The Deconstruction of Dyninst: Current, New, and Upcoming Components

Bill Williams
Matthew LeGendre
Nate Rosenblum

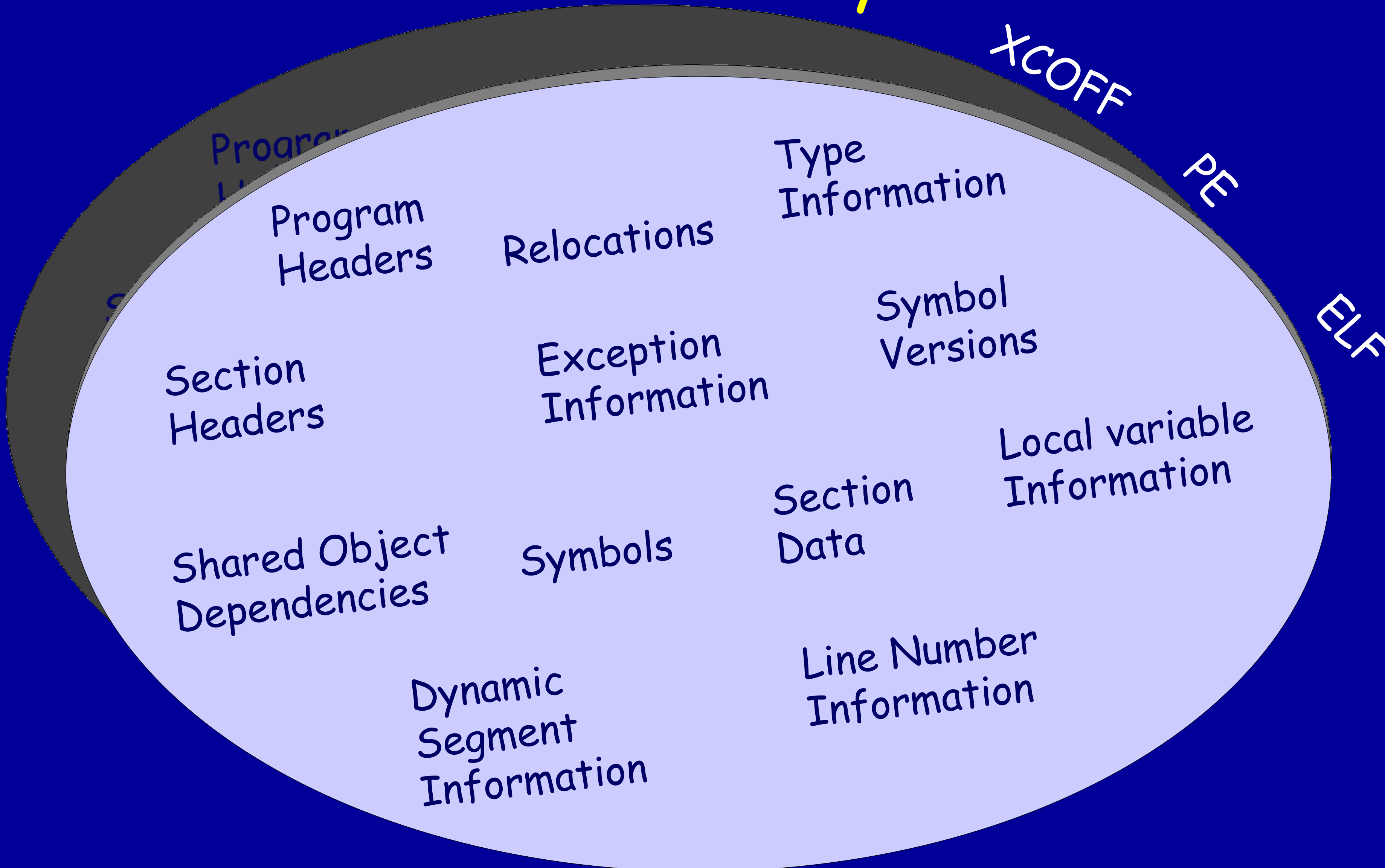
University of Wisconsin
http://www.paradyn.org

DyninstAPI and The Components

Talk overview

- Current Components
  - SymtabAPI
  - StackwalkerAPI
  - InstructionAPI
- Coming Soon
  - DepGraphAPI
  - FlowGraphAPI
- On The Horizon
  - Process Control
  - Binary Patching
  - Code Generation

The Binary


XCOFF
PE
ELF
Program Headers
Relocations
Type Information
Symbol Versions
Section Headers
Exception Information
Local variable Information
Section Data
Shared Object Dependencies
Symbols
Line Number Information
Dynamic Segment Information

Motivation

- Binaries are increasingly complex
  - Different formats
  - Lots of information
- Lack of portability
- Need for a tool that provides a simple view of binaries on different platforms.

