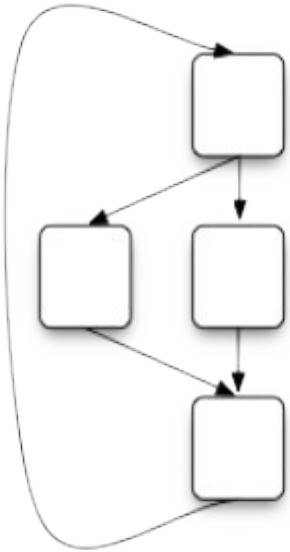
A Dyninst Primer
and
Project Updates

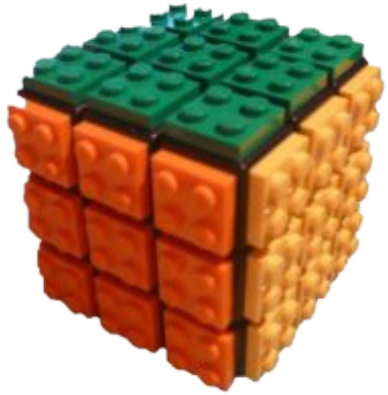
Josef (Bolo) Burger, James A. Kupsch, Tim Haines

Computer Sciences Department
University of Wisconsin

Scalable Tools Workshop
Granlibakken Resort
Lake Tahoe, California
June 19, 2023

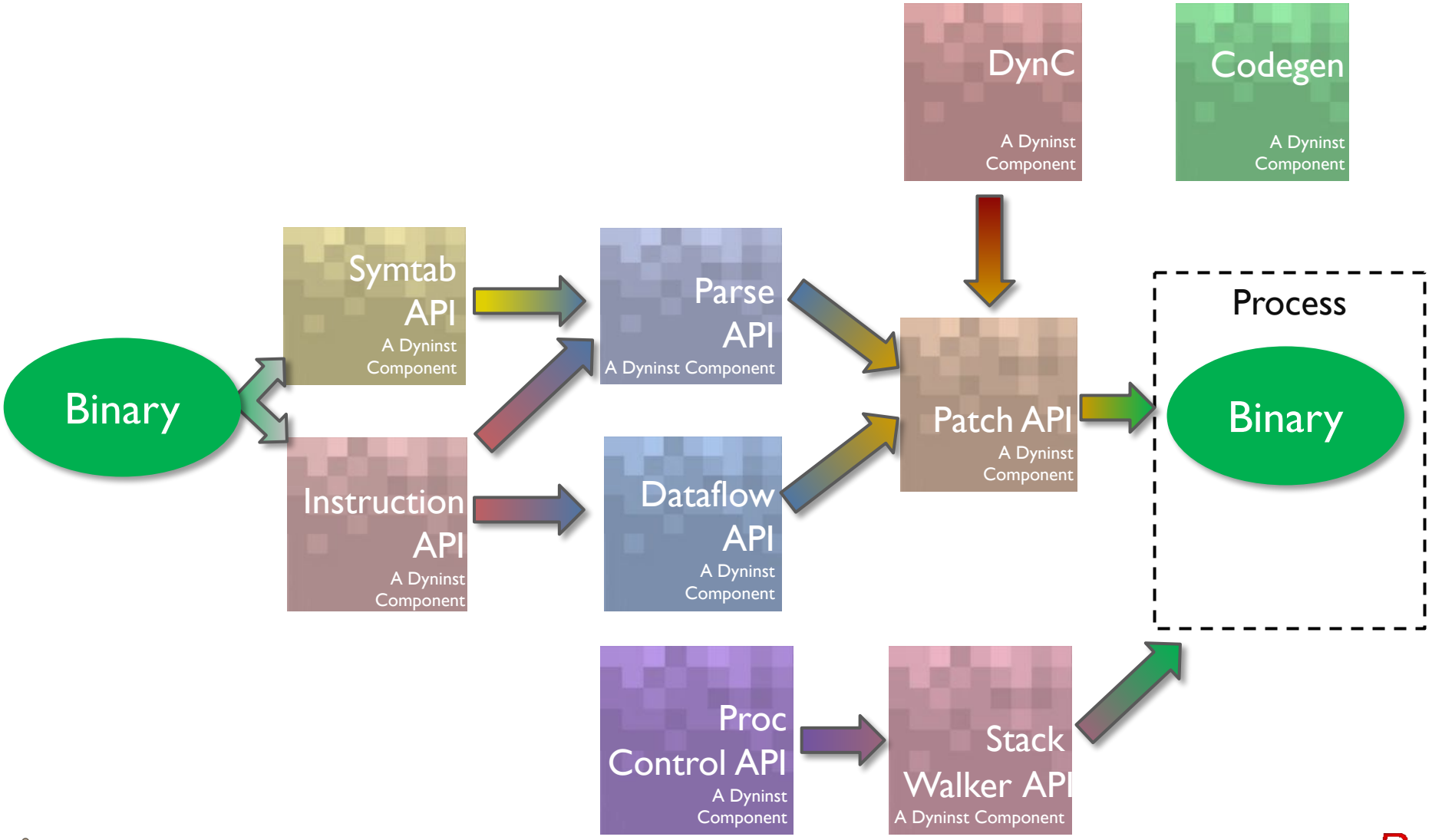
A Brief Introduction to Dyninst





Dyninst: a tool for binary analysis, static and dynamic instrumentation, modification, and control

Dyninst Components



Overview of Dyninst

An machine independent interface to machine level binary analysis, instrumentation and control.

- Control flow analysis, including basic blocks, loops, and functions. Produces intra- and inter-procedural CFGs
- Key abstraction is editing the CFG - not individual instrumentation replacement.
  - Enormously simpflies instrumentation
  - Closed under valid CFGs
- Dataflow analysis to support refined control flow analysis, register liveness and slicing.
- Dynamic: Modify running programs

Some of Dyninst's Capabilities

- Analysis of executables and libraries
  - Opportunistic: stripped, normal, and debug symbols.
- Instrumentation
  - Static: Rewrite binaries
  - Dynamic: Modify running programs
- Can instrument at almost any instruction
- Instrumentation code specified by AST's
- Platform independent process control

What you can do with Dyninst

Analysis

- find by name or address
  - functions
  - global variables
  - local variable
  - basic blocks
- analyze control flow
- analyze instructions
  - by operand expressions
  - by opcode
  - by type
- forward & backward slicing
- analyze loops

Instrumentation

- functions
  - entry
  - exit
  - call site
- loops
  - entry
  - exit
  - body
- branches
  - taken
  - not taken
- instructions

What you can do with Dyninst

Runtime features

- process control
- read/write process memory
- stack walking
- load library

Applications

- code coverage
- performance time/counts
- peephole optimizations
- find all memory accesses
- change program behavior
- fix bugs via patching
- examine call stack
- create call graph
- disassembly
- and more…

Dyninst - Analysis

Binary file:

