A Dyninst Primer
and
Project Updates

James A. Kupsch

Computer Sciences Department
University of Wisconsin

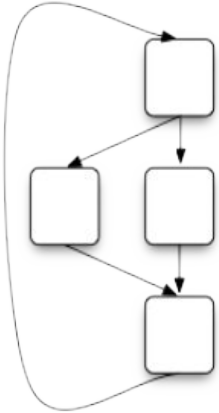
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

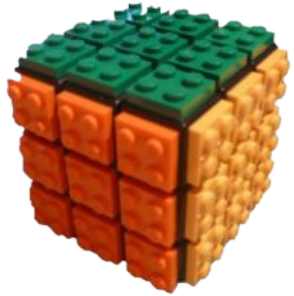
Scalable Tools Workshop

July 27, 2026

Paradyn

A Brief Introduction to Dyninst





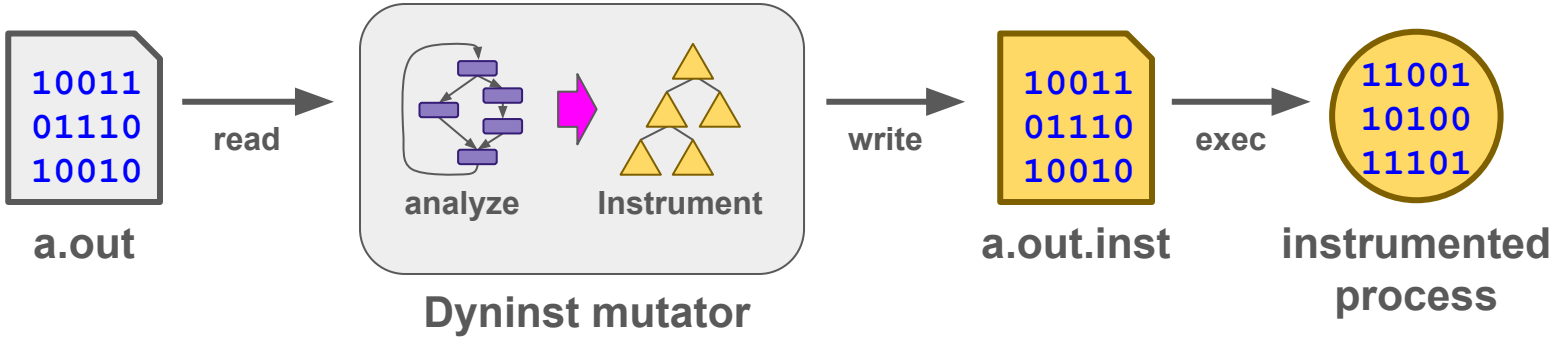
Dyninst: a tool for binary analysis, static and dynamic instrumentation, modification, and control

Overview of Dyninst

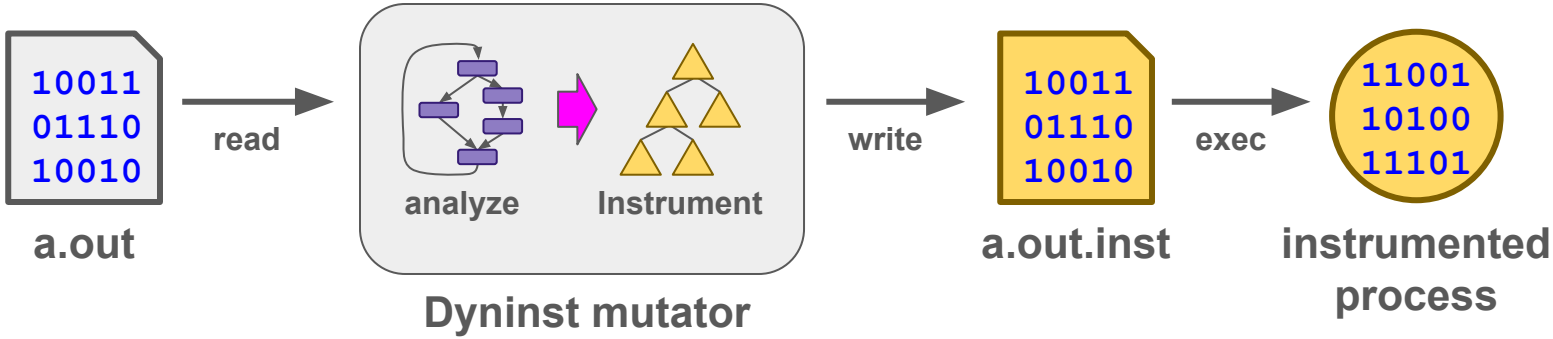
An machine independent interface to machine level binary analysis, instrumentation and control.

