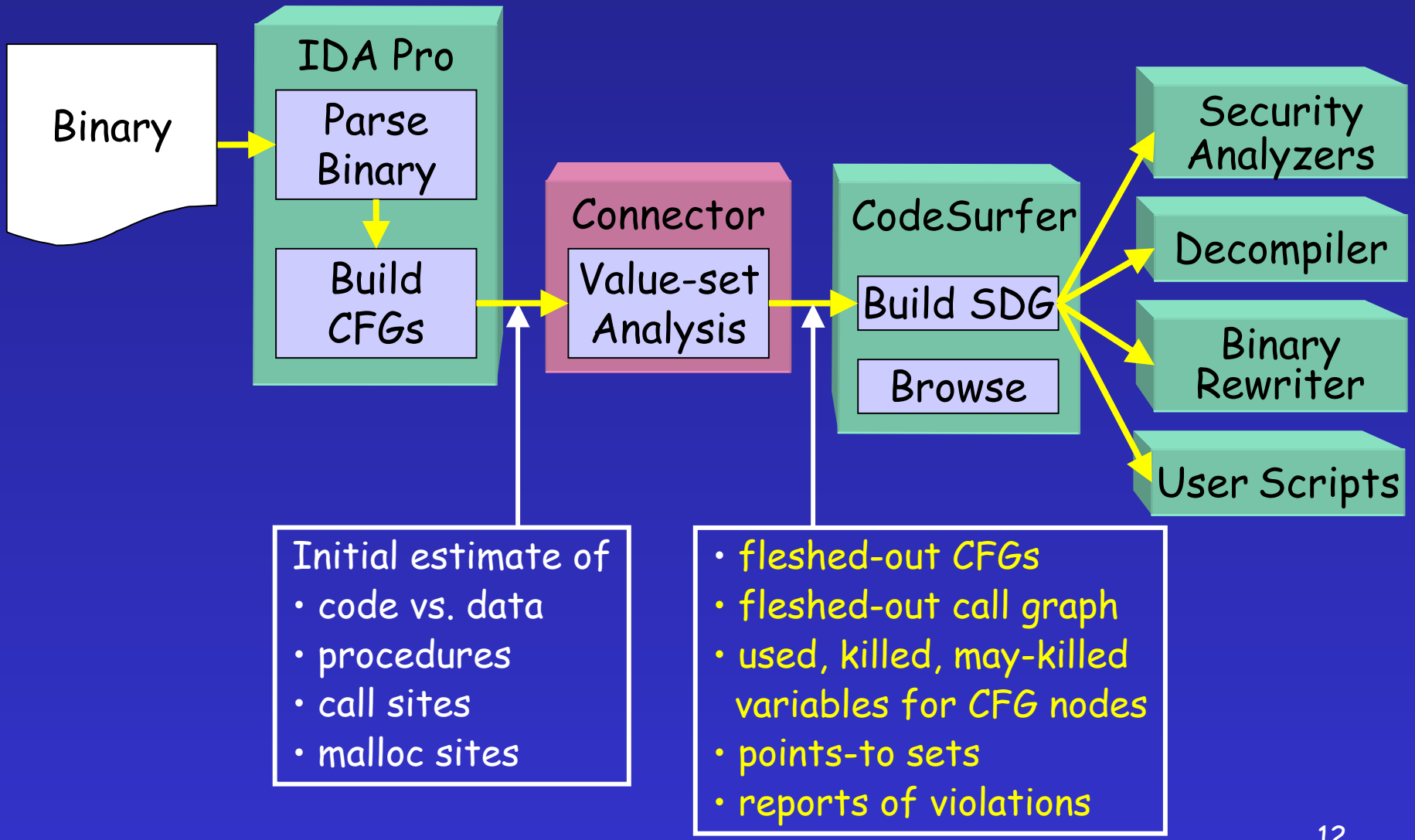
Answer: 1
(for the Microsoft compiler)

```
mov   [ebp+var_8], 5
mov   [ebp+var_C], 7
mov   eax, [ebp+var_C]
push  eax
mov   ecx, [ebp+var_8]
push  ecx
call  _callee
...
```

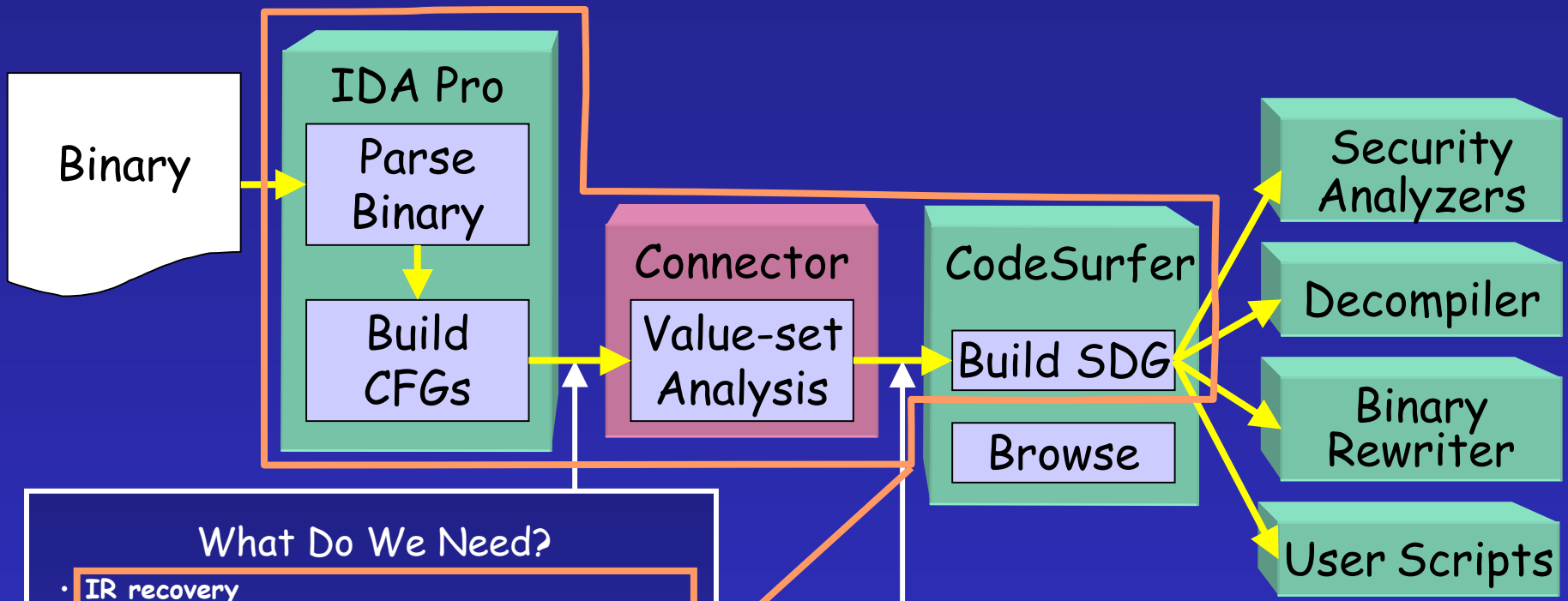
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- No use of symbol-table or debugging information!!!
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

CodeSurfer/x86 Architecture



CodeSurfer/x86 Architecture



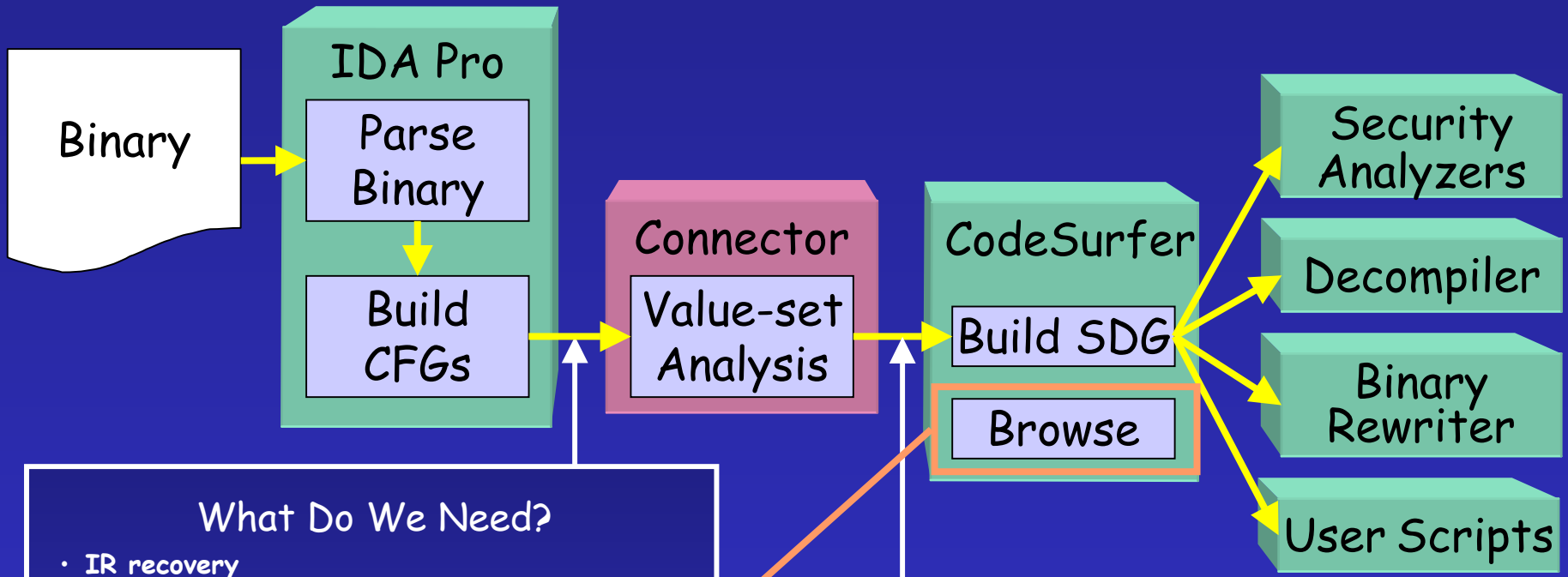
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



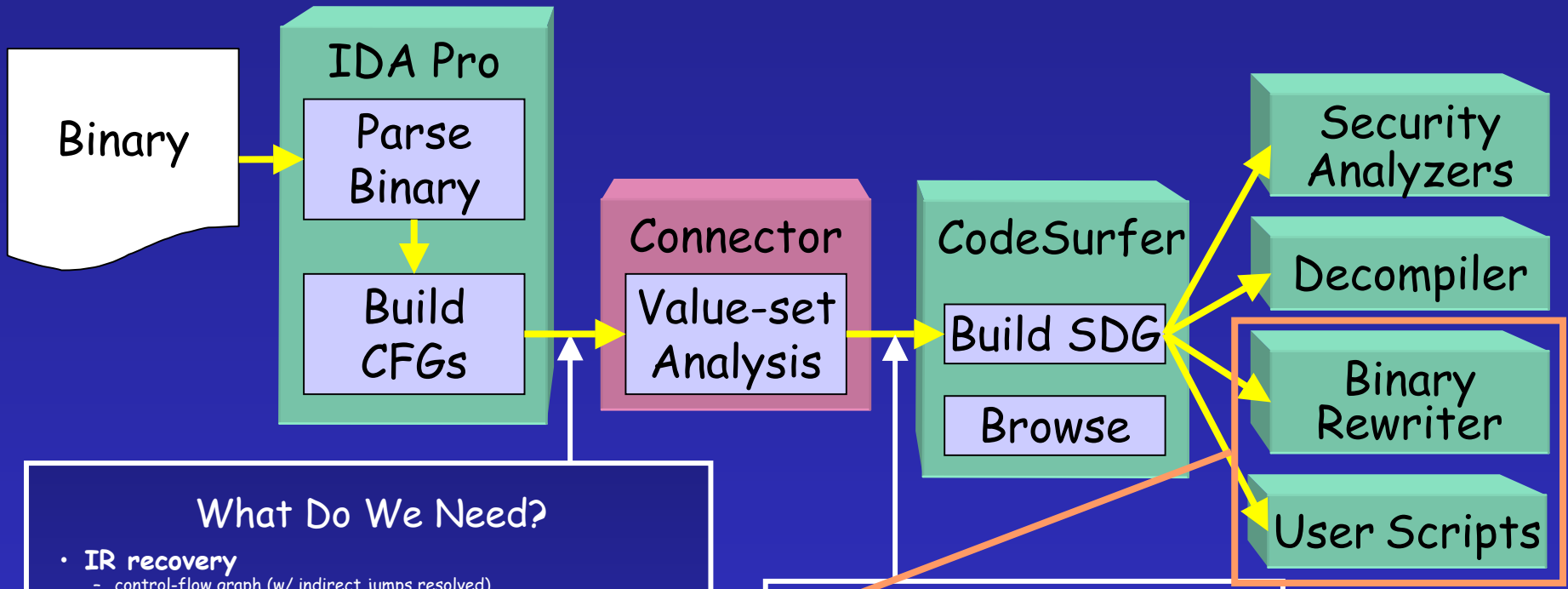
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