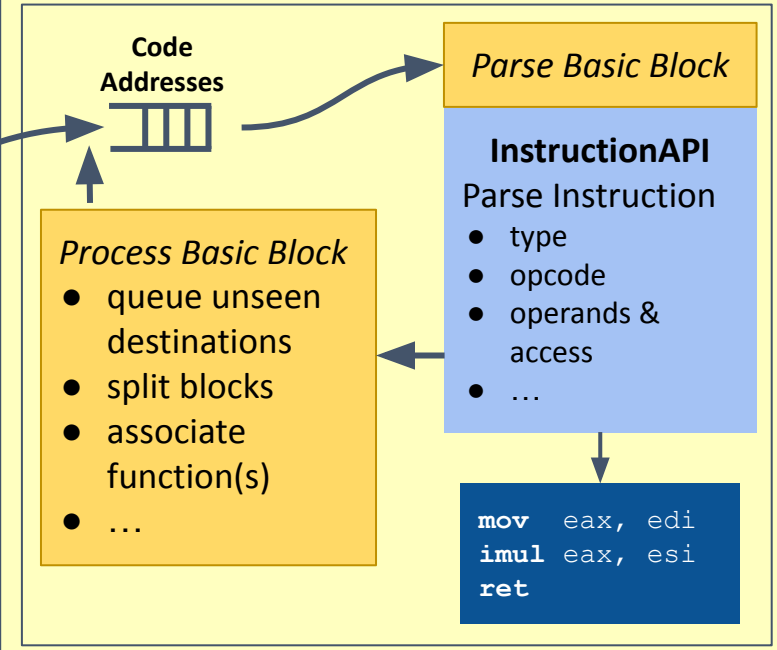
```
7a 77 0e 20 e9 3d e0 09 e8 68
c0 45 be 79 5e 80 89 08 27 c0
73 1c 88 48 6a d8 6a d0 56 4b
fe 92 57 af 40 0c b6 f2 64 32
f5 07 b6 66 21 0c 85 a5 94 2b
20 fd 5b 95 e7 c2 42 3d f0 2d
7a 77 0e 20 e9 3d e0 09 e8 68
c0 45 be 79 5e 80 37 1b 2f b9
```

SymtabAPI

Symbols
- functions
- variables
- types
- …

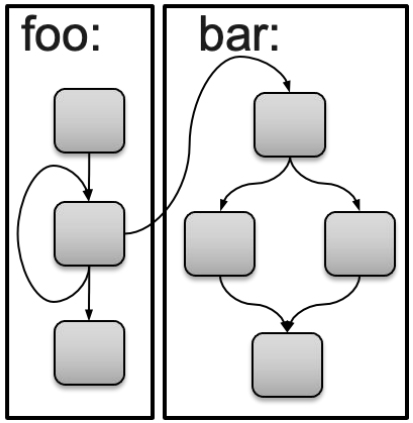
Binary Properties
- segments
- sections
- ELF properties
- …

ParseAPI



- parse code
- produce CFG
  - basic block nodes
    - straightline code
    - control flow only to beginning and from end of block
    - associated with functions(s)
  - control flow edges
    - from block to block
    - type: call, fallthrough, jump, branch taken, branch not taken, return, …
- loop analysis

Control Flow Graph



DataFlowAPI
- register liveness
- doward slicing - instructions affected by data
- backward slicing - instructions that affected data
- stack height analysis

Dyninst - Code Modification

- snippet - machine-independent AST of operations
  - read/write memory, registers, variables
  - basic math
  - function calls
  - conditional branches
  - jumps
  - …
- point - abstract location to modify CFG
  - function entry/exit
  - basic block entry/exit
  - memory writes
  - …
- snippet insertion - modification abstraction
  - modify CFG with snippet at point
  - generates machine specific code
  - can maintain existing code's semantics

