

A Next-Generation Platform for Analyzing Executables

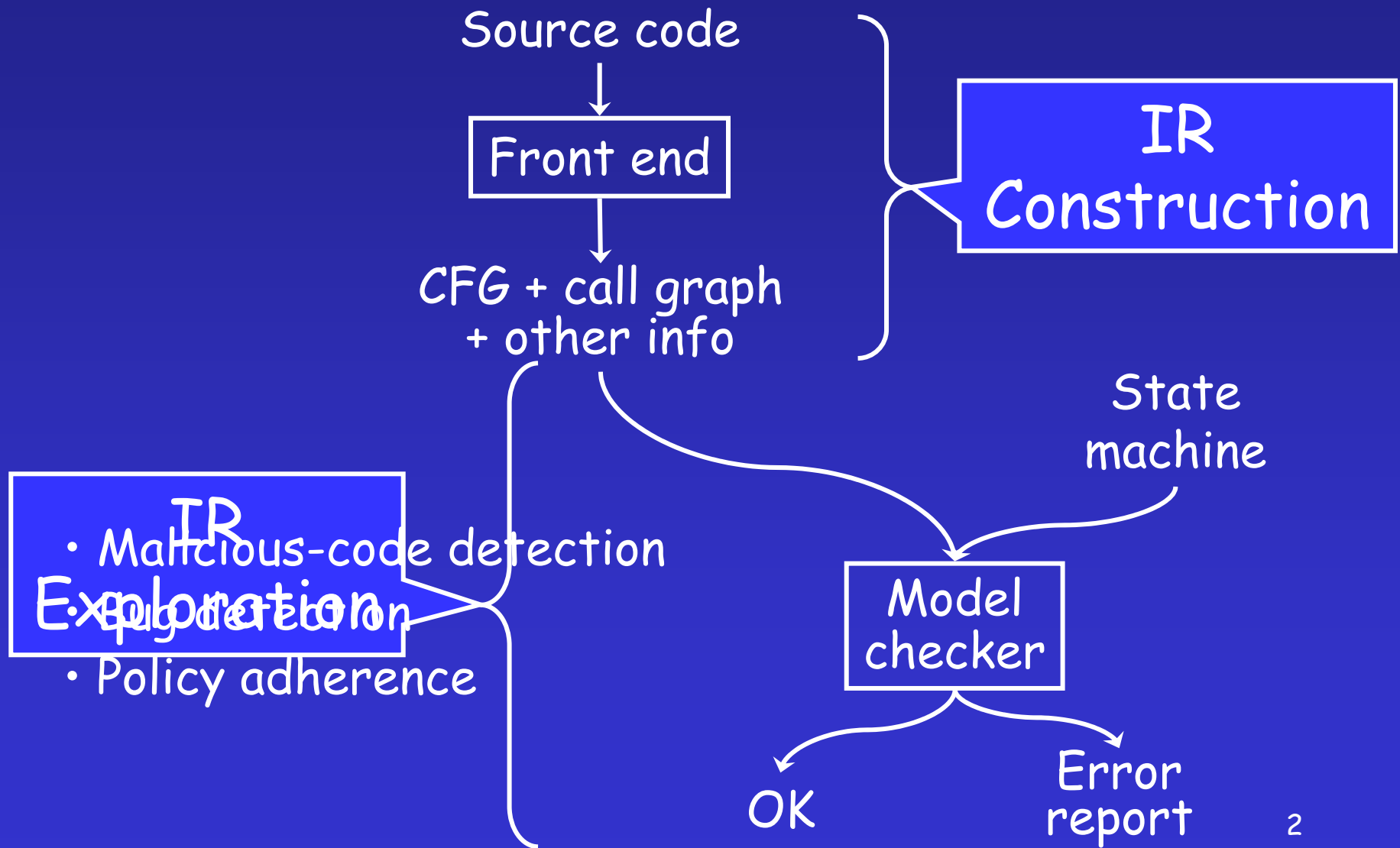
Thomas Reps^{1,2} and Tim Teitelbaum^{1,3}

¹GammaTech, Inc.