SymtabAPI - Overview

- A multi platform library for parsing object files
- Goals -

Abstraction

• Be file format independent

Interactivity

• Update data incrementally

Extensibility

• User-extensible data structures

Generality

• Parse ELF/XCOFF/PE object files

• On-Disk/ In-Memory parsing

SymtabAPI: Updates

- Moving to version 6.0
  - Synchronizing version to DyninstAPI 6.0
- New variable and function abstractions
- Extended output interface
- Stackwalking debug information

Functions and Variables

- Old: One `Symbol` object aggregates info from all symbols.

Symbol: `malloc`
`__libc_malloc`
`__malloc`

→ Param Information
→ Local Variables
→ …

- New: One `Function` object has function info and multiple symbols/names.

Function

- Symbol: `malloc`
- Symbol: `__libc_malloc`
- Symbol: `__malloc`

→ Param Information
→ Local Variables
→ …

Write Interface

- Infrastructure for editing file format info and emitting new binaries.
- Insert and Edit:
  - Symbols
  - Relocations
  - Sections
- Binary Instrumentation and Editing still part of core DyninstAPI.

Stackwalking Debug Info

- Given a code pointer, describes how to locate the stack frame base.

Code Address	Expression Evaluating to the Frame Base
0x8048700	%esp + 4
0x8048714	%esp+%rax
0x8048718	%ebp
0x804873c	%rax * 4 + 16
0x8048750	*(0x804900)

- `getRegValueAtFrame` – Given the machine state and an address, compute the value of the frame pointer at that address.

StackwalkerAPI

- Stackwalking used for
  - Performance analysis sampling
  - Path profiling
  - Dynamic instrumentation
  - Debugging

- A library for walking call stacks
  - Accurate
  - Portable
  - Extensible

Stackwalking Challenges

- Stacks can contain:
  - Regular frames
  - Optimized frames
  - Signal handlers
  - System calls
  - Instrumentation
  - Uninitialized frames
  - …
- Difficult to recognize what kind of frame we're in.