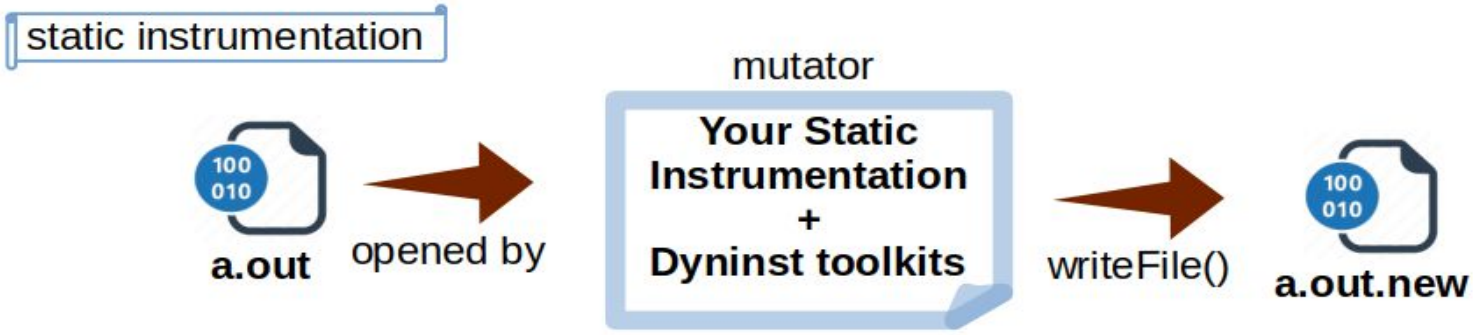
Dyninst - Process Control

- Debugger-like functionality
- Platform-independent interface
- Attach to running process
- Launch existing binary
- Suspend/resume process
- Catch user events like signals
- Detect user process and thread creations

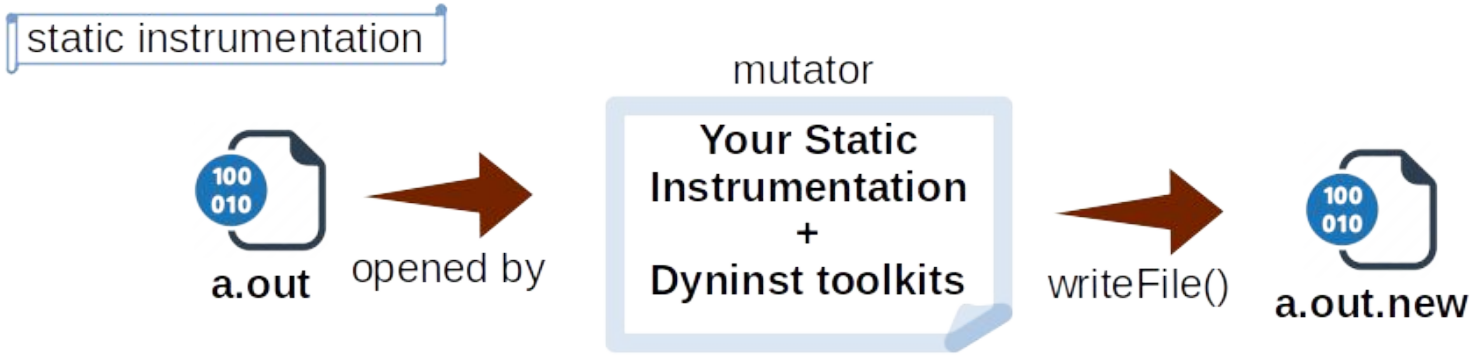
Instrumentation Techniques

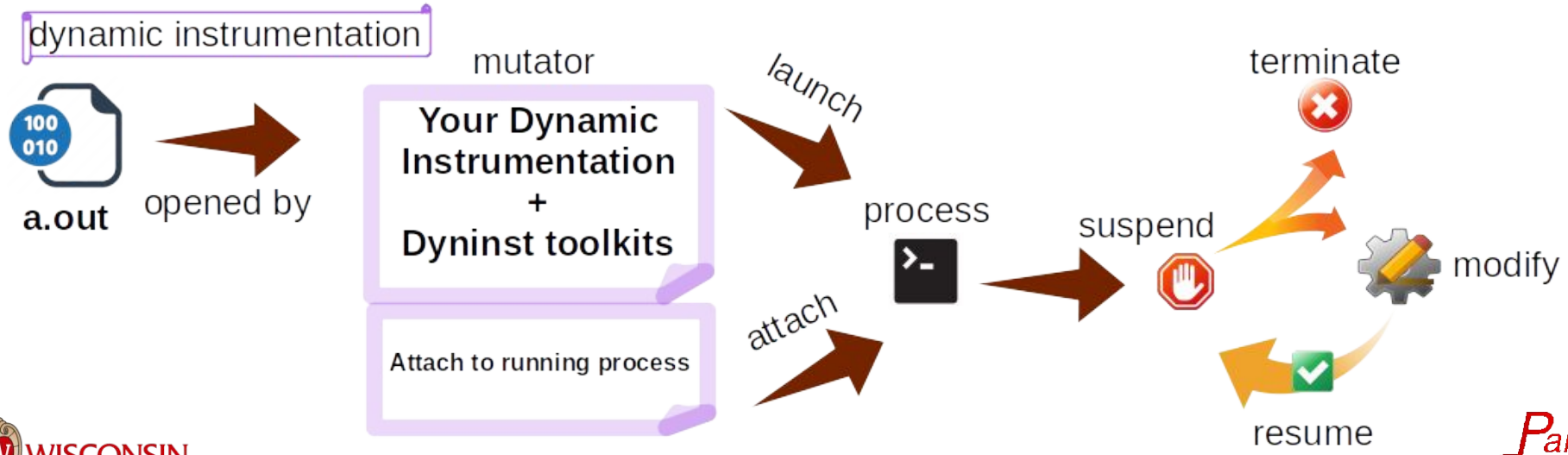
Technique	Source code required	Source modification	Recompile	Effort to Apply	Flexibility
User-inserted code	Y	Y	Y	Med to High	Limited by library
Tool-inserted e.g., gcov, gprof	Y	N	Y	Low to Med	Low
Dyninst - roll your own	N	N	N	High due to specialized knowledge	High
Dyninst - existing tool e.g., HPCToolkit	N	N	N	Low	Limited by library tends to be more flexible