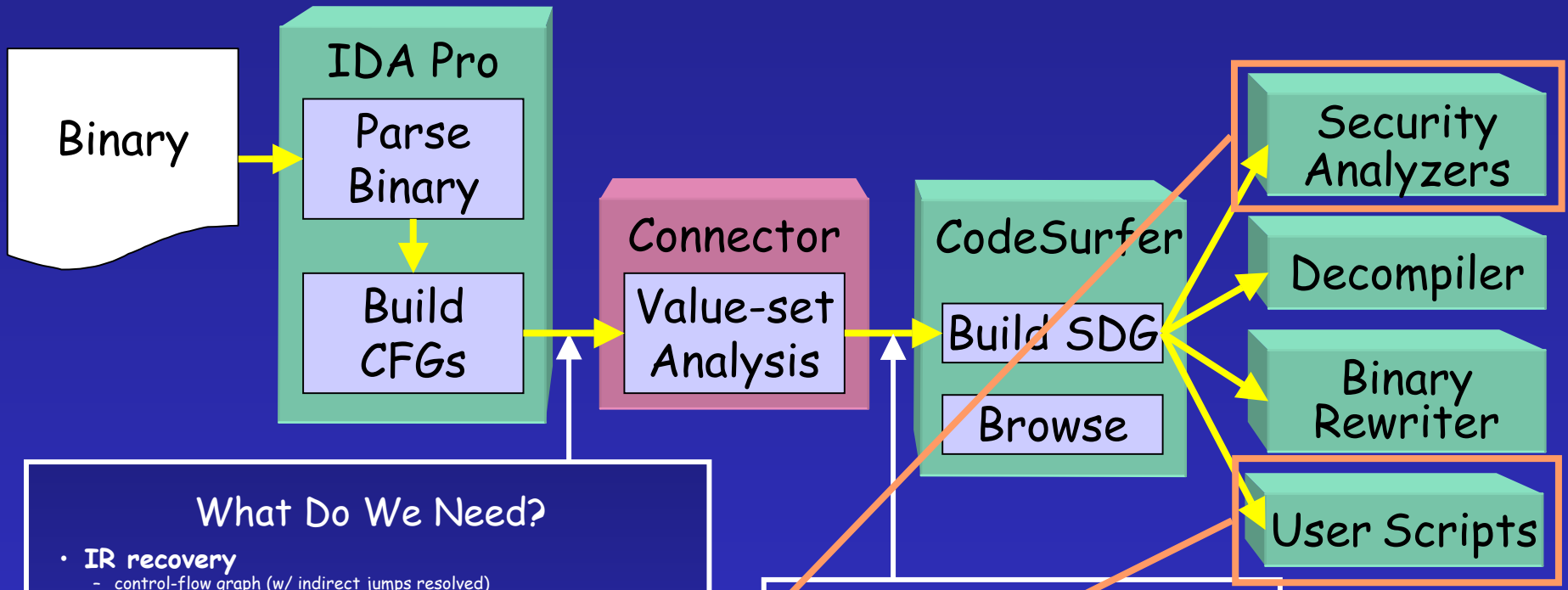
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



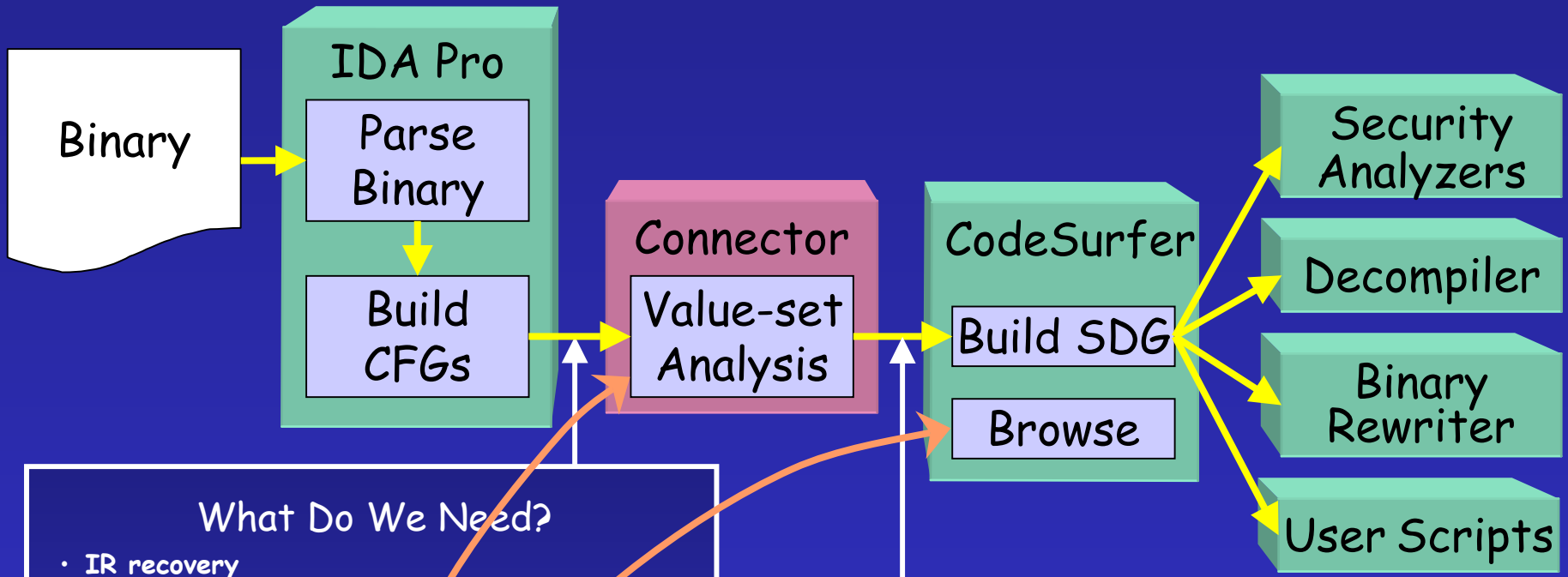
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



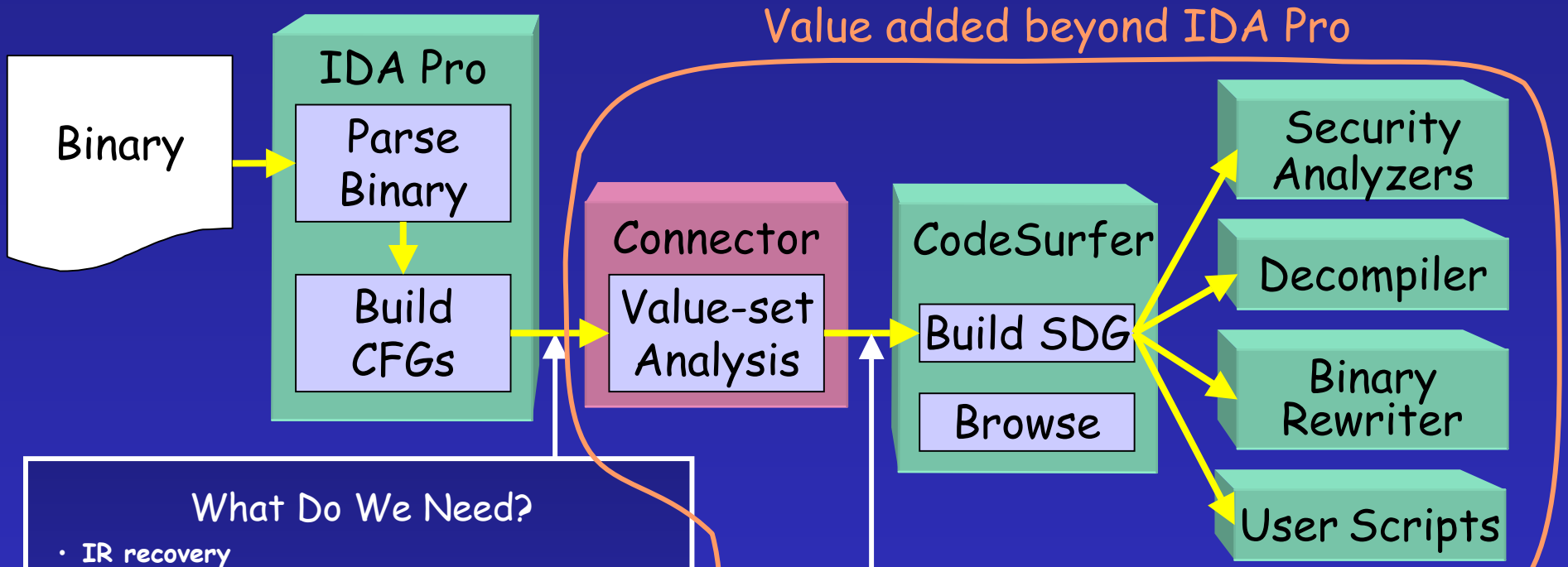
What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



What Do We Need?

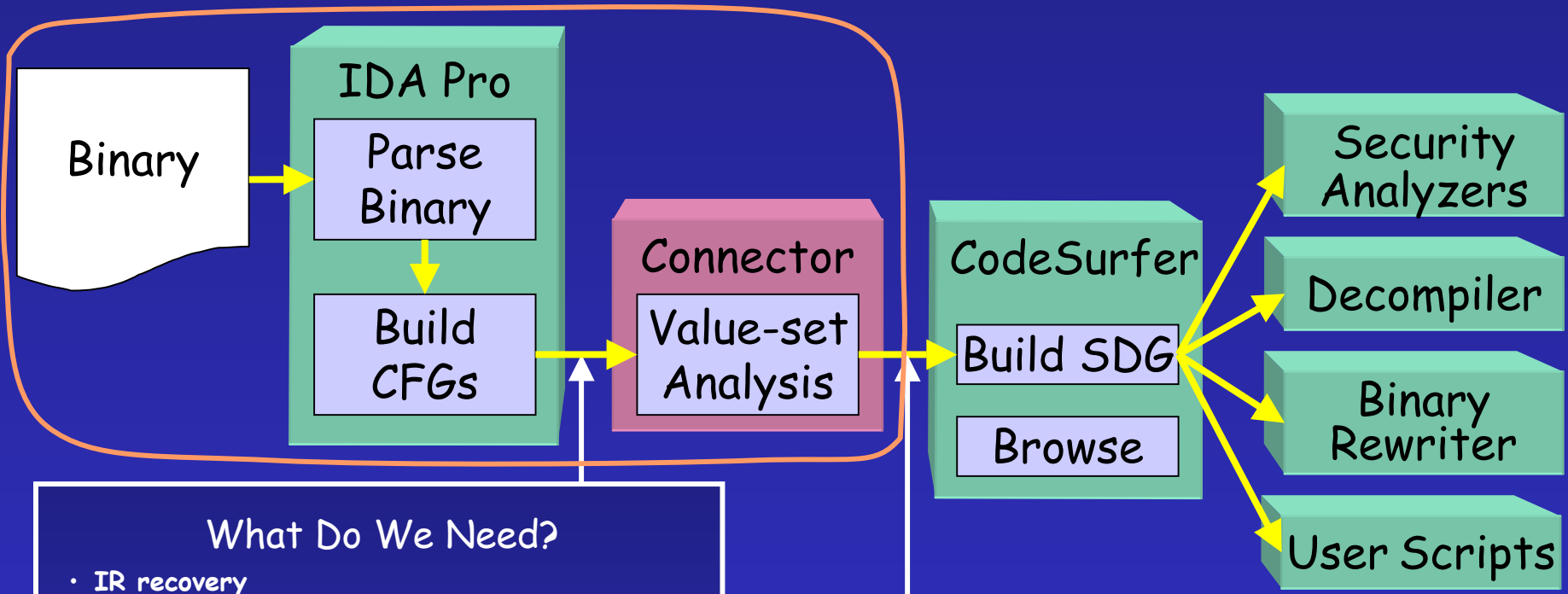
- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture

An x86 front end for CodeSurfer



What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

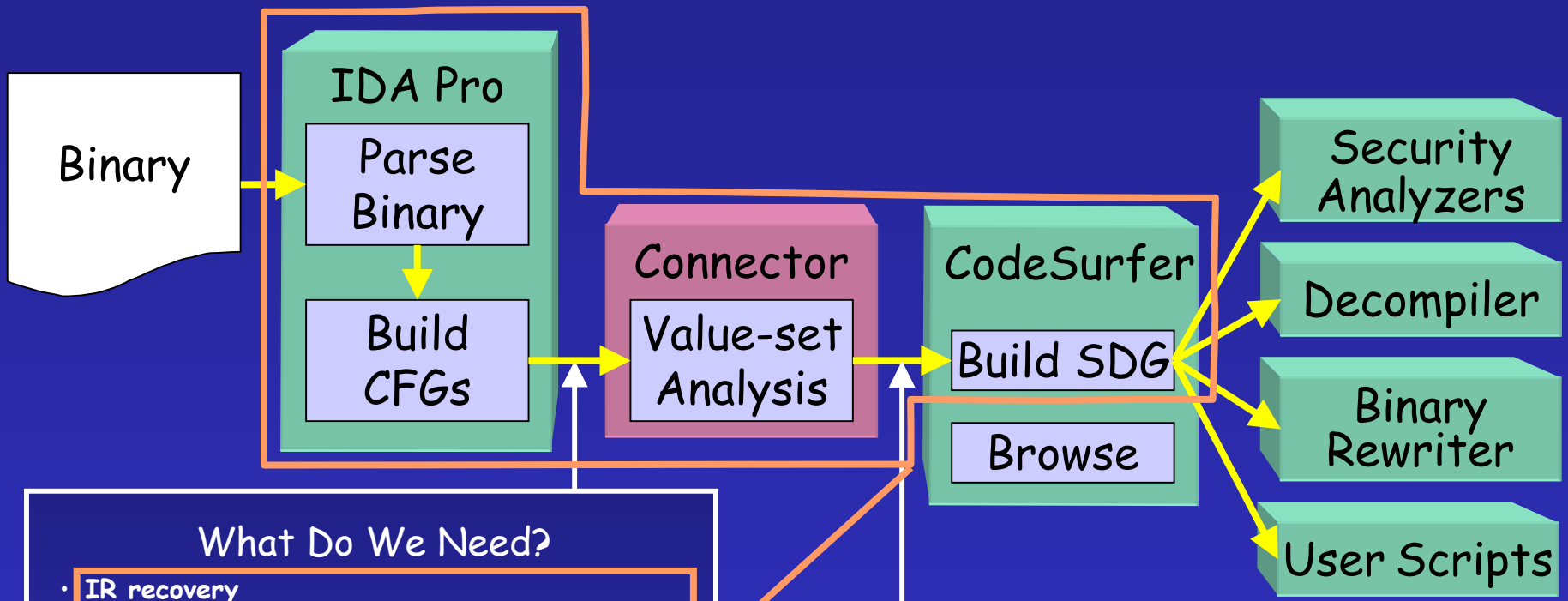
1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

An Application of CodeSurfer/x86

- Classified project at MIT Lincoln Labs
  - Adopted CodeSurfer/x86 (replacing IDA Pro)
  - DARPA funding under "Dynamic quarantine of worms"
  - PI: Rob Cunningham; PM: Anup Ghosh
- Given a worm . . .
  - What are its target-discovery, propagation, and activation mechanisms?
  - What is its payload?
- Use of CodeSurfer/x86's analysis mechanisms
  - Find system calls
  - Find their arguments
  - Follow dependences backwards to find where their values come from
  - . . .

Today's Talk



What Do We Need?

- **IR recovery**
  - control-flow graph (w/ indirect jumps resolved)
  - call graph (w/ indirect calls resolved)
  - identification of variables
  - values of pointers
  - used, killed, and possibly-killed variables for CFG nodes
  - data dependences
  - [identification of types: base types, pointer types, structs, and classes]
- **GUI for code browsing and navigation**
- **Scripting language**
  - API for accessing the IR
  - API for modifying the IR
- **IR exploration**
  - API for traversal/searching/pattern matching
  - API for defining static-analyzers/model-checkers
  - Path Explorer tool
- **Cooperation with dynamic tools**

1

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

Outline

- Challenges in analysis of executables
- Demo of CodeSurfer/x86
- Value-set analysis
- Better identification of variables
- Wrap-up

IR Recovery: Scope of our Ambitions

- Programs that conform to a "standard compilation model"
  - procedures
  - activation records
  - global data region
  - heap-
  - virtu
  - dynamica
- Report violations
  - violations of stack protocol
  - return address modified within procedure

Memory-safety violations!

Static Analysis of Executables: State of the Art

- Relies on symbol-table/debugging info
  - Atom, EEL, Vulcan, Rival
- Able to track only data movements via registers
  - EEL, Cifuentes, Debbabi, Debray
- Poor treatment of memory operations
  - Overly conservative treatment ⇒ many false positives
  - Non-conservative treatment ⇒ many false negatives
- Limited usefulness for security analysis

Technical Challenges

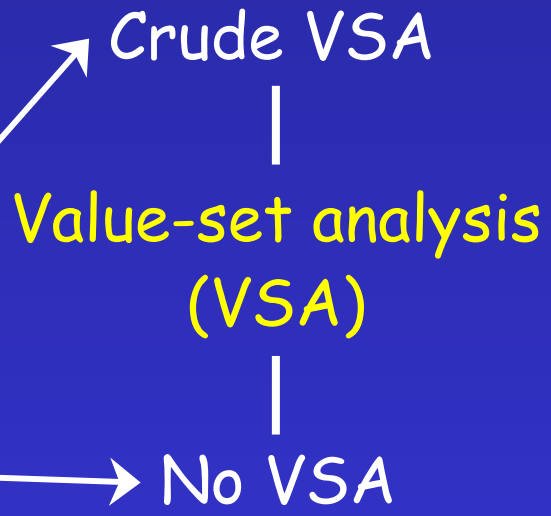
- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

Technical Challenges

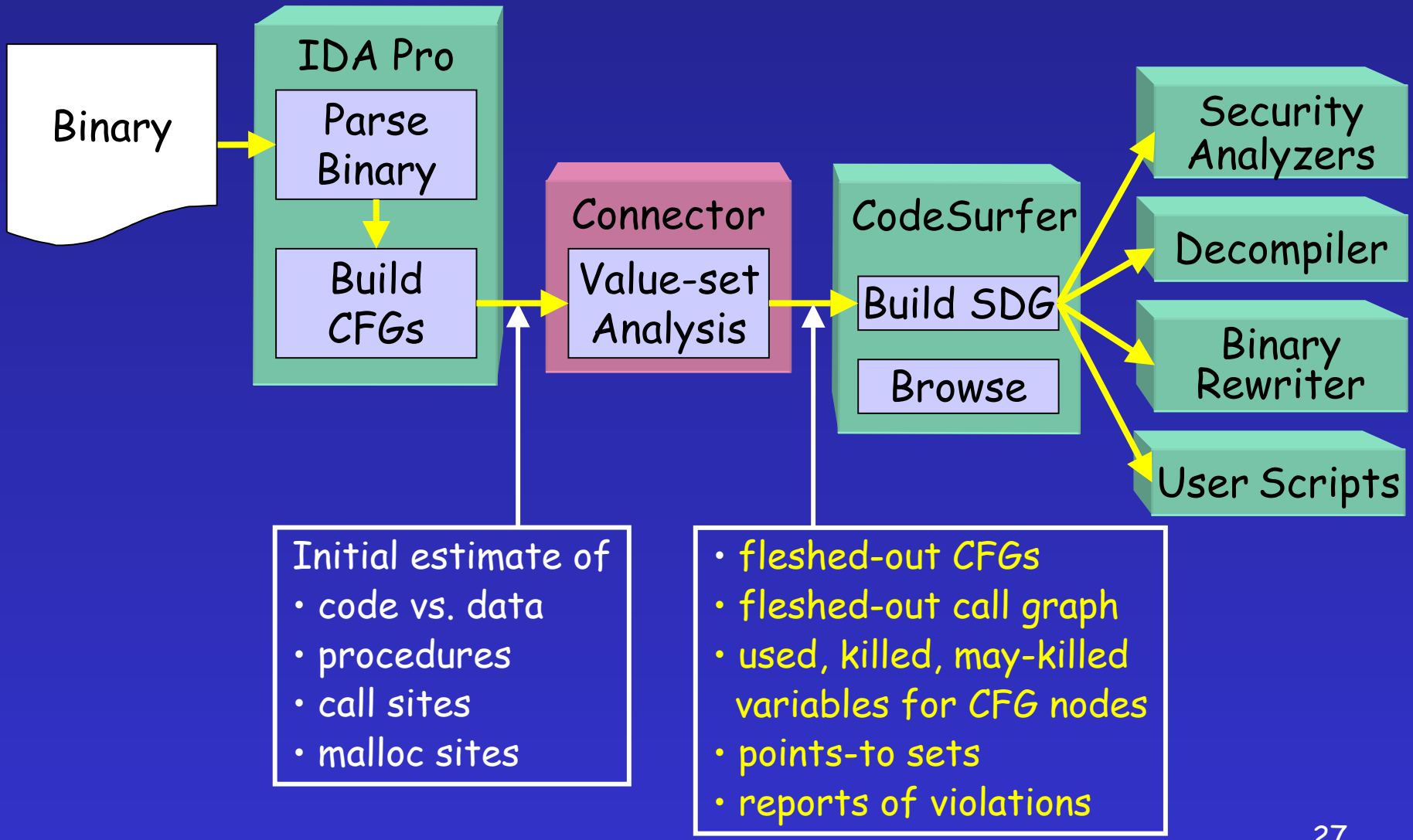
- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

Static Analysis of Executables: State of the Art

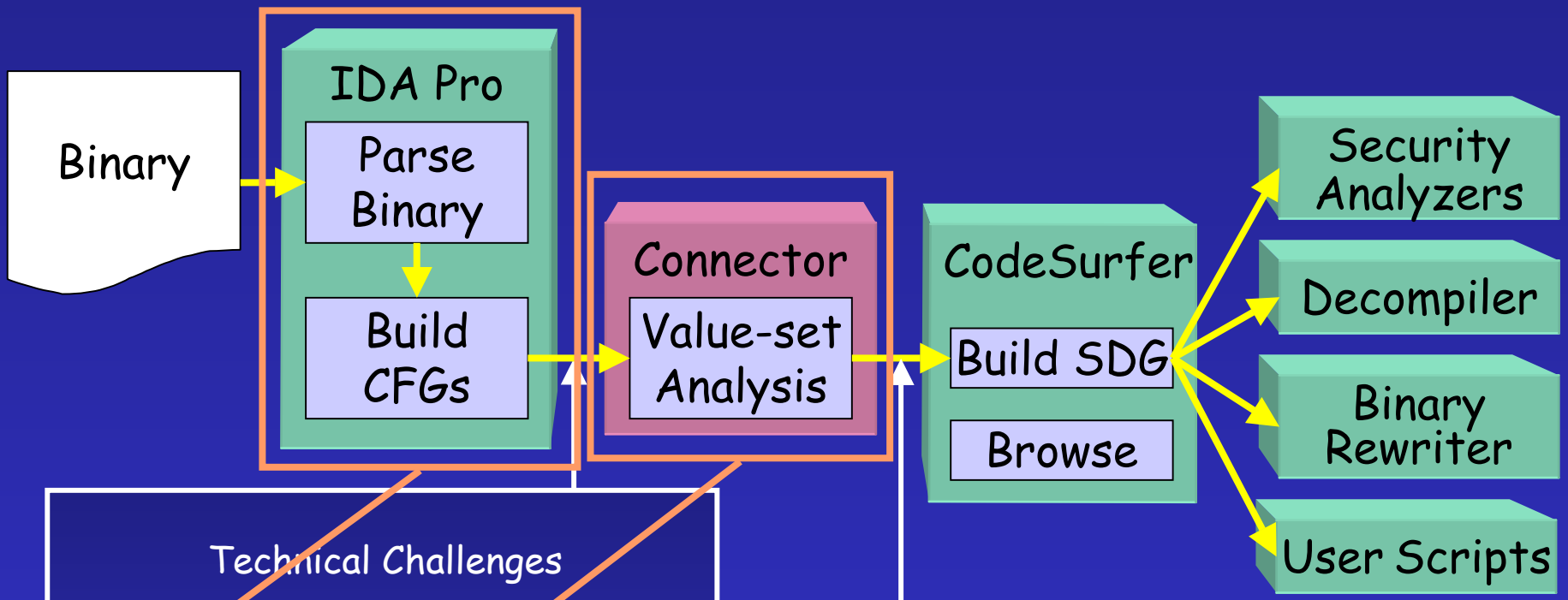
- Relies on symbol-table/debugging info
  - Atom, EEL, Vulcan, Rival
- Able to track only data movements via registers
  - EEL, Cifuentes, Debbabi, Debray
- Poor treatment of memory operations
  - Overly conservative treatment ⇒ many false positives
  - Non-conservative treatment ⇒ many false negatives
- Limited usefulness for security analysis