- Control flow analysis produces intra- and inter-procedural control flow graphs (CFGs) with basic blocks, loops, and functions
- Dataflow analysis supports refined control flow analysis, register liveness and slicing
- Key abstraction is editing the CFG - not individual instruction replacement.
  - Enormously simpflies instrumentation
  - Closed under valid CFGs
- Static and Dynamic: Modify executable/libraries and running programs

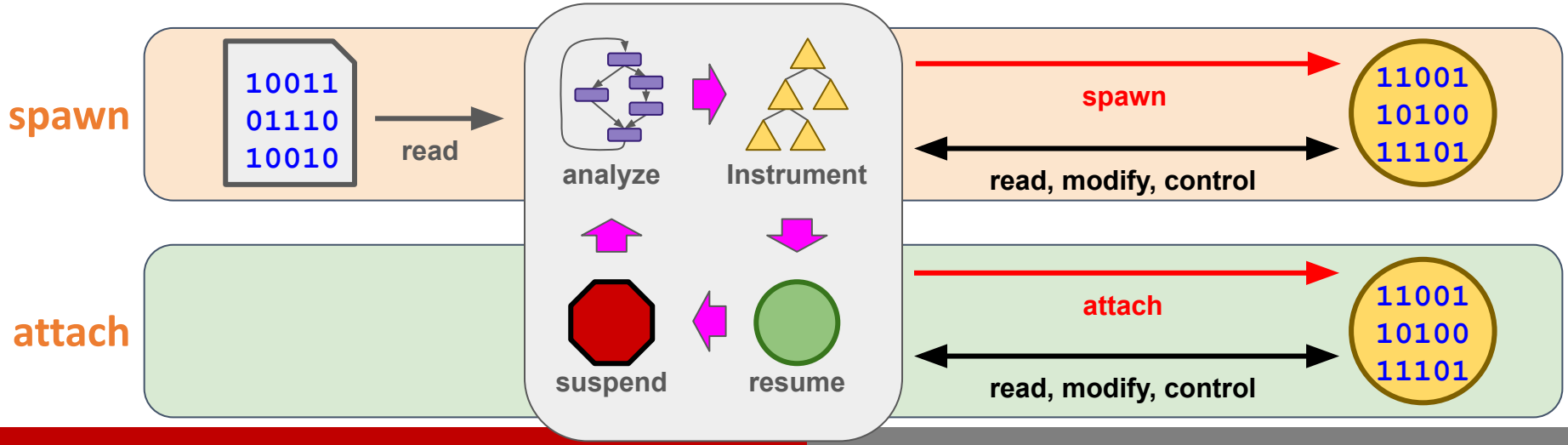
Static Instrumentation



Static Instrumentation



Dynamic Instrumentation



Some of Dyninst's Capabilities

- Analysis of executables and libraries
  - Opportunistic: stripped, normal, and debug symbols.
- Instrumentation code specified by AST's
- Can instrument any location in the CFG or almost any instruction
- Instrumentation
  - Static: Rewrite binaries
  - Dynamic: Modify running programs
- 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
- jump table analysis
- forward & backward slicing
- loop analysis