Stackwalker Interface

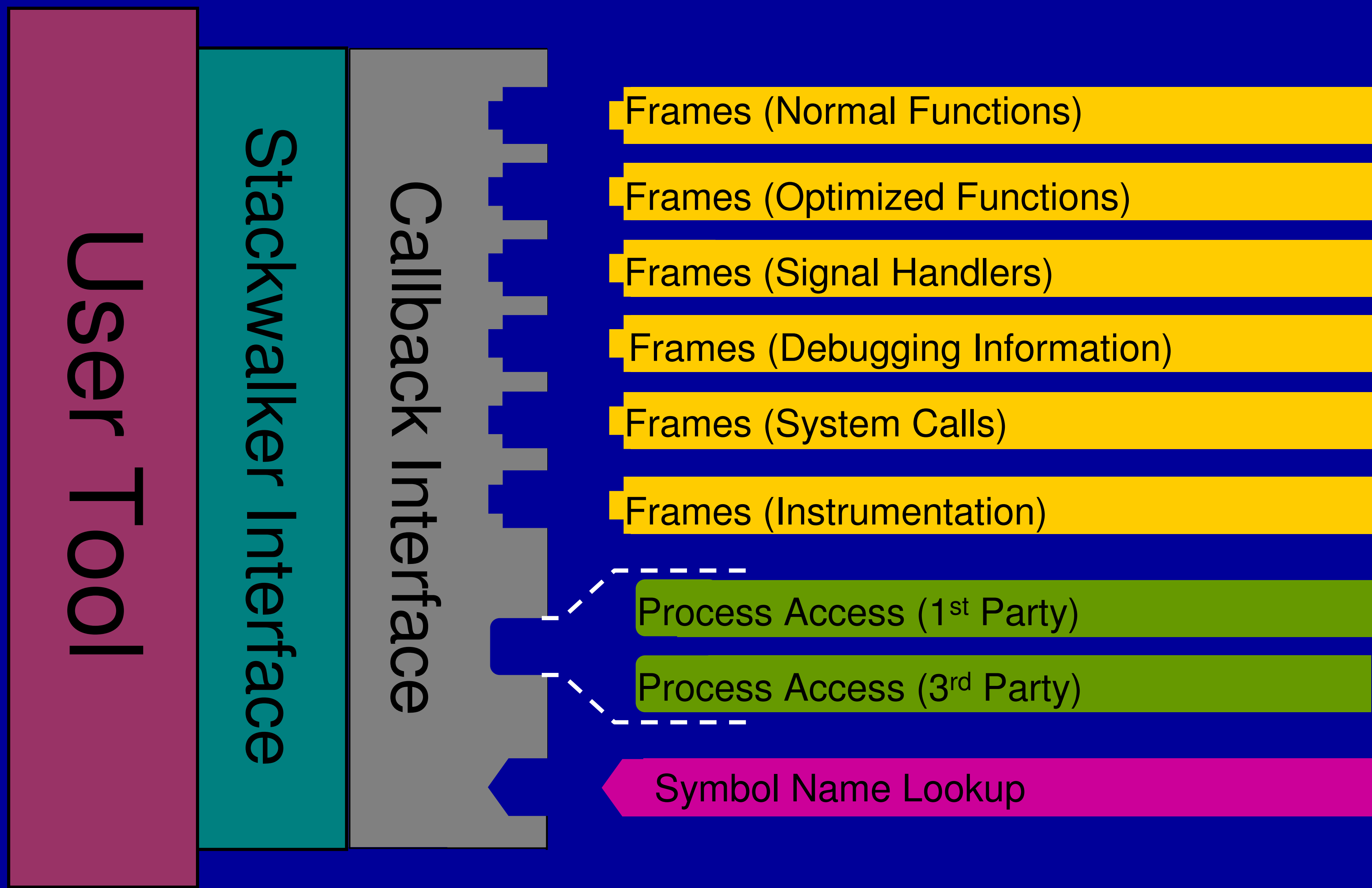
- Collecting a 3rd party stackwalk, with symbols

```
Walker *walker = Walker::newWalker(PID);
std::vector<Frame> stackwalk;

walker->walkStack(stackwalk);

std::string s;
for (int i=0; i<stackwalk.size(); i++) {
    stackwalk[i].getFuncName(s);
    cout << s;
}
```

Callback Interface



Stackwalker: Overview

- Initial 1.0 release
  - First party & third party stackwalks
  - Walk through
    - Normal Frames
    - Uninitialized Frames
    - Signal Handlers
    - Optimized Frames
    - Frames with Debugging Information
  - Multithreaded Applications

SymtabAPI & StackwalkerAPI: Technology Transfers

- HPCToolkit at Rice – SymtabAPI collects information to supplement parsing, writing symbol information to binaries.
- LLNL – StackwalkerAPI and SymtabAPI used with STAT and PNMPI projects.
- O|SS – Using SymtabAPI to collect function and symbol information.
- Cray – StackwalkerAPI and SymtabAPI used with APT.

InstructionAPI Motivation

- Many binary analysis concepts are architecture-independent
- Many binary analysis tools are tied to a specific architecture
- If we had an instruction model that let us do analysis in a platform-independent manner…

InstructionAPI: Overview

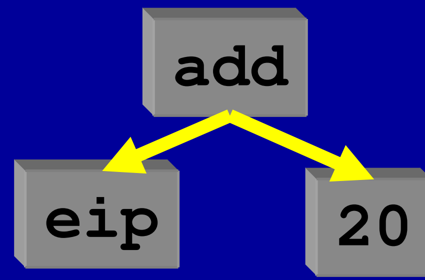
- Feature set targeted towards binary analysis
- Instructions are *operations* over multiple *operands*
- Operations:
  - Abstract categories
  - Detailed mnemonics
- Operands:
  - AST representation
  - Interactive evaluation