CodeSurfer/x86 Architecture



CodeSurfer/x86 Architecture



Technical Challenges

- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

Outline

- Challenges in analysis of executables
- Demo of CodeSurfer/x86
- Value-set analysis
- Better identification of variables
- Wrap-up

Demo

- CodeSurfer/C
- CodeSurfer/x86

Outline

- Challenges in analysis of executables
- Demo of CodeSurfer/x86
- Value-set analysis
- Better identification of variables
- Wrap-up

Challenges

- Explicit memory addresses and offsets
  - yet we need something similar to C variables
- Indirect-addressing mode
  - need "pointer analysis"
- Pointer arithmetic
  - need "numeric analysis" (e.g., range analysis)
- Checking for non-aligned accesses
  - pointer forging? e.g., 4-byte fetch from [1020,1028]

  - need to keep stride information:
    e.g., 4-byte fetch from 4*[0,2] + 1020

Arithmetic on Integers vs. Ints

- Machine arithmetic is 32-bit two's complement (`int`)
- Static analysis based on infinite-precision integer arithmetic is unsound

$2^{32} \geq 0$

Unreachable!

```
x = 2^32 - 1;
x = x + 1;
if (x == 0) {
   ⟨Do something malicious⟩
}
```

Running Example

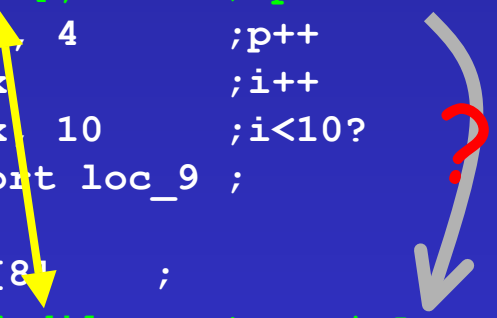
```
int arrVal=0, *pArray2;

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 ⇔ variable i
; ecx ⇔ variable p

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

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, [8]      ;
mov     eax, [edi]    ;return *pArray2
add     esp, 40
retn
```

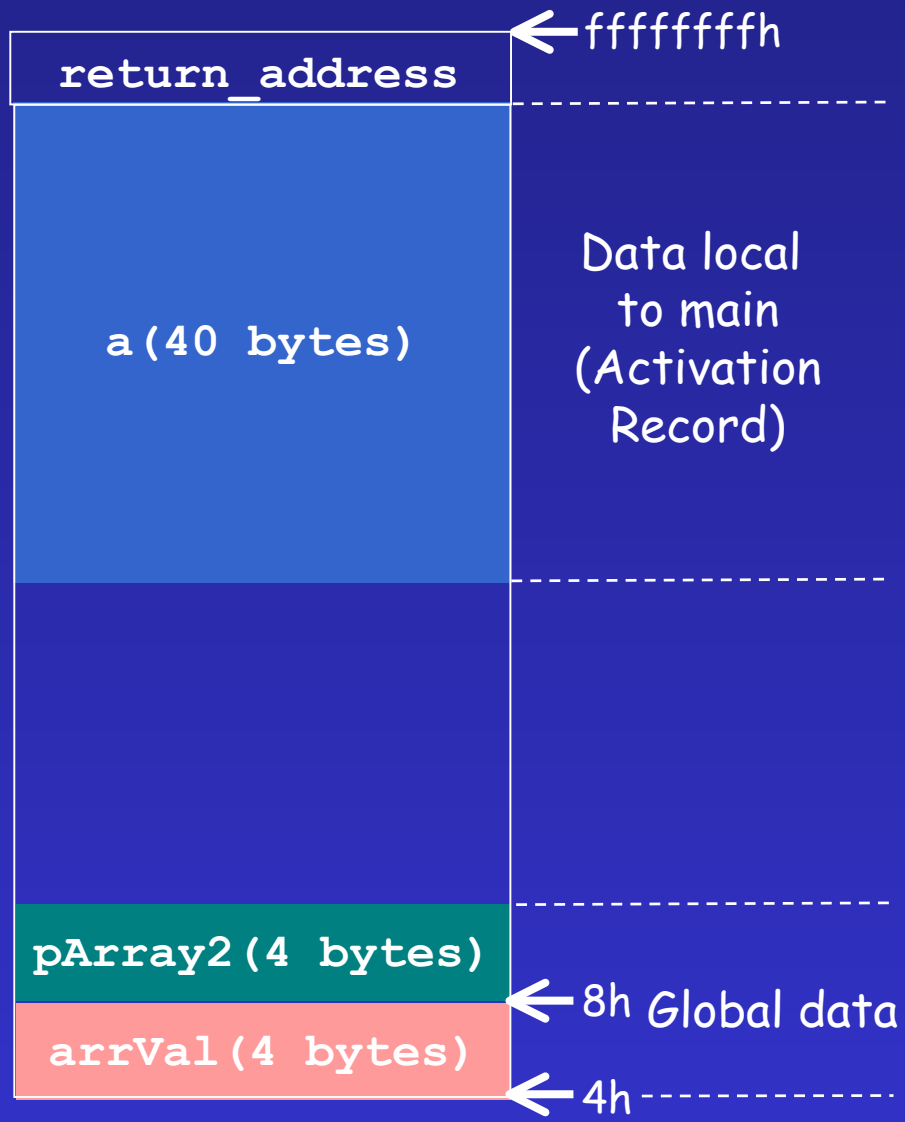
Running Example

```
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 ⇔ variable i
; ecx ⇔ variable p

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

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, [8]        ;
mov     eax, [edi]      ;return *pArray2
add     esp, 40
retn
```

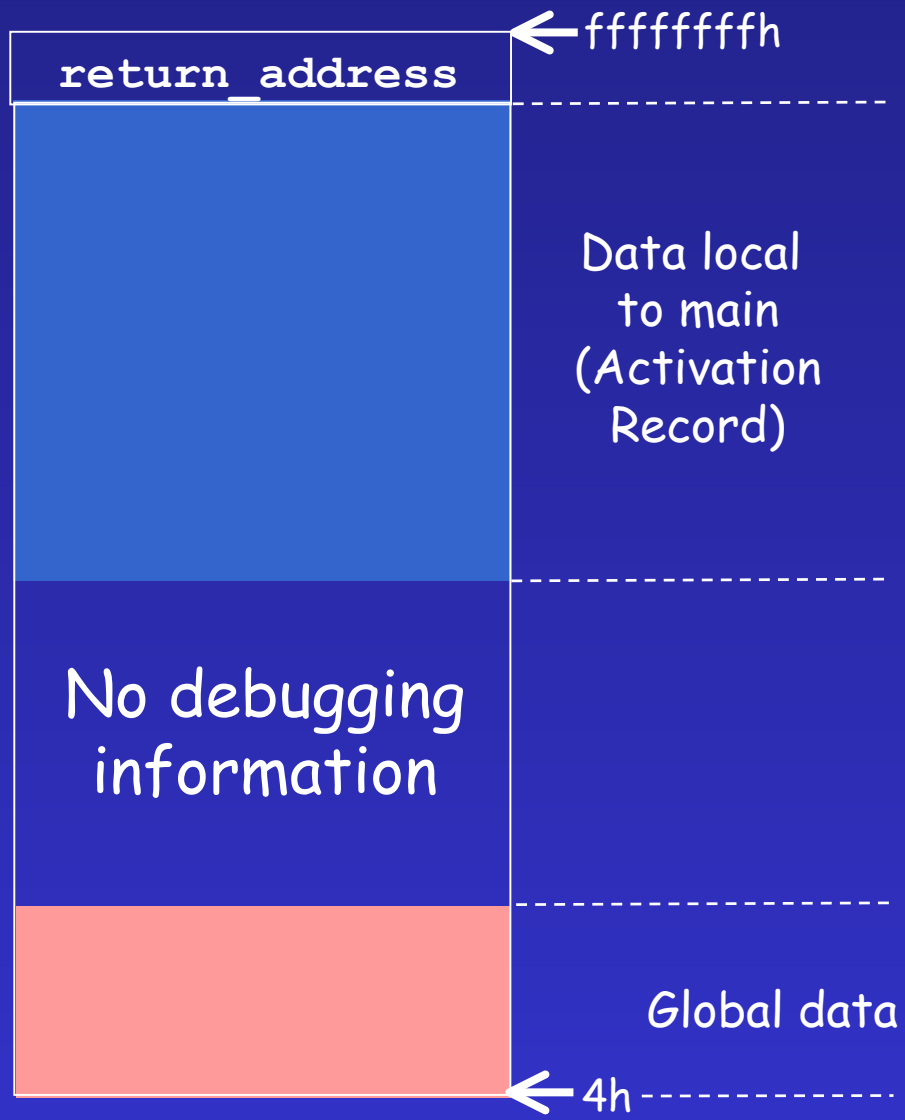


Running Example - Address Space




```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]        ;
mov     eax, [edi]      ;return *pArray2
add     esp, 40
retn
```

Running Example - Address Space



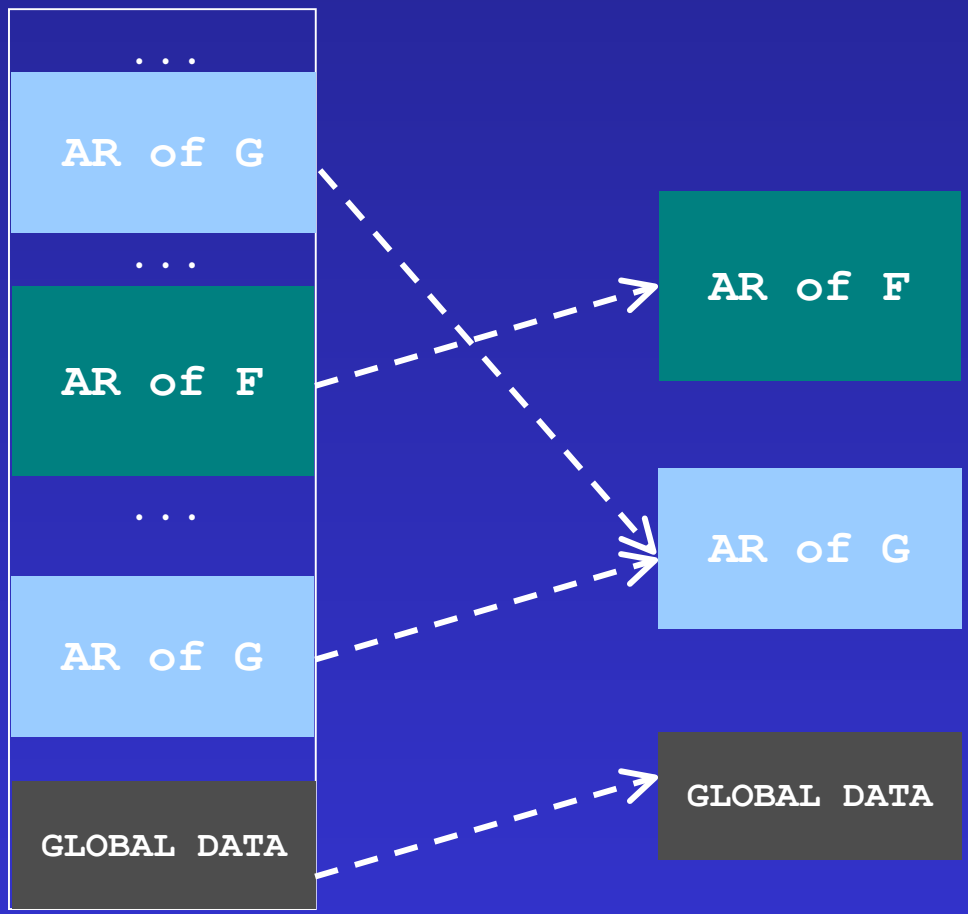

```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]        ;
mov     eax, [edi]      ;return *pArray2
add     esp, 40
retn
```