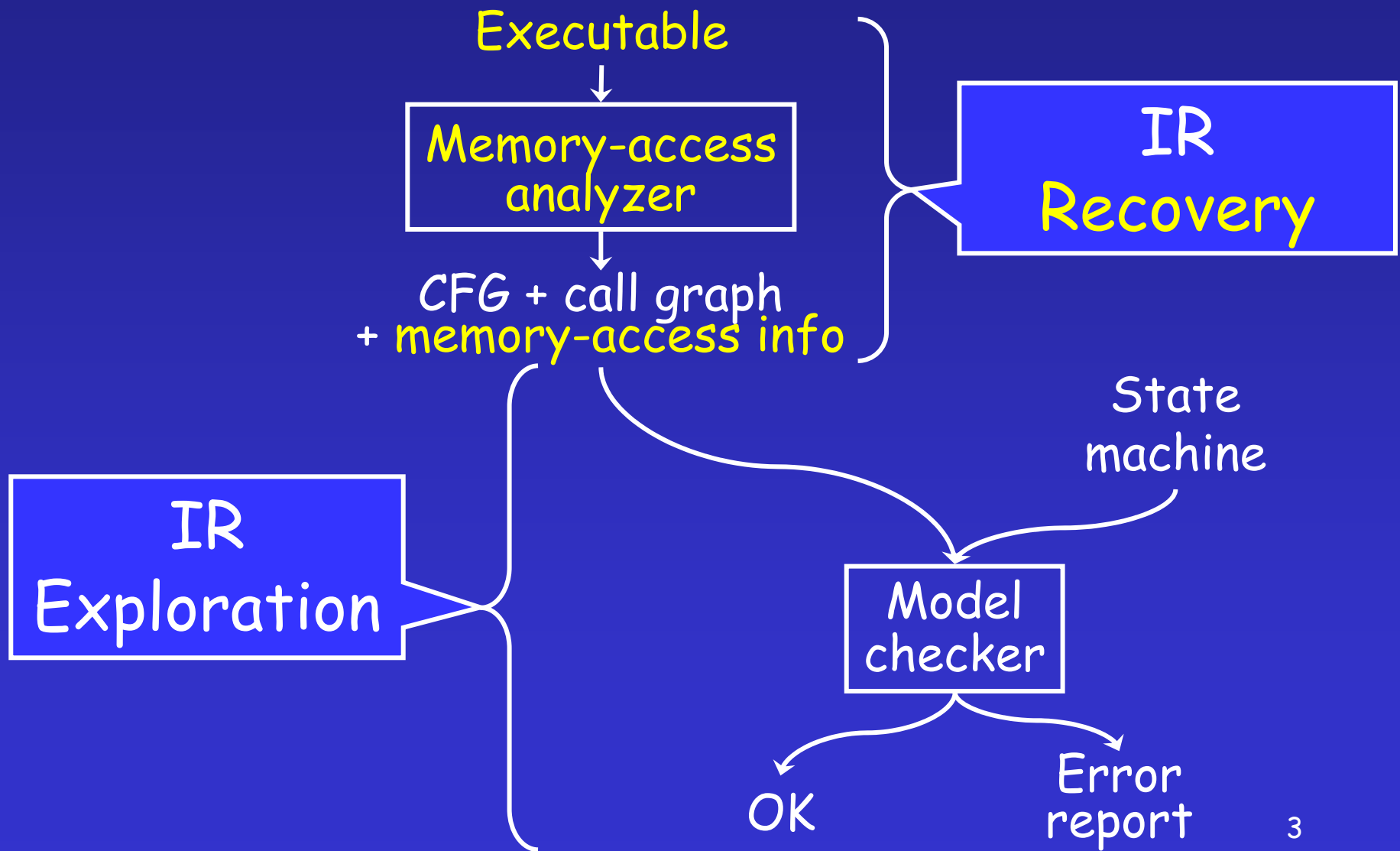
²University of Wisconsin

³Cornell University

State-Space Exploration



State-Space Exploration




Why Executables?