InstructionAPI: Operand Evaluation

- Many analyses only care about a small number of instruction types
  - Parsing: branches, calls, returns
  - Stack height: push, pop, assignment to SP
- Each of these requires details of operands
  - Parsing: branch destination
  - Stack height: value of SP afterwards
- Operand evaluation gives you this information
  - Once you fill in the context

InstructionAPI: parsing example

`jmp 0x20`



Relative jump needs PC value to be evaluated

Substitute the address of the instruction for the PC

```
Expression::Ptr cft = insn.getControlFlowTarget();

cft->bind(eip, jumpInsnAddr);

worklist.push_back(cft->eval());
```

Talk overview

- Current Components
  - SymtabAPI
  - StackwalkerAPI
  - InstructionAPI
- Coming Soon
  - FlowGraphAPI
  - DepGraphAPI
- On The Horizon
  - Process Control
  - Binary Patching
  - Code Generation

Isolating Dyninst's Parsing

Dyninst Functionality

High-level Dyninst

instrumentation, process control, etc.

Low-level Dyninst

gritty binary code details

SymtabAPI, InstructionAPI

Map binary code to useful structures

CFG creation

Function lookup

Resolving indirect control flow

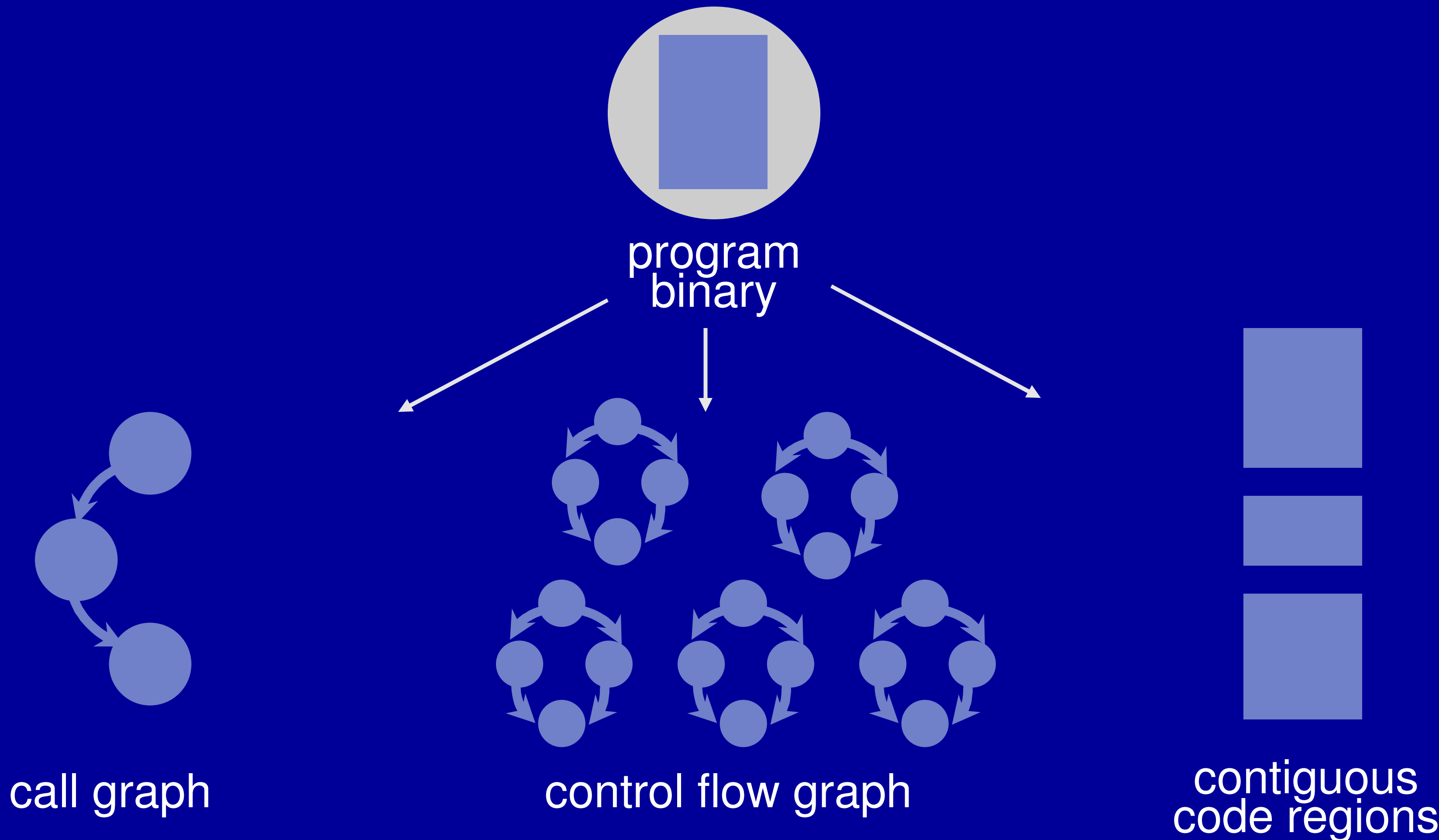
A parsing component?