A Bird's Eye View

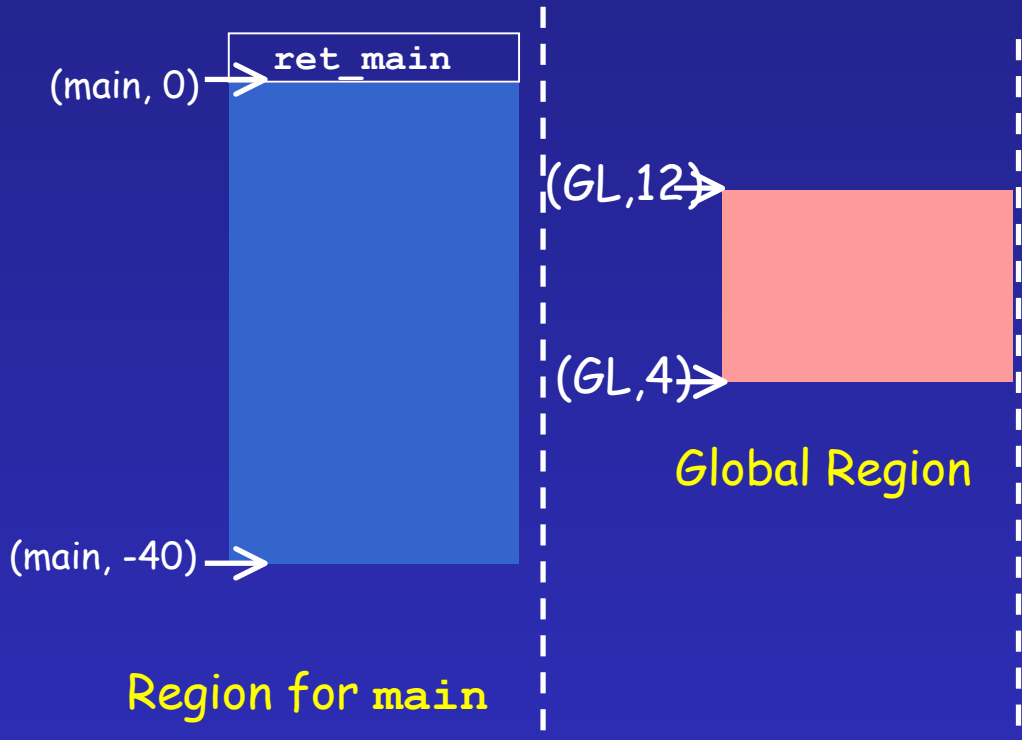
- An abstraction of concrete memory configurations
  - Memory regions
- Infer layout of memory regions
  - A-locs (like variables)
- Perform a combined pointer and numeric analysis
  - Value-set analysis
  - Over-approximate the set of values/addresses held by an a-loc

Memory Regions

- An abstraction of concrete memory configurations
  - Idea: group similar runtime addresses
  - e.g., collapse the runtime ARs for each procedure, malloc-sites, global data



Example - Memory Regions

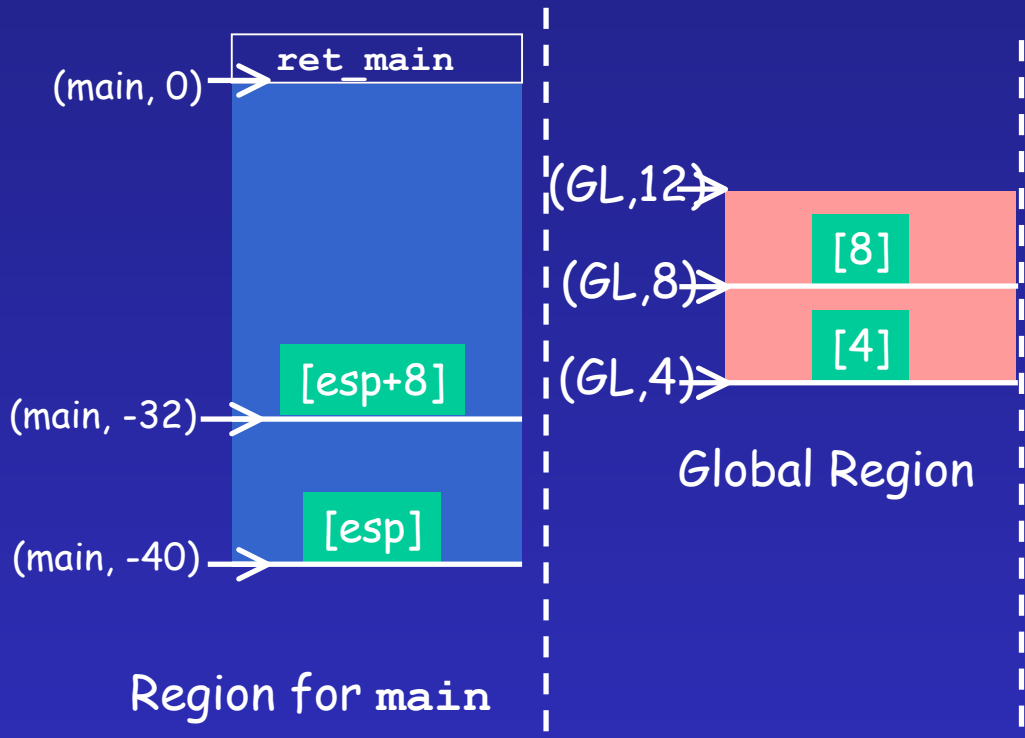



```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]      ;
mov     eax, [edi]    ;return *pArray2
add     esp, 40
retn
```

Infer Layout of Memory Regions

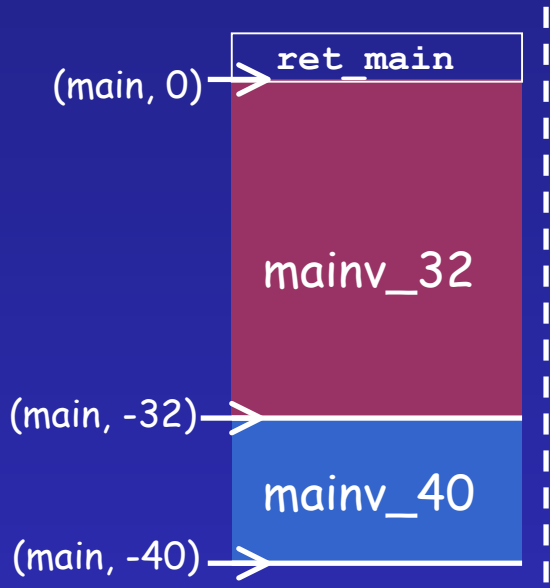
- What does the compiler do?
  - some variables held in registers
  - global variables → absolute addresses
  - local variables → offsets in stack frame
- *A*-locs
  - locations between consecutive addresses
  - locations between consecutive offsets
  - registers

Example - A-locs

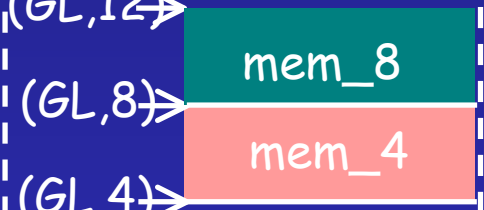



```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]         ;
mov     eax, [edi]       ;return *pArray2
add     esp, 40
retn
```

Example - A-locs



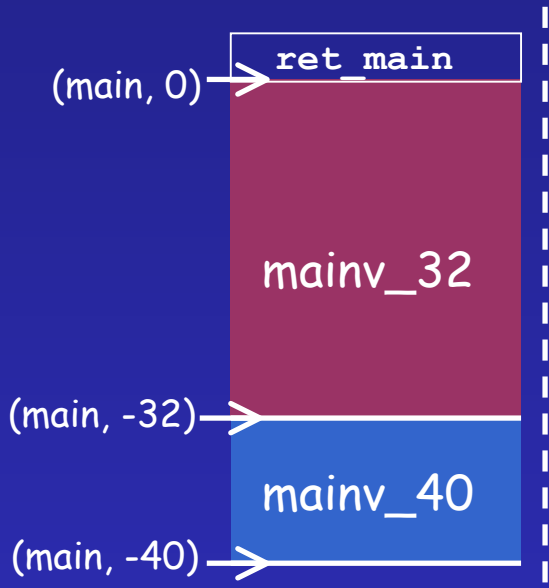
Region for `main`



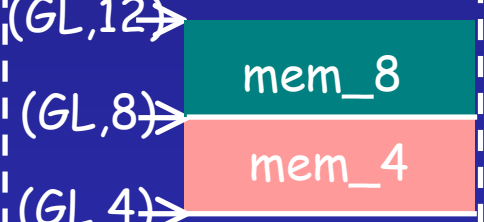
Global Region

```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]         ;
mov     eax, [edi]       ;return *pArray2
add     esp, 40
retn
```

Example - A-locs



Region for `main`



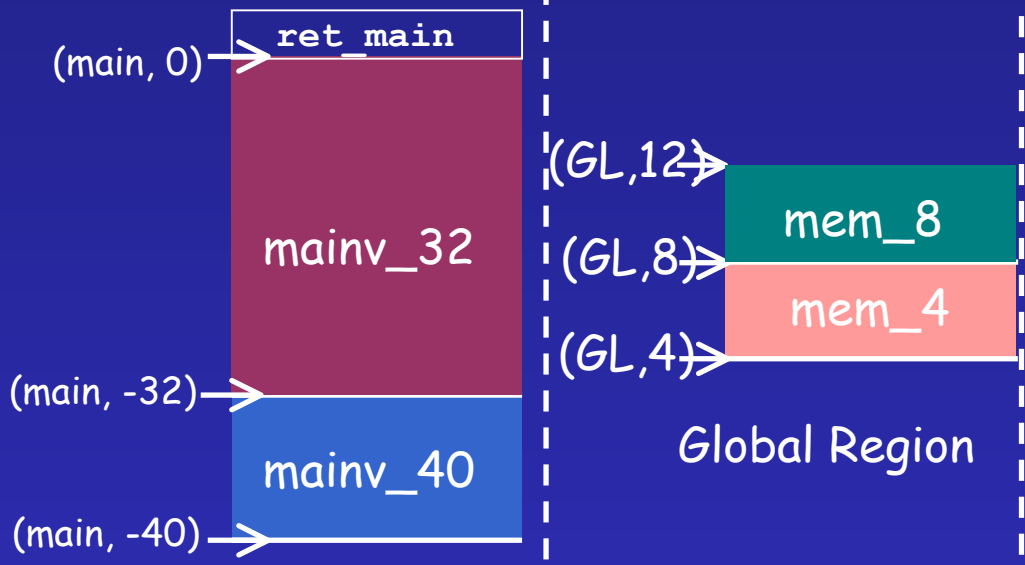
Global Region

```
; ebx ⇔ variable i
; ecx ⇔ variable p

sub     esp, 40             ;adjust stack
lea     edx, &mainv_32;
mov     mem_8, edx          ;pArray2=&a[2]
lea     ecx, &mainv_40;p=&a[0]
mov     edx, mem_4          ;

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_8      ;
mov     eax, [edi]      ;return *pArray2
add     esp, 40
retn
```

Value-Set Analysis

- Dataflow analysis to identify contents of a-locs
  - over-approximate the set of addresses/values held by an a-loc
- Reduced Interval Congruence (RIC)
  - represents a set of values
  - records a range and a stride
  - {1, 3, 5, 9} represented as 2[0,4] + 1
  - conservative: 2[0,4]+1 represents {1, 3, 5, 7, 9}
- Value-set
  - tuple of RICs: (ric_1, . . ., ric_r)
  - i^{th} component: the offsets in the i^{th} memory-region