- Reveals platform-specific choices made by compiler
 - *What you see is what you get*
- Some source-level issues go away
- Better platform for finding security vulnerabilities
 - *Source-code tools: Lack of fidelity can allow vulnerabilities to escape detection*

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 ("useless-code" elimination)

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



```
free(buffer);
```

WYSINWYX:

What You See Is Not What You eXecute

- Computers do not execute source-code programs; they execute machine-code programs that are generated from source code
- What is actually executed by the processor may not be what the programmer intends
 - overzealous optimizer
 - a language's semantics may leave some behaviors unspecified

The Vision

- Code-inspection tools for security analysts
- Analyses for identifying
 - security vulnerabilities and bugs
 - malicious behavior (code vs. memory snapshots)
 - commonalities and differences
- Platform for
 - de-compilation
 - code obfuscation
 - installation of protection mechanisms
 - remediation of security vulnerabilities
 - de-obfuscation (w/ assistance from dyn. tools)

What Should a Tool Provide?

• 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

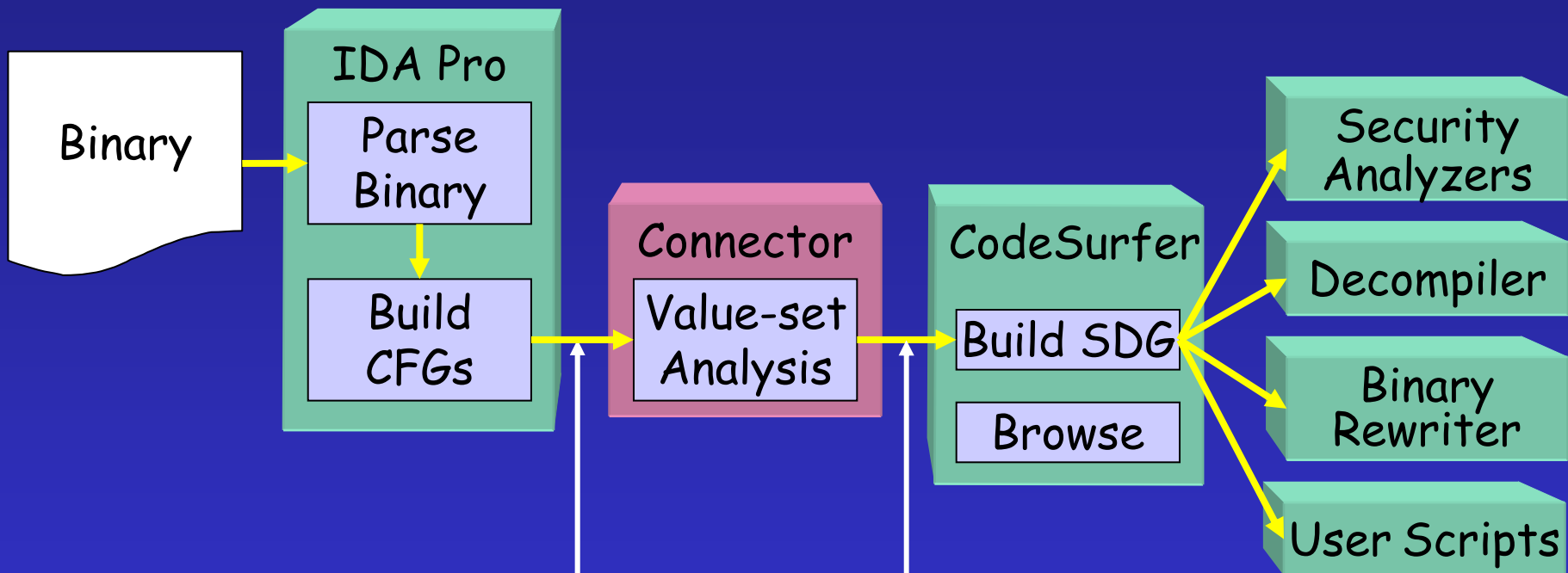
• No use of symbol-table or debugging information!!!