Instrumentation

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

What you can do with Dyninst

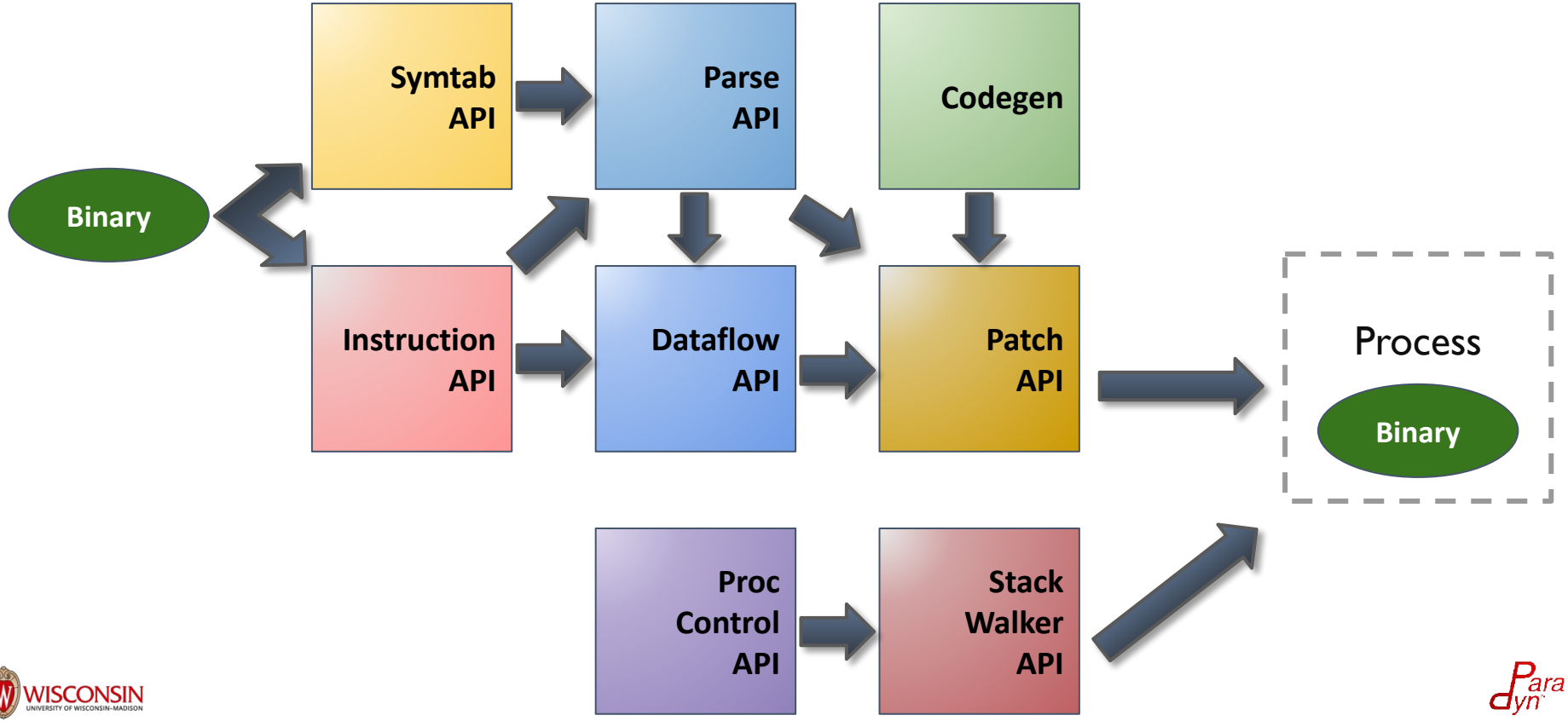
Runtime features

- 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 Components



Dyninst - Analysis

Binary file or running process:

```
7a 77 0e 20 e9 3d e0 09 e8
68 c0 45 be 79 5e 80 89 08
27 30 73 1c 88 48 6a d8 5a
d0 56 4d 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
09 e8 68 c0 45 be 79 5e 37
```

SymtabAPI

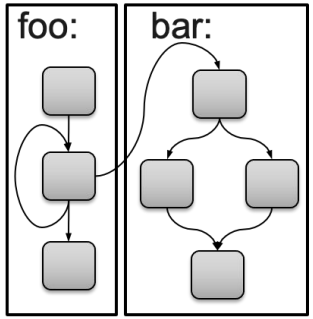
Symbols
- functions
- variables
- types
- …

Binary Properties
- segments
- sections
- ELF properties
- …

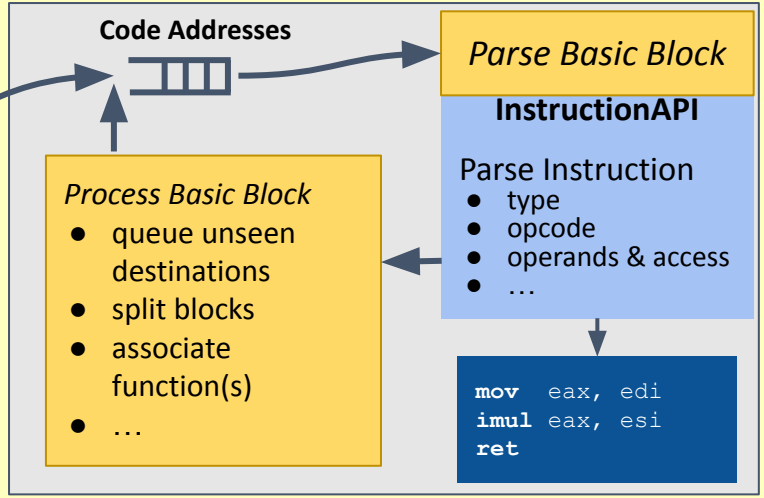
DataFlowAPI

- register liveness
- forward slicing - *instructions affected by data*
- backward slicing - *instructions that affected data*
- stack height analysis
- loop analysis