Example - Value-set analysis



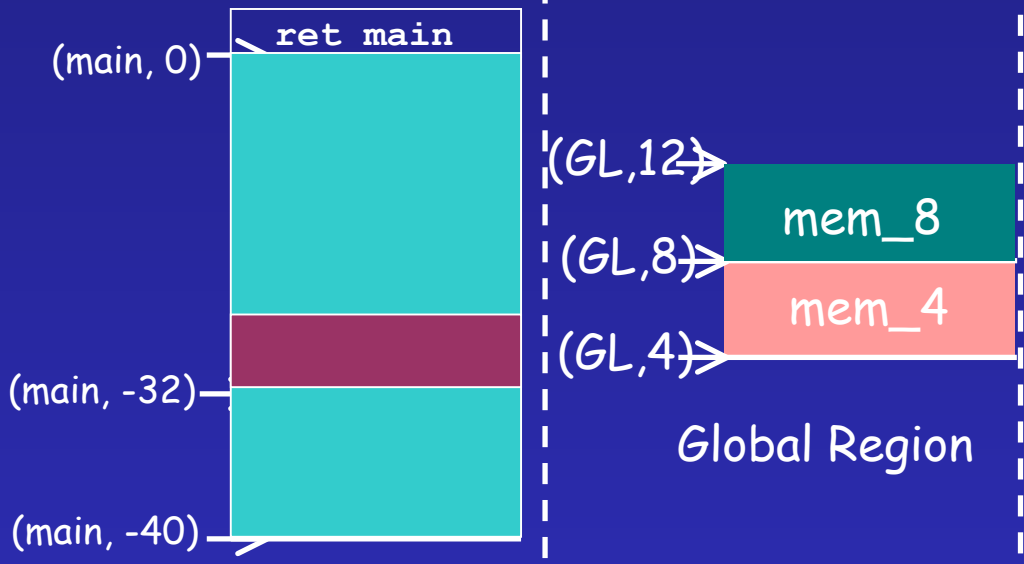
Region for **main**

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

2 **edi** → (⊥, -32)
esp → (⊥, -40)

```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]        ;
mov     eax, [edi]      ;return *pArr   2
add     esp, 40
retn
```

Example - Value-set analysis

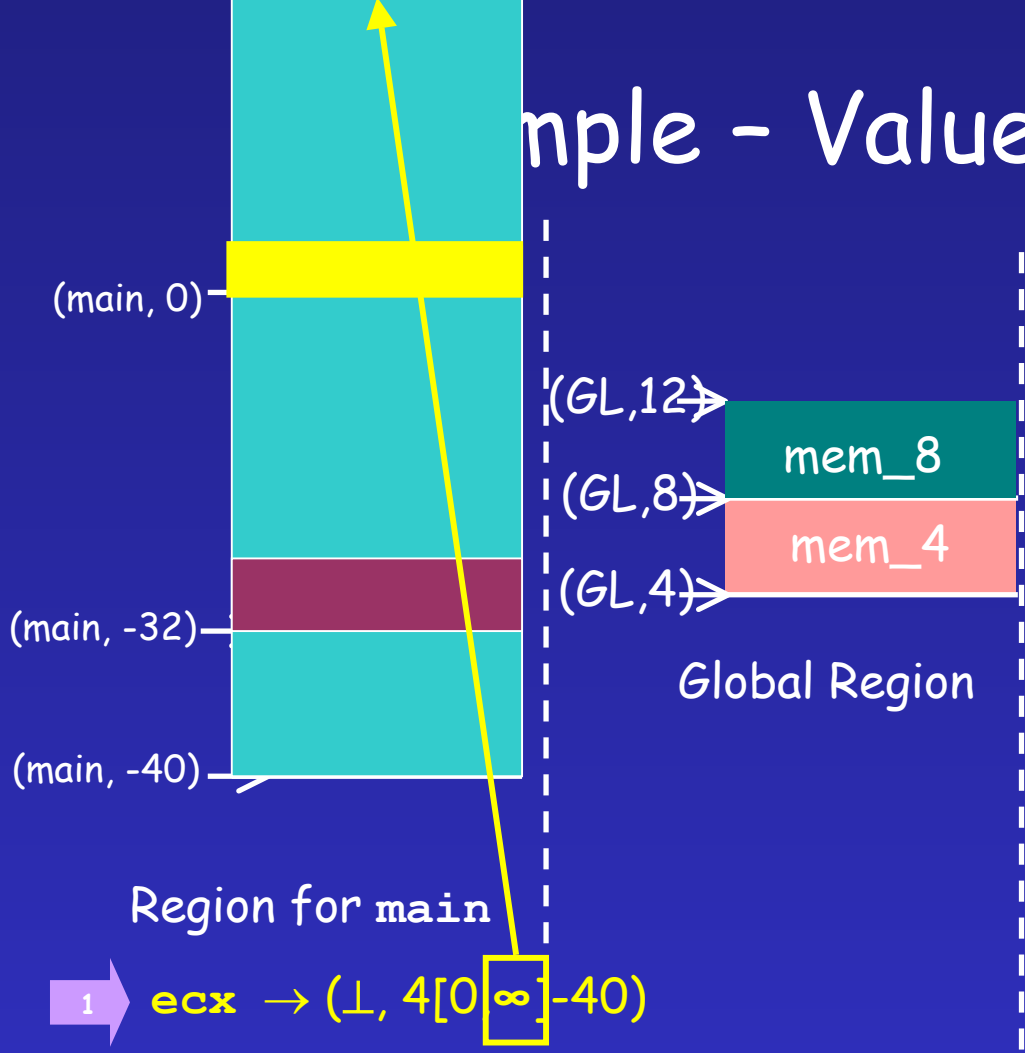


1 **ecx** → (⊥, 4[0,9]-40)

2 **edi** → (⊥, -32)

```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]     ;
mov     eax, [edi]   ;return *pArr    2
add     esp, 40
retn
```

mple - Value-set analysis




```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]     ;
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
- Improved by discovering affine relations
  - identifies a loop's induction variables

```
. . .
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

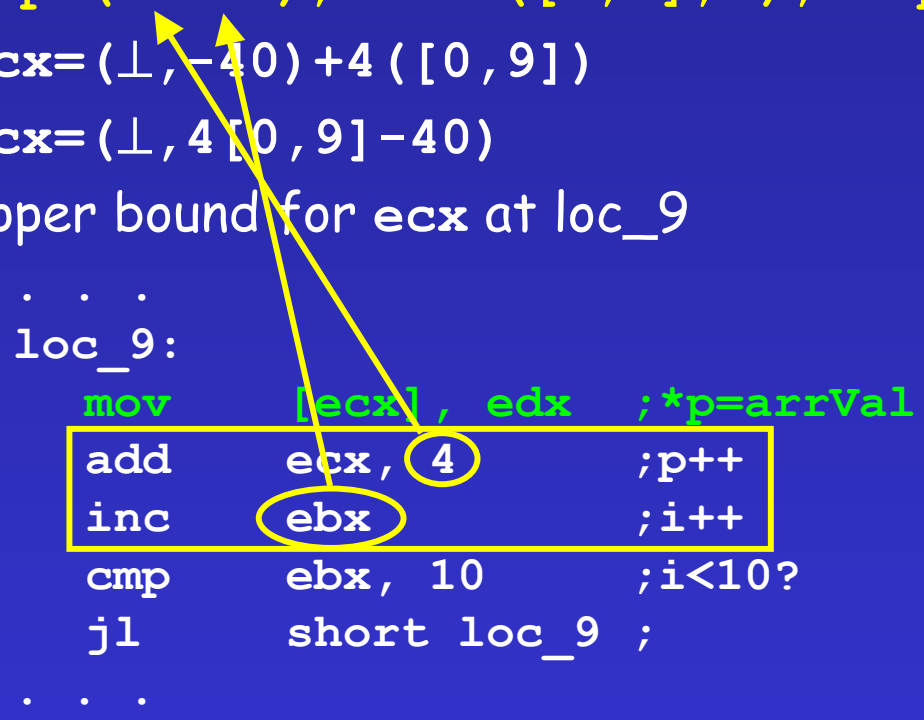
- Obtain affine relations via static analysis
- Use affine relations to improve precision
  - e.g., at loc_9

ecx=esp+(4×ebx), ebx=([0,9],⊥), esp=(⊥,-40)

⇒ **ecx=(⊥,-40)+4([0,9])**

⇒ **ecx=(⊥,4[0,9]-40)**

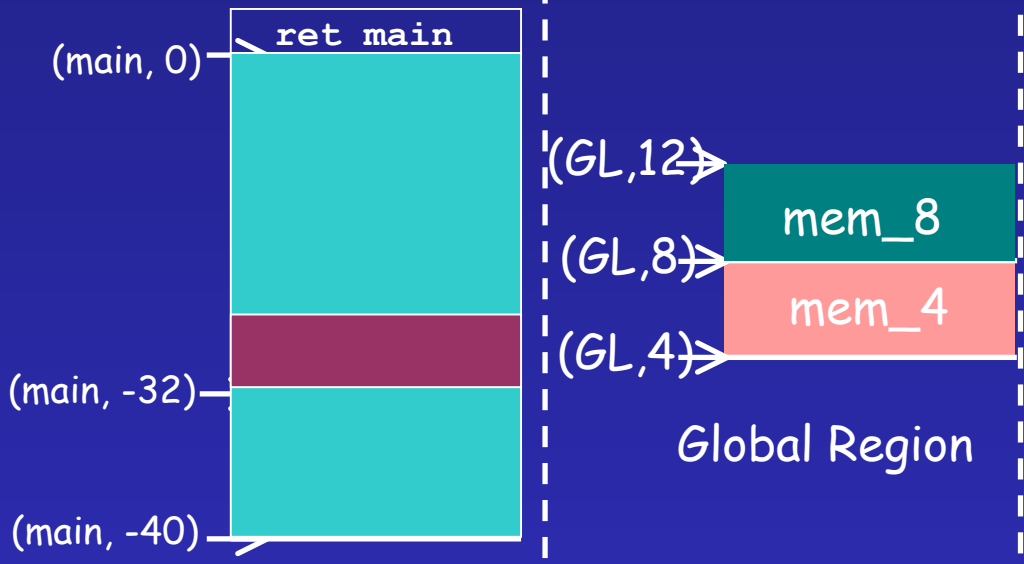
⇒ upper bound for **ecx** at loc_9



Affine-Relation Analysis

- Affine relation
  - $x_1, x_2, \ldots, x_n$ – a-locs
  - $a_0, a_1, \ldots, a_n$ – integer constants
  - $a_0 + \sum_{i=1..n}(a_i x_i) = 0$
  - more general than
    - constant propagation
    - induction-variable analysis
- Idea: determine affine relations on registers
  - propagate loop-bound info to other registers
- Implemented using WPDS++

Example - Value-set analysis



1 **ecx** → (⊥, 4[0,9]-40)

2 **edi** → (⊥, -32)

```
; ebx ⇔ variable i
; ecx ⇔ variable p

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

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, [8]     ;
mov     eax, [edi]   ;return *pArr    2
add     esp, 40
retn
```

Related Work

- Debray et al., "Alias analysis of executable code" [POPL 98]
- Cifuentes et al., "Assembly to high-level language translation" [ICSM 98]
- Ramalingam et al., "Aggregate structure identification and its application to program analysis" [POPL 99]
- A. Mycroft, "Type-based decompilation" [ESOP 99]
- Dor et al., "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C" [PLDI 03]
- Linn et al., "Stack analysis of x86 executables" [Unpublished]

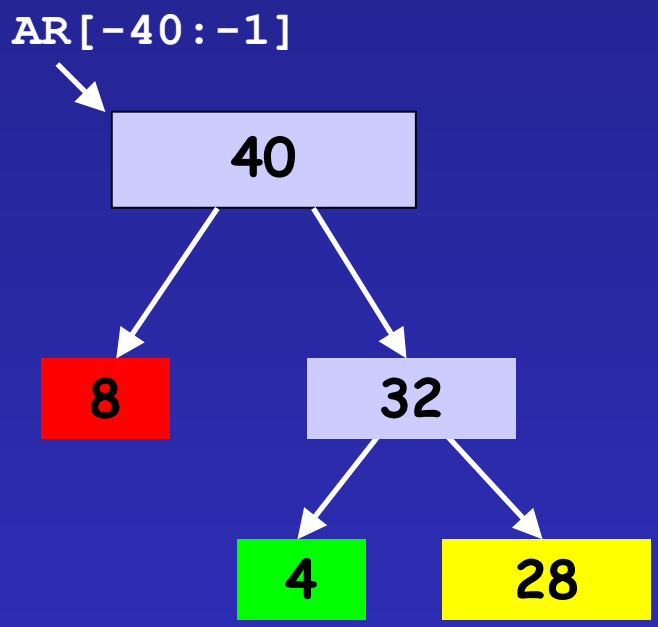
Outline

- Challenges in analysis of executables
- Demo of CodeSurfer/x86
- Value-set analysis
- Better identification of variables
- Wrap-up

On-Going Work: Aggregate Structure Identification

- Partition aggregates according to the program's memory-access patterns
  - Original motivation: Y2K [Ramalingam et al. POPL 99]
- Uses in our context
  - improved identification of variables
    - identifies a better set of a-locs
  - recovery of type information
    - identifies structs and arrays
    - propagates type information from known parameter types (system calls & library functions)