• IR exploration

- API for traversal/searching/pattern matching
- API for defining static-analyzers/model-checkers
- GUI to investigate warnings

• Cooperation with dynamic tools

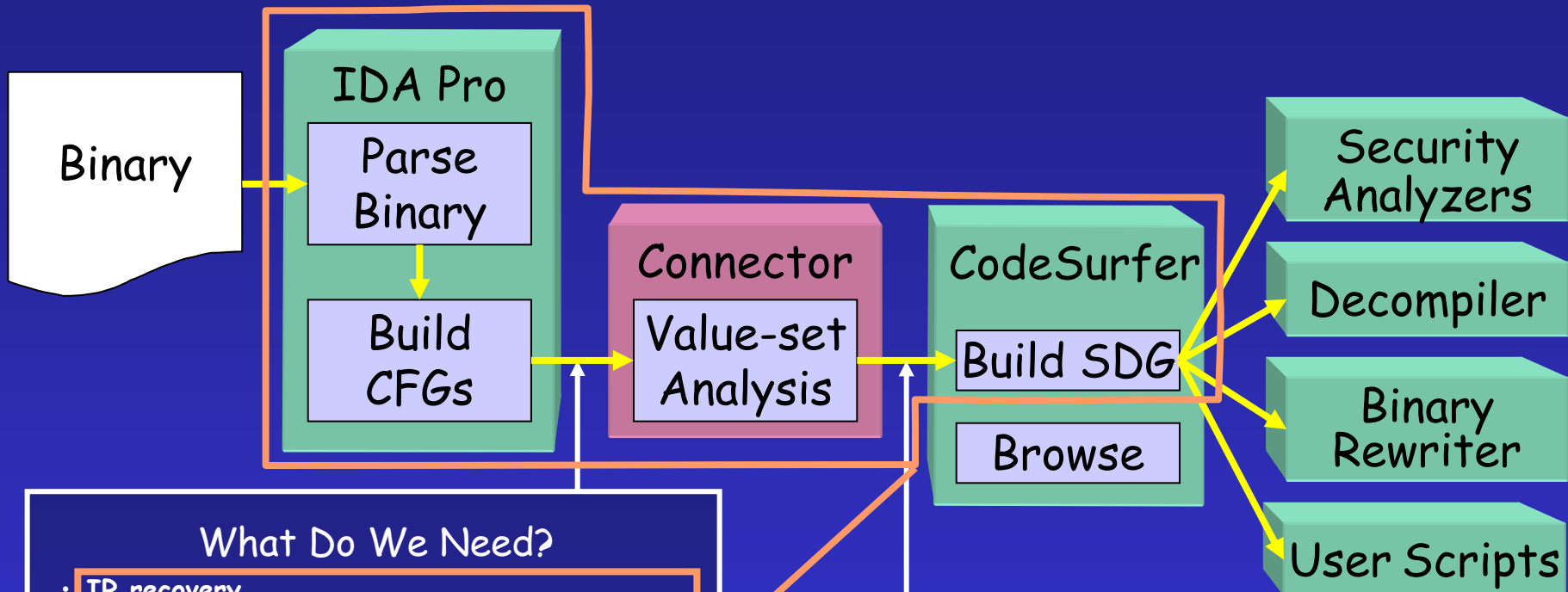
CodeSurfer/x86 Architecture



- Initial estimate of
- code vs. data
 - procedures
 - call sites
 - malloc sites