Instrumentation in Dyninst



Instrumentation in Dyninst





Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

Example of Dyninst inserting entry/exit instrumentation into a function.

```
int add(int a, int b)
{
    return a + b;
}
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find the entry/exit points of the function

```
entry = add→findPoint(BPatch_locEntry);
exit  = add→findPoint(BPatch_locExit);
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find the entry/exit points of the function

```
entry = add→findPoint(BPatch_locEntry);
exit  = add→findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…);
```

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

Paradyn

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:
      call Trace
5fa:  push   %rbp
5fb:  mov    %rsp,%rbp
5fe:  mov    %edi,-0x4(%rbp)
601:  mov    %esi,-0x8(%rbp)
604:  mov    -0x4(%rbp),%edx
607:  mov    -0x8(%rbp),%eax
60a:  add    %edx,%eax
60c:  pop    %rbp
      call Trace
60d:  retq
```

libtrace.so

```
XXX <Trace>:
      …
…:  // trace functionality
      …
…:  retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find the entry/exit points of the function

```
entry = add→findPoint(BPatch_locEntry);
exit  = add→findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…);
```

7. Insert snippets

```
addrSpace→insertSnippet(traceExpr,entry);
addrSpace→insertSnippet(traceExpr,exit);
```

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:
     …
…: // trace functionality
     …
…: retq
```

Only minor modifications are needed to extend this example to:

- Basic Block Instrumentation
- Memory Tracing

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:
      call Trace
5fa:  push   %rbp
5fb:  mov    %rsp,%rbp
5fe:  mov    %edi,-0x4(%rbp)
601:  mov    %esi,-0x8(%rbp)
604:  mov    -0x4(%rbp),%edx
607:  mov    -0x8(%rbp),%eax
60a:  add    %edx,%eax
60c:  pop    %rbp
      call Trace
60d:  retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find the entry/exit points of the function

```
entry = add→findPoint(BPatch_locEntry);
exit  = add→findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…);
```

7. Insert snippets

```
addrSpace→insertSnippet(traceExpr,entry);
addrSpace→insertSnippet(traceExpr,exit);
```

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Basic Block Entry/Exit Instrumentation

```
0000000000005fa <add>:
      call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
      call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find the entry/exit points of the basic blocks

```
add→getCFG()→getAllBasicBlocks(blocks);
for(auto block : blocks) {
  entry.push_back(block→findEntryPoint())
  exit.push_back(block→findExitPoint());
}
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…);
```