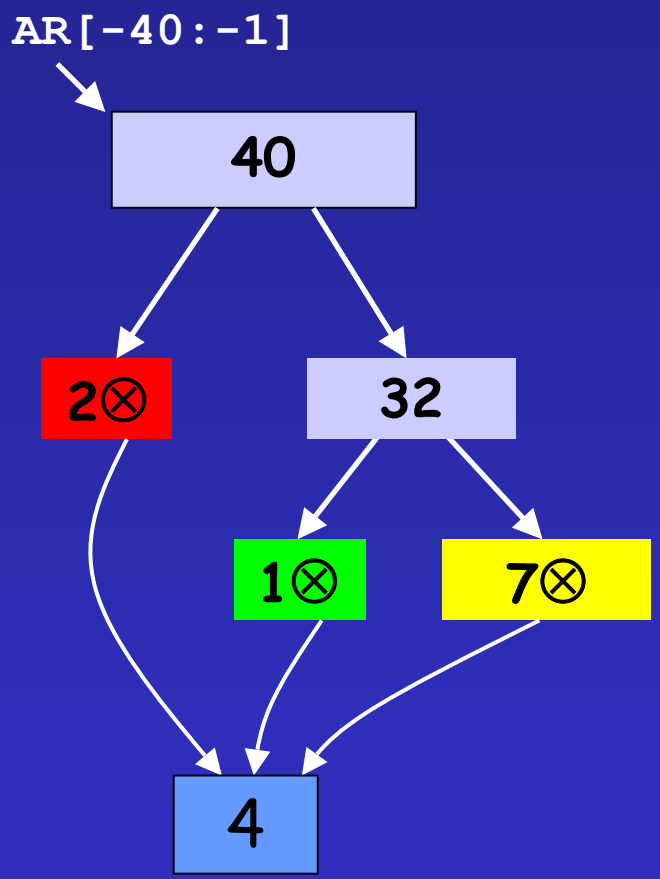
Aggregate Structure Identification



edi → (⊥, -32)

```
; ebx ⇔ variable 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
```

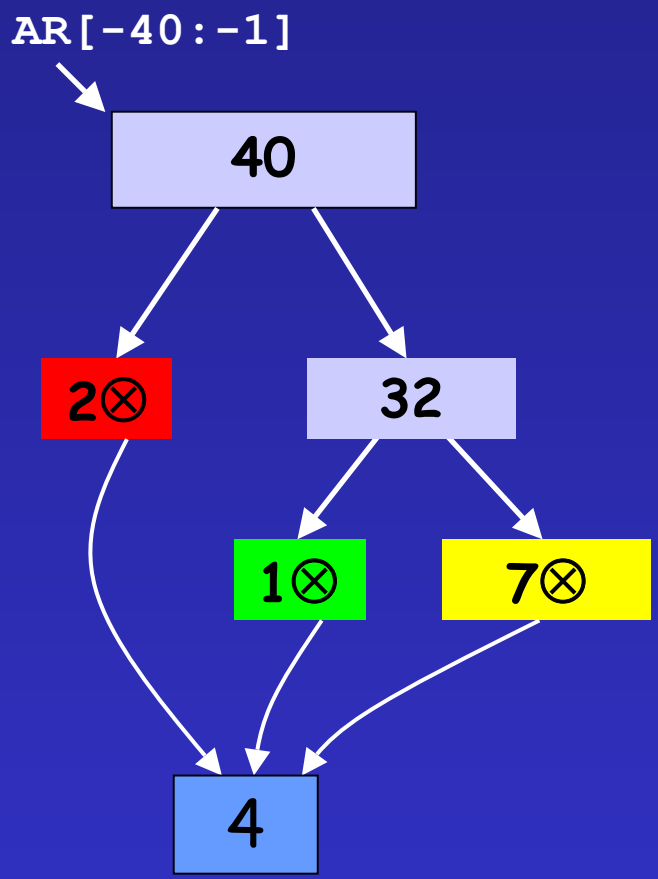
Aggregate Structure Identification



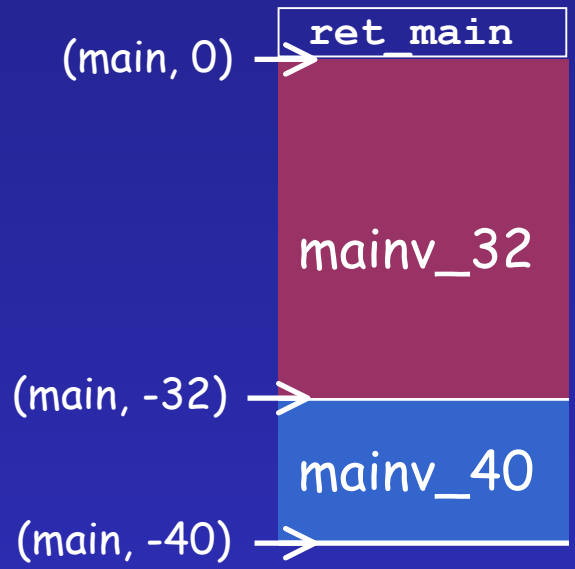
ecx → (⊥, 4[0,9]-40)

```
; ebx ⇔ variable 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
```