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

CodeSurfer/x86 Architecture

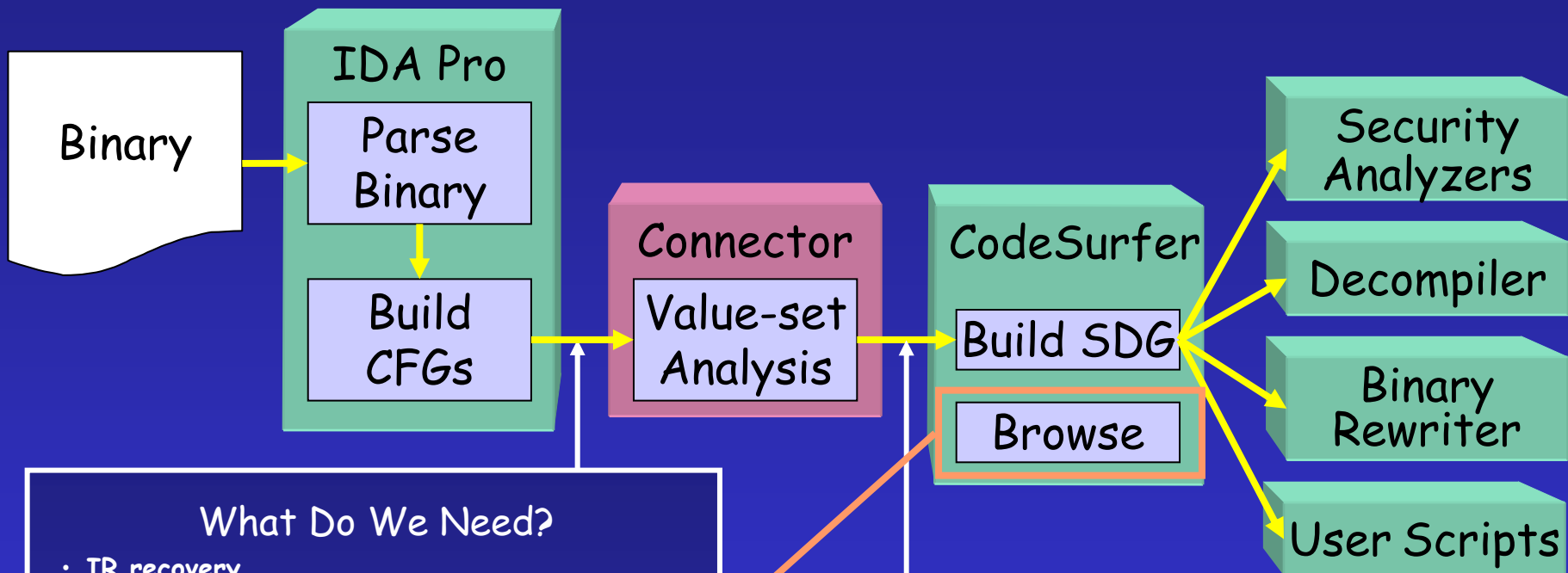


What Do We Need?

- **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
 - GUI to investigate warnings
- **Cooperation with dynamic tools**