Control Flow Graph



ParseAPI



- parse code
- produce CFG
  - basic block nodes
    - straightline code
    - associated with functions(s)
  - control flow edges
    - from block to block
    - type: call, fallthrough, jump, branch taken, branch not taken, return, …
- jump table 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
- maintains 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

Function Entry/Exit Instrumentation

```
00000000000005fa <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;
}
```

compiles to

Function Entry/Exit Instrumentation

```
00000000000005fa <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

```
00000000000005fa <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

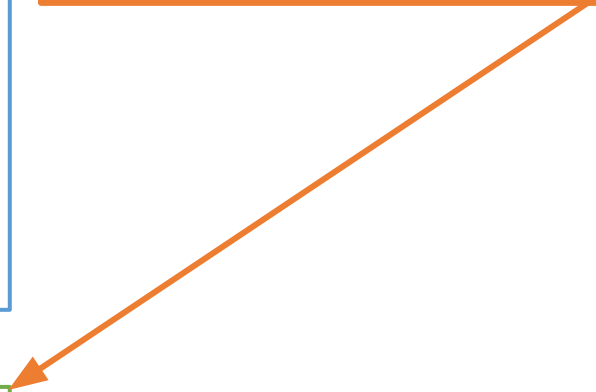
```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

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");
```



Function Entry/Exit Instrumentation

```
00000000000005fa <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

```
00000000000005fa <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

```
00000000000005fa <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

```
00000000000005fa <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,…);
```

Function Entry/Exit Instrumentation

```
00000000000005fa <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);
```

Function Entry/Exit Instrumentation

```
00000000000005fa <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

```
00000000000005fa <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);
```

Basic Block Entry/Exit Instrumentation

```
00000000000005fa <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 all 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);
```

Load/Store Operations Instrumentation

```
00000000000005fa <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 the load/store instructions in the function**

```
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's new since July, 2025

Changes	**294** commits **1419** files changed	**66967** insertions **48935** deletions
Outside Code Contributors	• Tim Haines (Kitware) • Jonathon Anderson (Rice) • Mark Krentel (Rice)	• Kian Cossettini (AMD) • Marco Bartoli (UPC) • Lazar Aleksic • William Allen (Kitware)
Collaborators	• HPCToolkit Project • TAU Project • AMD	

CFG & Performance Improvements

CFG Improvements:

- .cold clones are not functions
- correctly classify non-returning trap-like instructions
- correct symbolic execution of several instructions
- removed duplicate edges for some indirect control flow blocks

Performance Improvements:

- loop analysis - big improvement for complex CFGs (7x libmpi) with 6% average improvement
- gap parsing - 2400x libtorch_cpu.so, 500x libtorch_python
- generally better due to CFG improvements

AMD GPU improvements (WIP)

- Initial code generation support
- Investigating refining CFGs for SIMD/SIMT architectures
- Added snippet ordering support
- Register pair support
- Developed strategy for GPU instrumentation variables
- Developed mechanism to make GPU to Host RPCs

more details in Hsuan-Heng's talk on Tuesday

RISC-V updates

- Merged support for analysis components:
  - SymtabAPI
  - InstructionAPI
  - ParseAPI/Dataflow
- Have branches to review and merge:
  - BPatch/DyninstAPI/codegen
  - Stackwalker
  - Proccontrol

Miscellaneous Updates

- fixed proccontrol to handle threads spawned before main (AMD)
- fixed determination of leaf functions
- ELF changes
  - removed assumption of program header order in finding the load_vaddr
  - support RELR relocation types (from AMD)
  - fixed ELF rewriting bug that failed to add Dyninst generated sections if there was a gap between ELF segments (libtorch_python.so)
- migrated final 2 manuals in MS Word format to LaTeX
- Removed tbbmalloc_proxy as it caused crashes for allocation made before Dyninst
- lots of clean-up, small bugs fixes
- improves testing
- refactoring to eventually support future cross architecture code gen

Future

- merge final RISC-V PRs
- release Dyninst 14.0
- update Dyninst's C++ standard: 11 → 20
- reduce Dyninst's use of Boost Libraries (AMD
- further AMD GPU instrumentation support
- replace instruction hand-crafted decoder: use combinations of capstone and Zydis

Questions?

https://github.com/dyninst/dyninst

dyninst-announce+subscribe@g-groups.wisc.edu