ParsingAPI Features

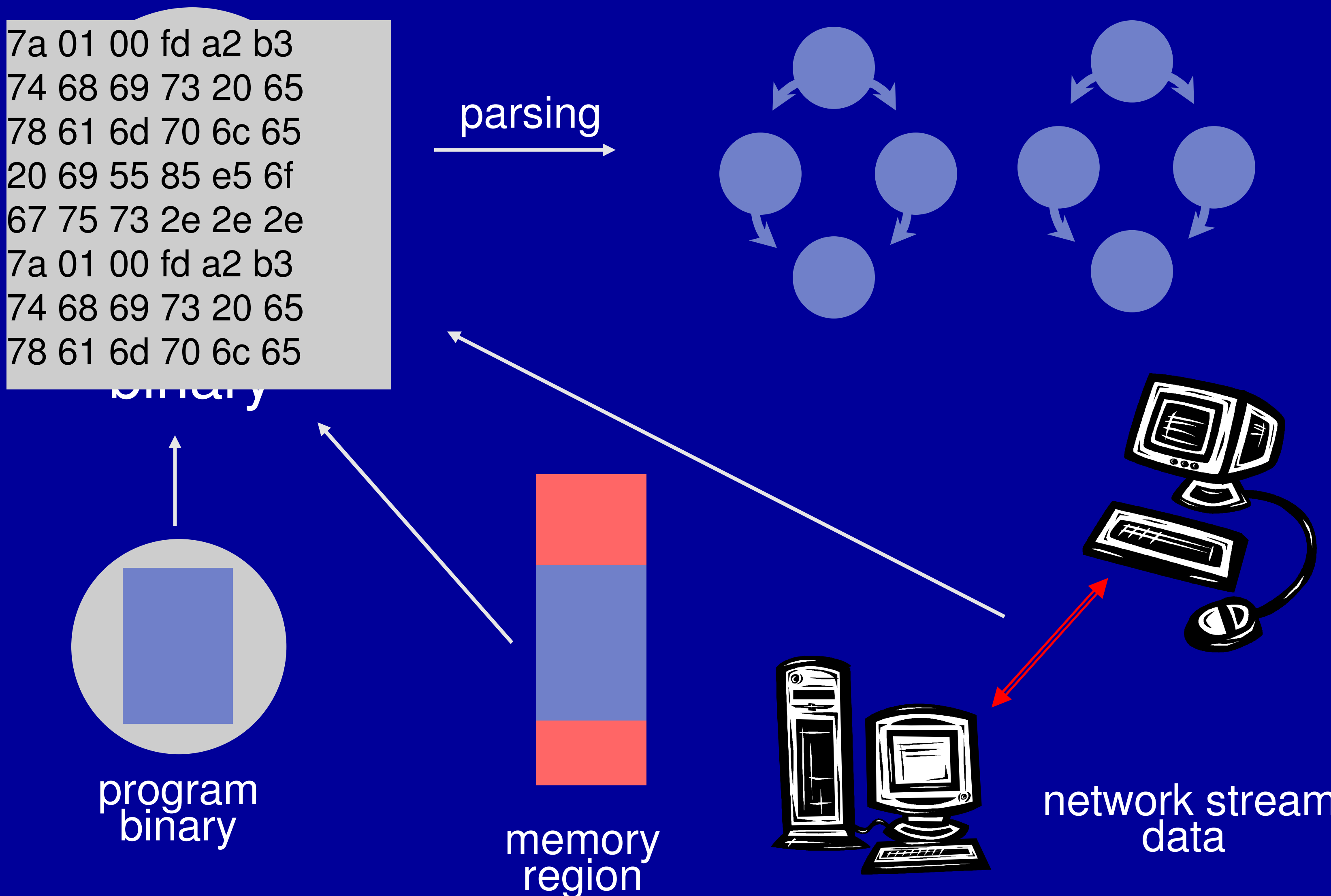
- Builds CFG from binary code
- Fine-grain lookup interface
  - functions, basic blocks, instructions
- Engineered to support "weird" binaries
  - e.g. self-modifying, "gappy"
  - easily updatable representations, "overlapping code"
- "Views"