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

CodeSurfer/x86 Architecture

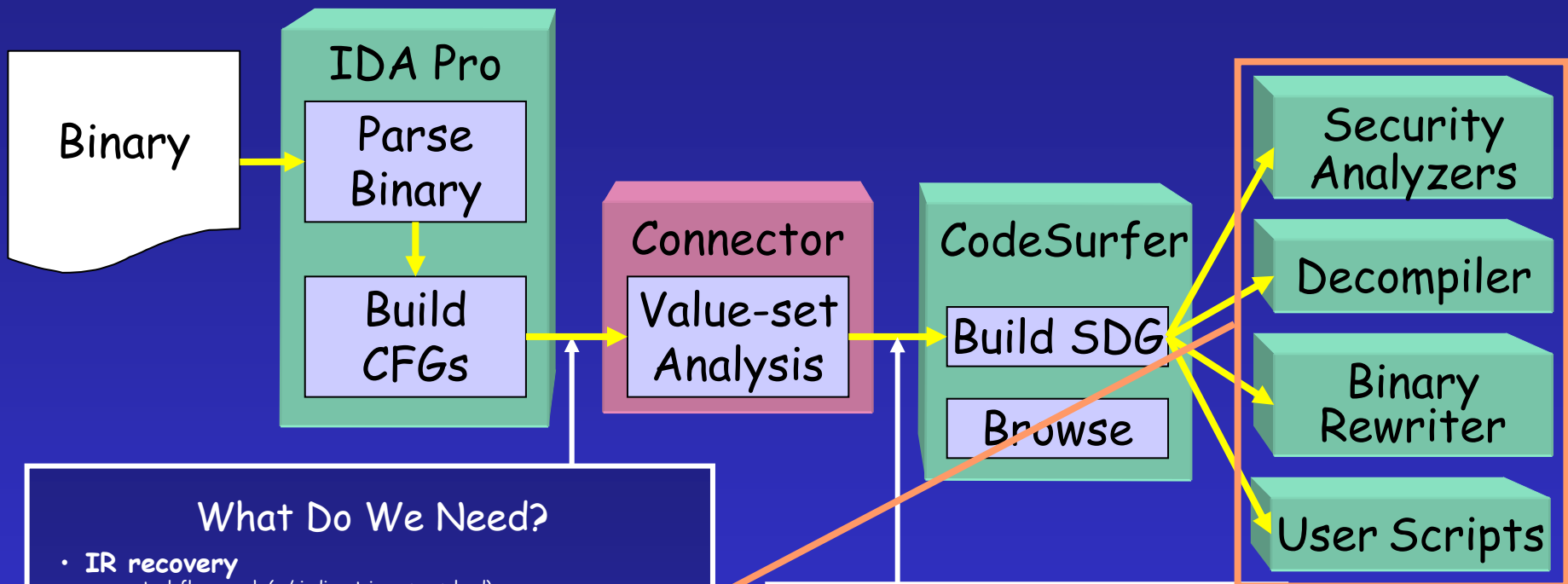


What Do We Need?

- **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
 - GUI to investigate warnings
- **Cooperation with dynamic tools**

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

CodeSurfer/x86 Architecture

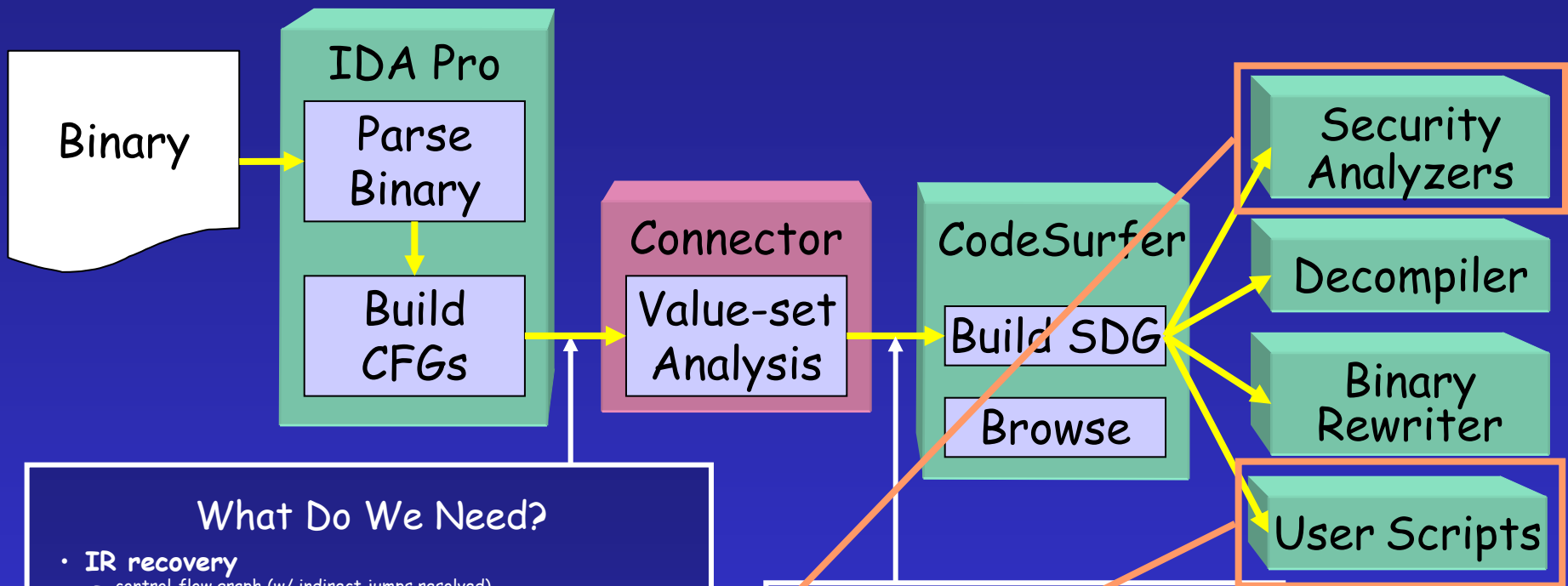


What Do We Need?