7. Insert snippets

```
addrSpace→insertSnippet(traceExpr,entry);
addrSpace→insertSnippet(traceExpr,exit);
```

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

Load/Store Operations Instrumentation

```
0000000000005fa <add>:
      call Trace
5fa:  push   %rbp
5fb:  mov    %rsp,%rbp
      call Trace
5fe:  mov    %edi,-0x4(%rbp)
      call Trace
601:  mov    %esi,-0x8(%rbp)
      call Trace
604:  mov    -0x4(%rbp),%edx
      call Trace
607:  mov    -0x8(%rbp),%eax
60a:  add    %edx,%eax
      call Trace
60c:  pop    %rbp
      call Trace
60d:  retq
```

libtrace.so

```
XXX <Trace>: …
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library containing the function you want to call at entry/exit

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace→findFunction("Trace");
```

5. Find load/store operations in functions

```
std::set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);
lsp = add→findPoint(axs);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…);
```

7. Insert snippets

```
addrSpace→insertSnippet(traceExpr,lsp);
```

What is new since June 2022?

- GPU Enhancements
- Bug Fixes and Enhancements
- New Features
- Code Cleanup
- Updates to CMake and testing
- Future Work

Enhancements - AMD GPUs

- MI100 (CDNA) Instruction Parsing
  - instruction decoder generated from AMD XML specs
- Symbol Parsing
- Direct Control Flow Analysis
- Updated MI200 (CDNA2) instruction decoding from latest XML specs

Bug Fixes & Enhancements

- Cross-architecture instruction interpretation is now correct
  - e.g., analyze ARM binary on x86
- Fixed parsing of DWARF location lists for local variables
  - Allows determining the location of a variable given an instruction location
- Finding name of called functions vastly sped up
  - Omnitrace parsing of PETSc: 30mins -> 3mins
  - Code provided by AMD Research (Thank you!)

New Features

- Basic CFG creation can be performed without full dataflow instruction semantics
  - Allows creating a CFG for ISAs we don't yet have dataflow information for yet
- Callback to allow decoding of unknown instructions by the user
- Support new ctor/dtor handling in static glibc
- DWARF-5 type support
  - enum classes
  - C++ references

Code Cleanup

- Compile cleanly with gcc6-13 and clang7-15
  - C++11, 14, 17, 20 and 23
  - C11 and 17
- Fixed 100's ***more*** real issues and 1000's of potential issues
- Dozens bugs from static analysis with cppcheck
  - integer overflow
  - resource leaks
- Correctly handle unaligned memory access across platforms

Building and Testing

- Ground-up rewrite of CMake to use modern target-based conventions. Using Dyninst is easier than ever!
- Expanded CI testing
  - for individual PRs
  - Build with multiple compilers
- Now support *all* versions of TBB since 2019

Future Work

- Static instrumentation for AMD GPU binaries
- Add instruction semantics for dataflow on AMD GPUs
  - To be provided by AMD XML Specs
- Support for more GPUs and CPUs, such as AMD MI300
- Move to Capstone for x86 instruction parsing
  - Provides complete coverage for x86_64
  - Enhance AVX-512 handling
  - AMD Zen 4 “Genoa”, Sapphire Rapids/AMX

Partners and Collaborators

Thanks to significant contributions from:

- John Mellor-Crummey and Mark Krentel (Rice)
- Matt Legendre (LLNL)
- Ben Woodard, Stan Cox, Will Cohen (Red Hat)
- Timour Paltashev, Jonathan R. Madsen (AMD)
- Xiaozhu Meng (Rice, now Amazon)
- Rashawn Knapp (Intel)

Who Uses Dyninst?

Some of our Dyninst users:

- HPCToolkit (Rice)
- SystemTap (Red Hat)
- AMD & Omnitrace (AMD Research)
- stat (LLNL)
- ATP (Cray/HPE)
- Common Tools Interface (Cray/HPE)
- Open|SpeedShop

We can't track all the uses of Dyninst; hundreds of papers have been published mentioning Dyninst use. If you're not on our list, please let us know!

Questions?

https://github.com/dyninst/dyninst