Binary Code Views

Support multiple abstractions of code



Being Generic



DepGraphAPI: Motivation

- How does an input affect the program?
- How is the value of a variable determined?
- Is this function reachable?

These are questions about program dependence

Program Dependence: Another View

- Allows you to focus on relevant parts of the program
  - Determine which instructions are affected by (or affect) a particular instruction
  - Extract the program *slice* consisting of only these instructions
- Why do this?
  - Efficiency
  - Simplicity

DepGraphAPI Interface

- Standard graph abstractions
  - Nodes: operations performed by the program
  - Edges: dependencies between these operations
- Provides standard component benefits
  - Interactivity: update graph with more sophisticated analysis results
  - Extensibility: annotate graph elements with extra data
  - Platform independence

DepGraphAPI Beta

- Supports InstructionAPI platforms
  - Currently IA-32 and x86-64
- Intraprocedural analysis
  - Over BPatch_functions
  - Interprocedural is coming soon
- Memory analysis
  - Stack is treated as an array of variables
  - Heap is a single variable

Talk overview

- Current Components
  - SymtabAPI
  - StackwalkerAPI
  - InstructionAPI
- Coming Soon
  - FlowGraphAPI
  - DepGraphAPI
- On The Horizon
  - Process Control
  - Binary Patching
  - Code Generation

Process Control: Goals

- Develop an API to manage processes and runtime events
  - Pause and Continue process
  - Attach, Create, detach process
  - Reads and write to process' address space.
  - Receive notification of
    - Fork/exec
    - Thread create/destroy
    - Library load/unload
    - Signals
  - …
- Use OS's debugger interface.

Process Control: Why?

- Build 3rd party runtime tools
  - Alternative to building 1st party tools that operate on their own processes.

- Examples:
  - Collect variable values from several process
  - Follow fork/exec operations through a process tree
  - Single-step through a function to build an instruction trace.
  - Dynamic instrumentation

Bases for Implementation

- DyninstAPI implementation
  - Supports Linux, AIX, Solaris, Windows
  - Already has a working (but complex) threading model
  - Old and well tested
- StackwalkerAPI's debugger interface
  - Supports Linux, BlueGene
  - Simpler design
  - Already has a component interface
- Likely to build something that descends from both

Binary Patching & Code Generation

- Binary patching: insert, remove or modify binary code. E.g,
  - Insert instrumentation
  - Remove unwanted functions
  - Retarget memory operations

- Code generation: Compiler for high-level snippets into binary code.

Binary Patching Interface

- InstPoints as interface for instrumentation.
  - Work well for insertion, not for modification and removal.
- Editing views as interface for insertion, removal, modification.
  - Users edit the view
  - Dyninst transforms the code under the view

Code Generation

- Other projects have focused on compiling

- Can we use someone else's compiler in Dyninst?
  - LLVM
  - Dtrace
  - Rose
  - Gcc

- Need compatible licenses, supported platforms

Conclusions

- Current Components
  - SymtabAPI
  - StackwalkerAPI
  - InstructionAPI
- Coming Soon
  - FlowGraphAPI
  - DepGraphAPI
- On The Horizon
  - Process Control
  - Binary Patching
  - Code Generation

Conclusions

- Componentization is a community effort
  - LaunchMON from LLNL
  - HPCToolkit from Rice
- Tools can be quickly built from components
  - STAT
  - ATP
- The more you do it, the easier it is