- **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
 - GUI to investigate warnings
- **Cooperation with dynamic tools**

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

CodeSurfer/x86 Architecture

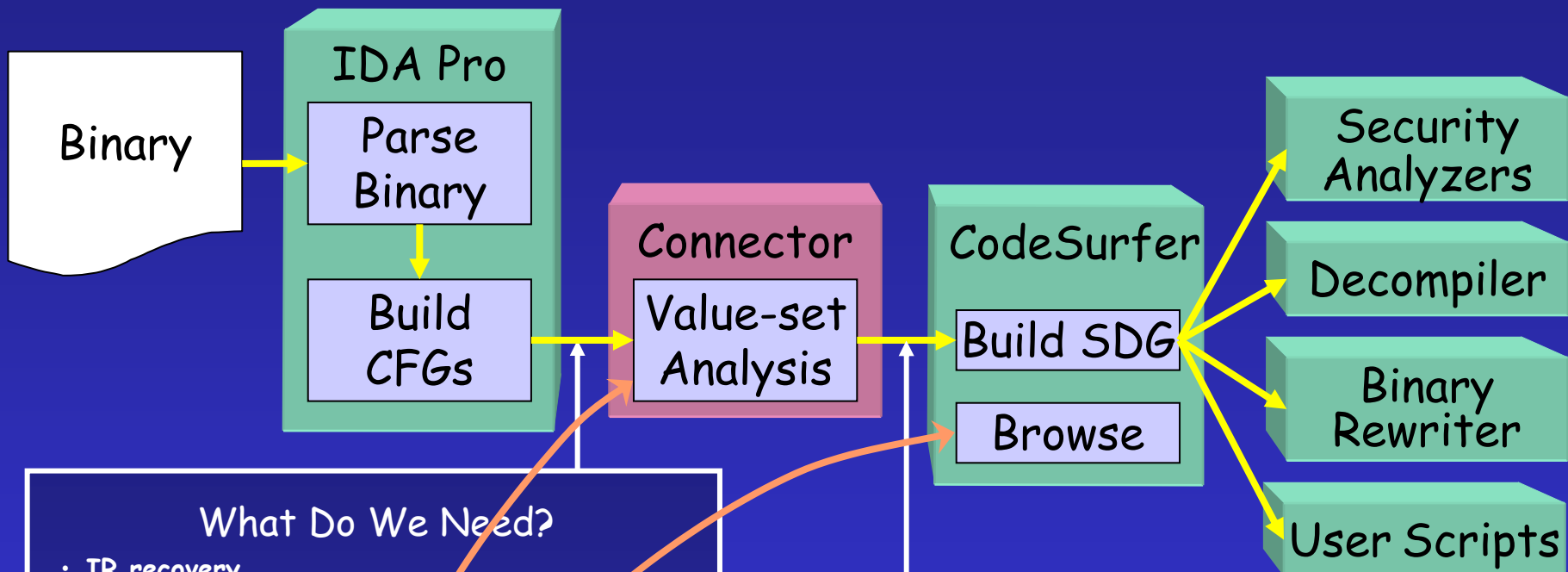


What Do We Need?

- **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
 - GUI to investigate warnings
- **Cooperation with dynamic tools**

- **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
 - GUI to investigate warnings
- **Cooperation with dynamic tools**

- **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

- Project at MIT Lincoln Labs (originally classified)
 - 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
 - . . .