Aggregate Structure Identification



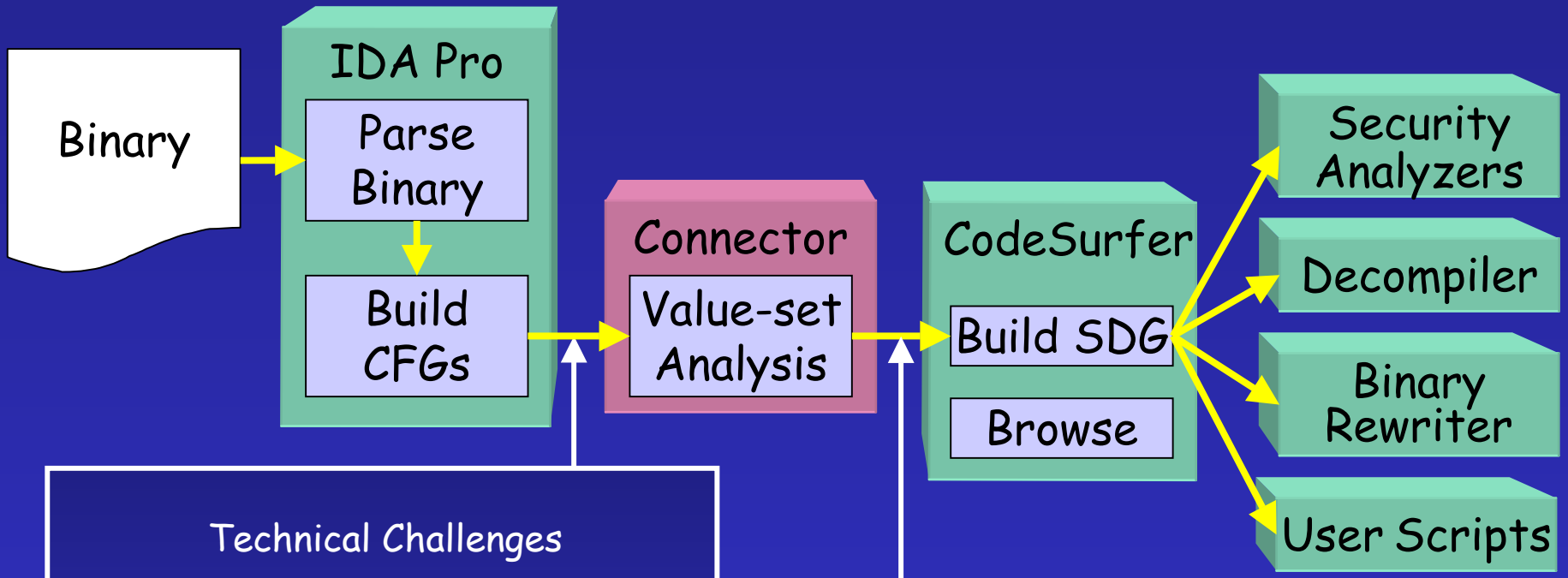
ASI: ten 4-byte a-locs



Region for **main**

IDA Pro
one 8-byte a-loc
one 32-byte a-loc

CodeSurfer/x86 Architecture

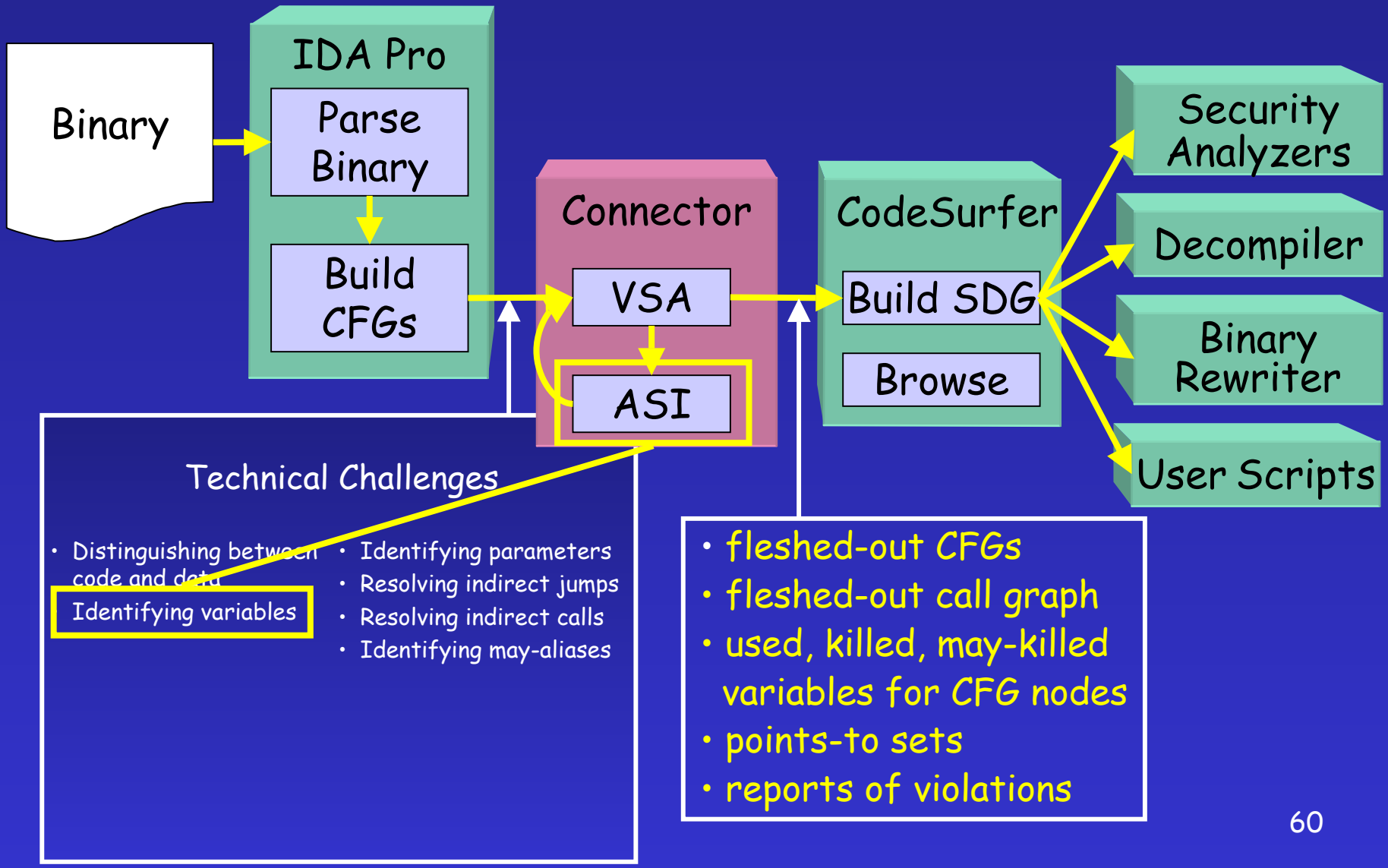


Technical Challenges

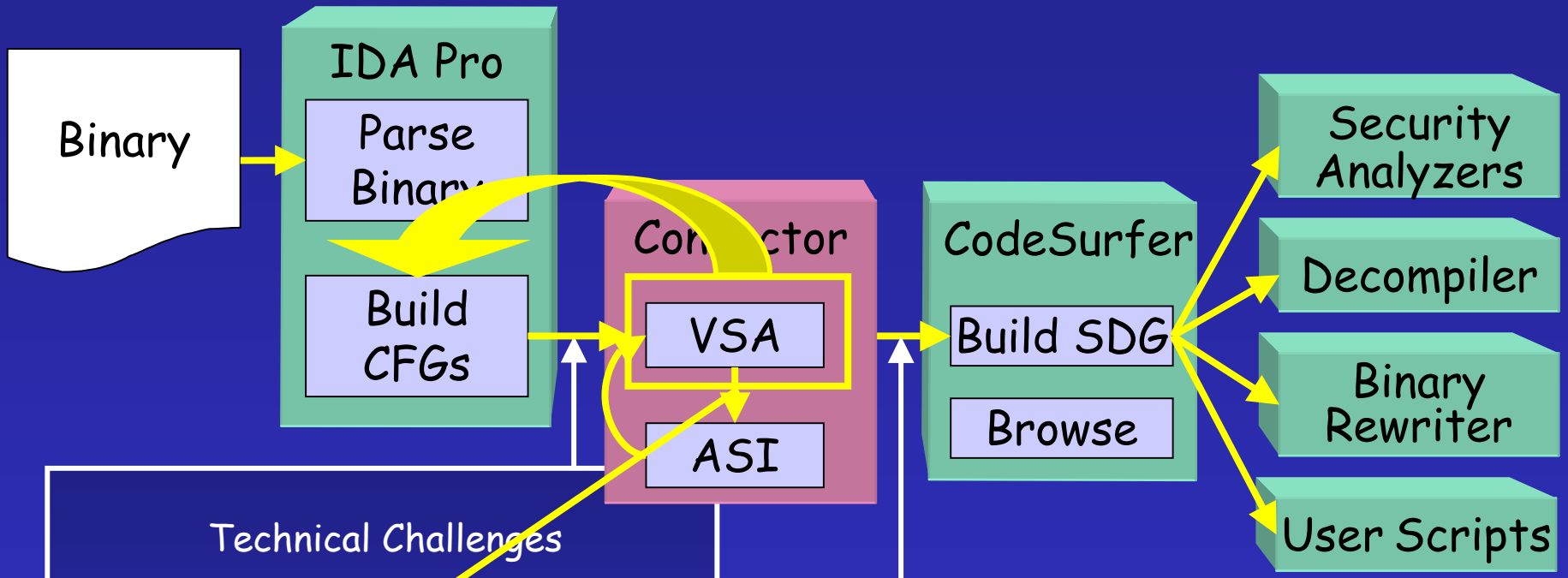
- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

CodeSurfer/x86 Architecture



CodeSurfer/x86 Architecture



Technical Challenges

- Distinguishing between code and data
- Identifying variables
- Identifying parameters
- Resolving indirect jumps
- Resolving indirect calls
- Identifying may-aliases

- fleshed-out CFGs
- fleshed-out call graph
- used, killed, may-killed variables for CFG nodes
- points-to sets
- reports of violations

Demo

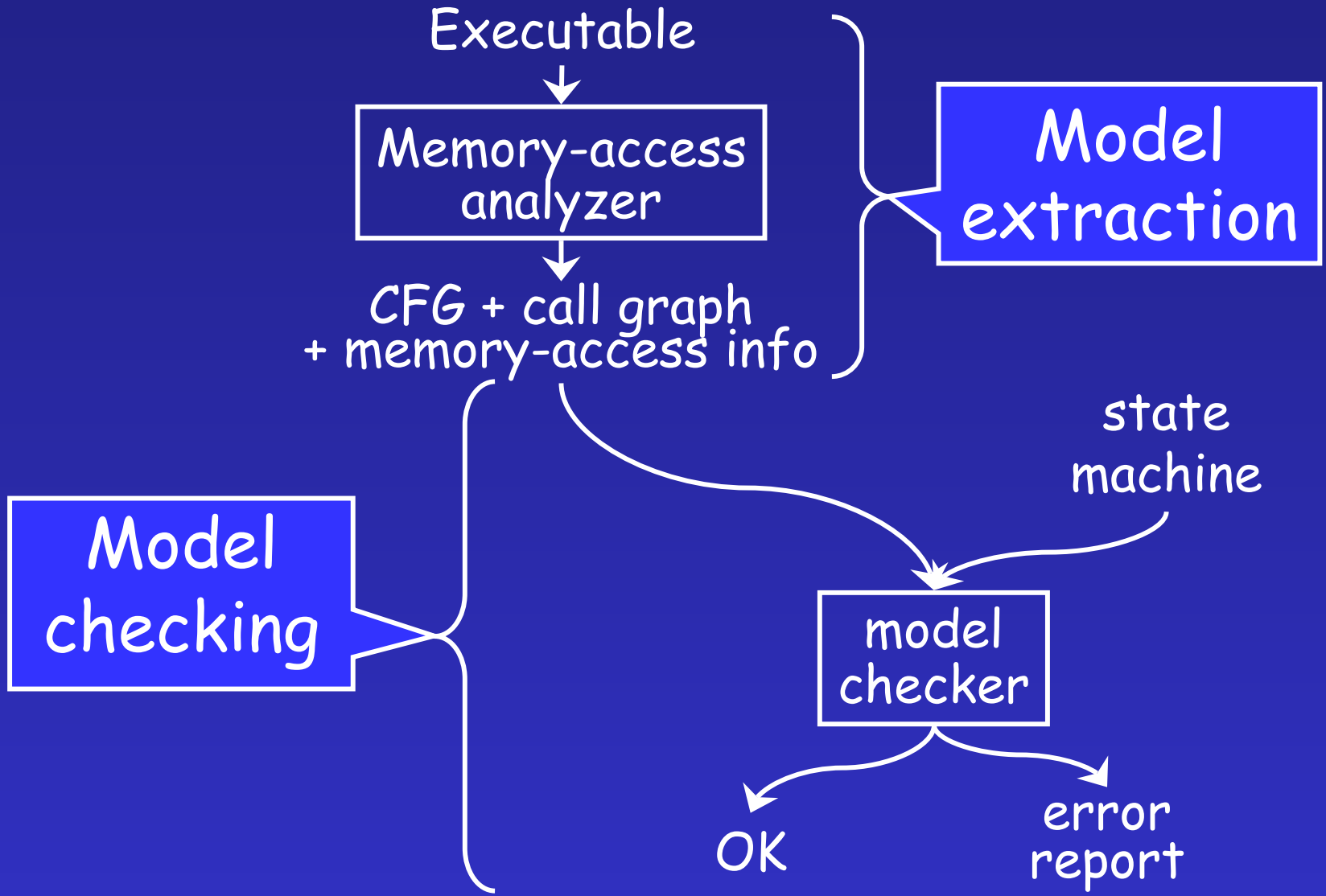
- CodeSurfer/C
- CodeSurfer/x86-0
- CodeSurfer/x86-1
- CodeSurfer/x86-2
- Memory-layout results

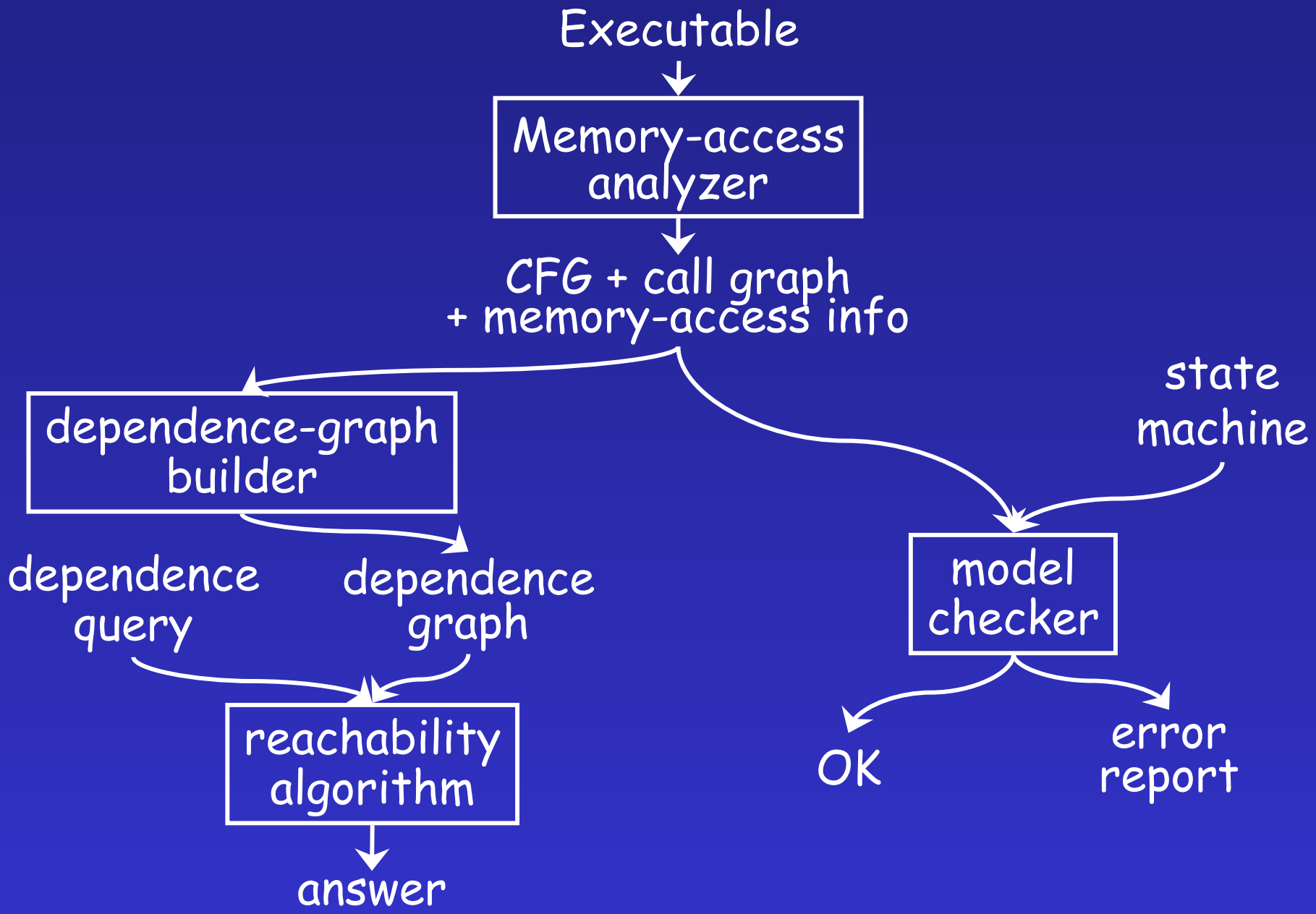
Outline

- Challenges in analysis of executables
- Demo of CodeSurfer/x86
- Value-set analysis
- Better identification of variables
- Wrap-up

Another Talk: IR Exploration

- API for traversal/searching/pattern matching
- API for defining static-analyzers/model-checkers
  - Use a script to traverse IR
  - Create a WPDS++ specification of desired analysis
  - Invoke WPDS++
- Path Explorer tool
  - Software-assurance plug-in to CodeSurfer/x86
  - Performs security-related analyses on the IR
    - Built on top of WPDS++
  - Uses the GUI to investigate warnings
- WPDS++ implements weighted pushdown-systems
  - cf. MOPS [Chen & Wagner, CCS04], PDSs w/o weights


Executable
Memory-access analyzer
CFG + call graph + memory-access info
Model extraction
Model checking
state machine
model checker
OK
error report


Executable
Memory-access analyzer
CFG + call graph + memory-access info
dependence-graph builder
dependence query
dependence graph
reachability algorithm
answer
state machine
model checker
OK
error report

What to Take Away

- Compelling case for performing assurance analyses on executables: "acceptance evaluation
  blem"
  basic science of abstract
  rpret nce m-
  sy em m an ture
  ide tificat deeply techn approach to
  assurance, originated in university labs with DARPA, NSF, and OSD/ONR sponsorship.

Bill Scherlis [1988-96]

Helen Gill [1991-96]

Ralph Wachter
Gary Toth

Who Cares?

Daniel Wolf, IA Director of NSA

"A significant cybersecurity improvement over the next decade will be found in enhancing our ability to find and eliminate malicious code in large software applications . . . There is little coordinated effort today to develop tools and techniques to examine effectively and efficiently either source or executable software. I believe that this problem is significant enough to warrant a considerable effort coordinated by a truly National Software Assurance Center."

For More Information

http://www.cs.wisc.edu/~reps/#staticAnalysisOfExecutables

http://www.cs.wisc.edu/~reps/#cc04