Demo

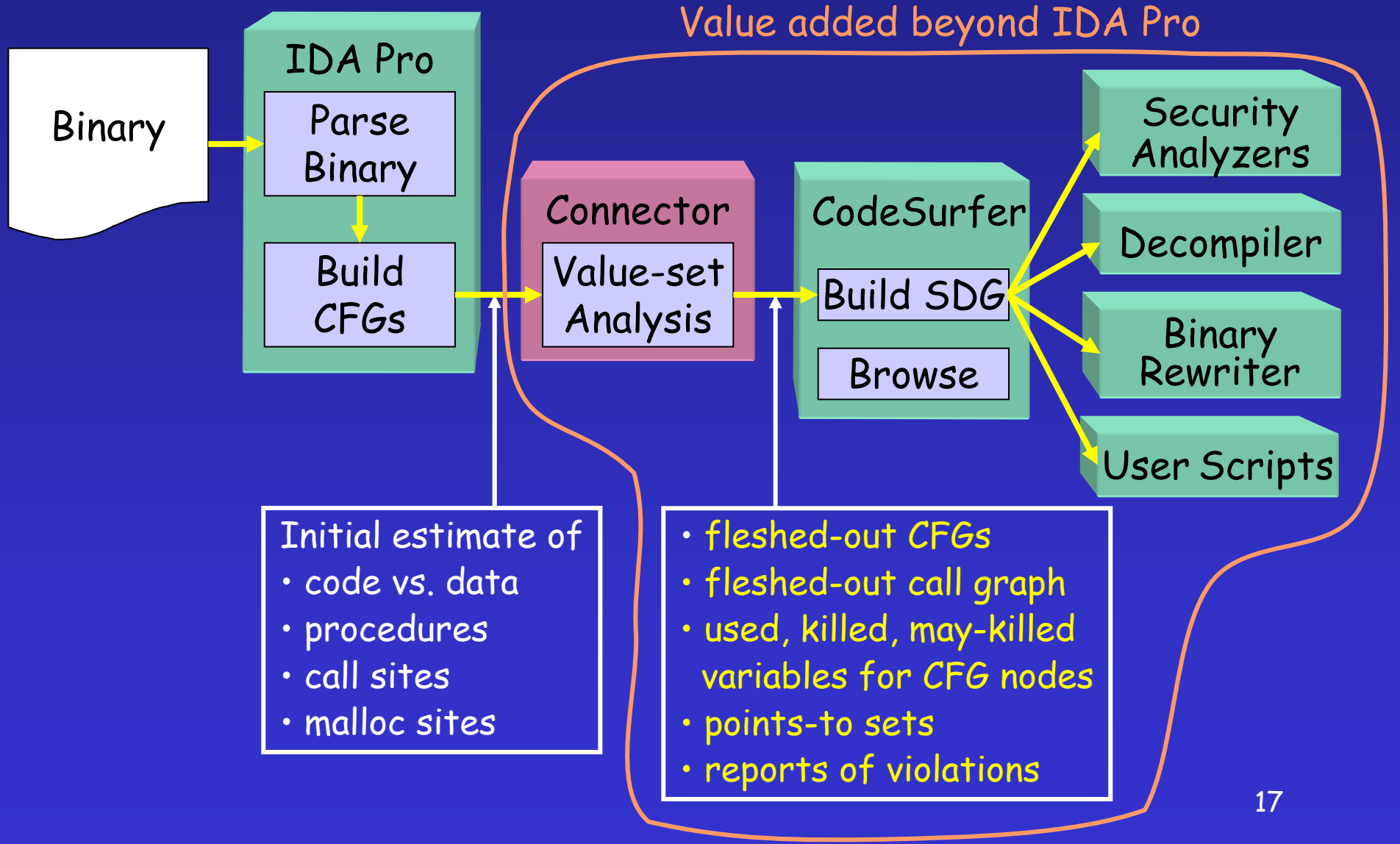
CodeSurfer/C



CodeSurfer/x86



CodeSurfer/x86 Architecture



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 model of the program
 - Invoke analyzer
- Path Explorer tool
 - Software-assurance plug-in to CodeSurfer/x86
 - Performs security-related analyses on the IR
 - Uses the GUI to investigate warnings

Why Executables?

- Reveals platform-specific choices made by compiler
- Makes visible other platform-specific artifacts
 - compiler bugs
- Some source-level issues go away
 - analyze the actual library code, not hand-written stubs
 - in-line assembly code
 - use of multiple source languages
- Better platform for finding security vulnerabilities
 - Source-code tools: Lack of fidelity can allow vulnerabilities to escape detection
- See <http://www.cs.wisc.edu/~reps/#cc04>

Balakrishnan and Reps, "Analyzing memory accesses in x86 executables" [CC04]
<http://www.cs.wisc.edu/~reps/#cc04>

Related Work

Debray et al., "Alias analysis of executable code" [POPL 98]

Cifuentes et al., "Assembly to high-level language translation" [ICSM 98]

A. Mycroft, "Type-based decompilation" [ESOP 99]

Linn et al., "Stack analysis of x86 executables" [Unpublished]

Guo et al., "Practical and accurate low-level pointer analysis" [CGO 05]

Amme et al., "Data dependence analysis of assembly code" [PACT 98]

Questions & Discussion

