A Dyninst Primer
and
Project Updates

James A. Kupsch, Angus He

Computer Sciences Department
University of Wisconsin

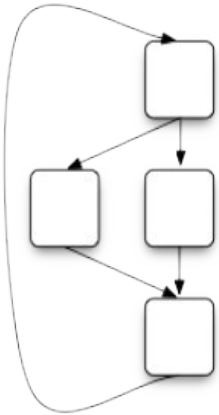
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

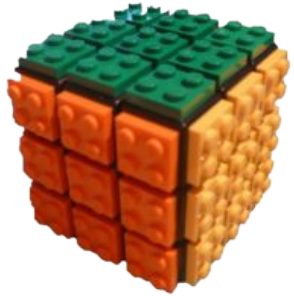
Scalable Tools Workshop

July 6, 2025

Paradyn

A Brief Introduction to Dyninst





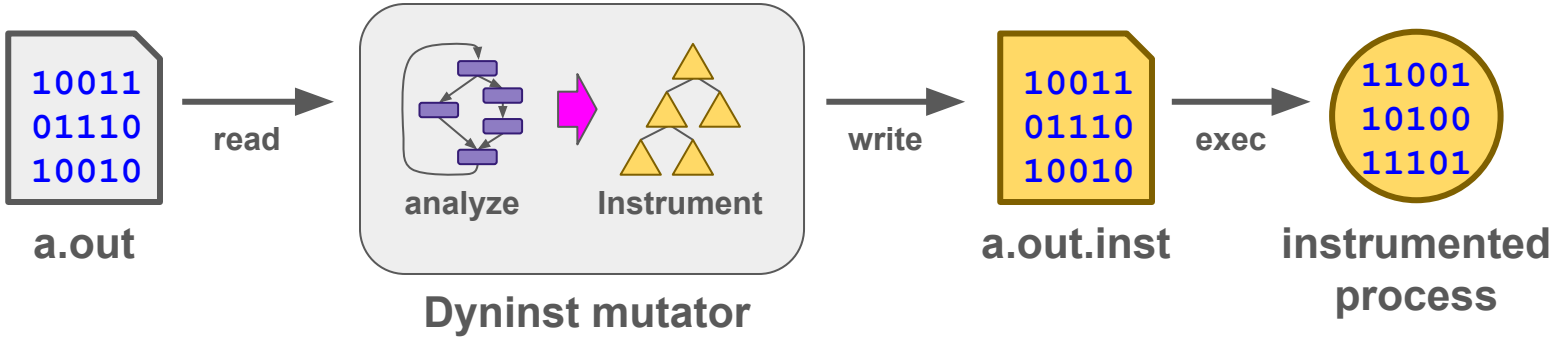
Dyninst: a tool for binary analysis, static and dynamic instrumentation, modification, and control

Overview of Dyninst

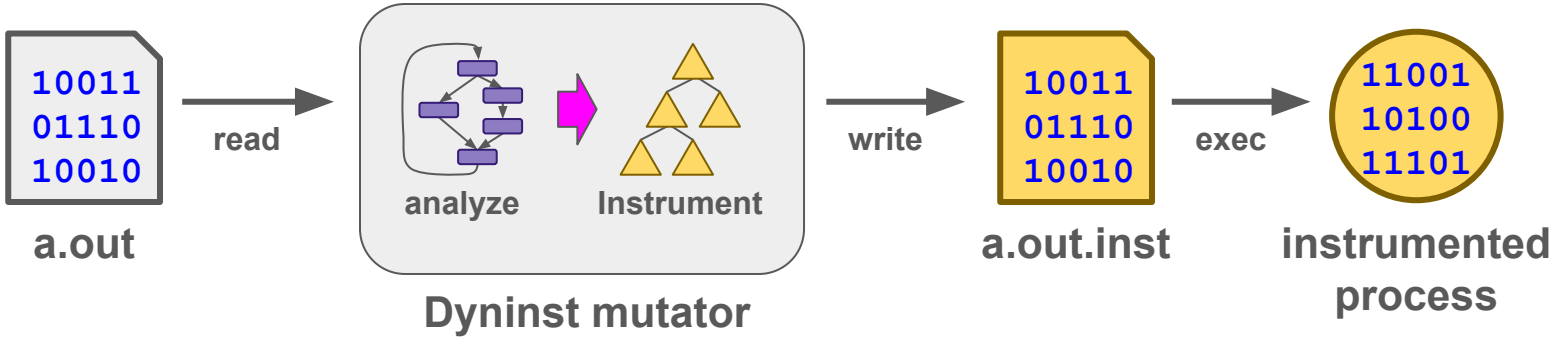
An machine independent interface to machine level binary analysis, instrumentation and control.

- 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

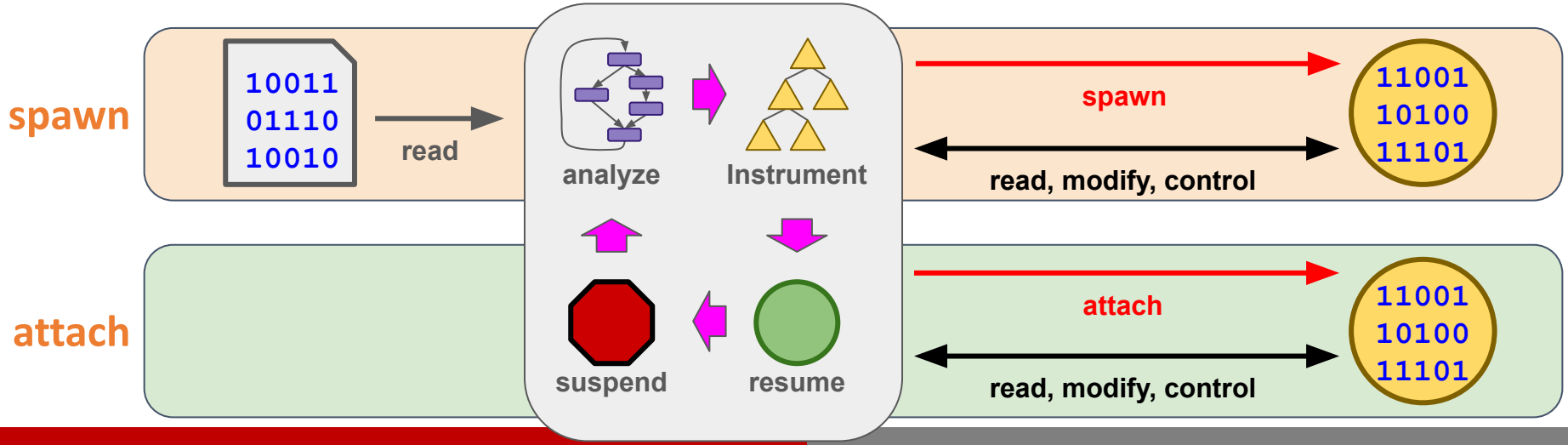
Static Instrumentation



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

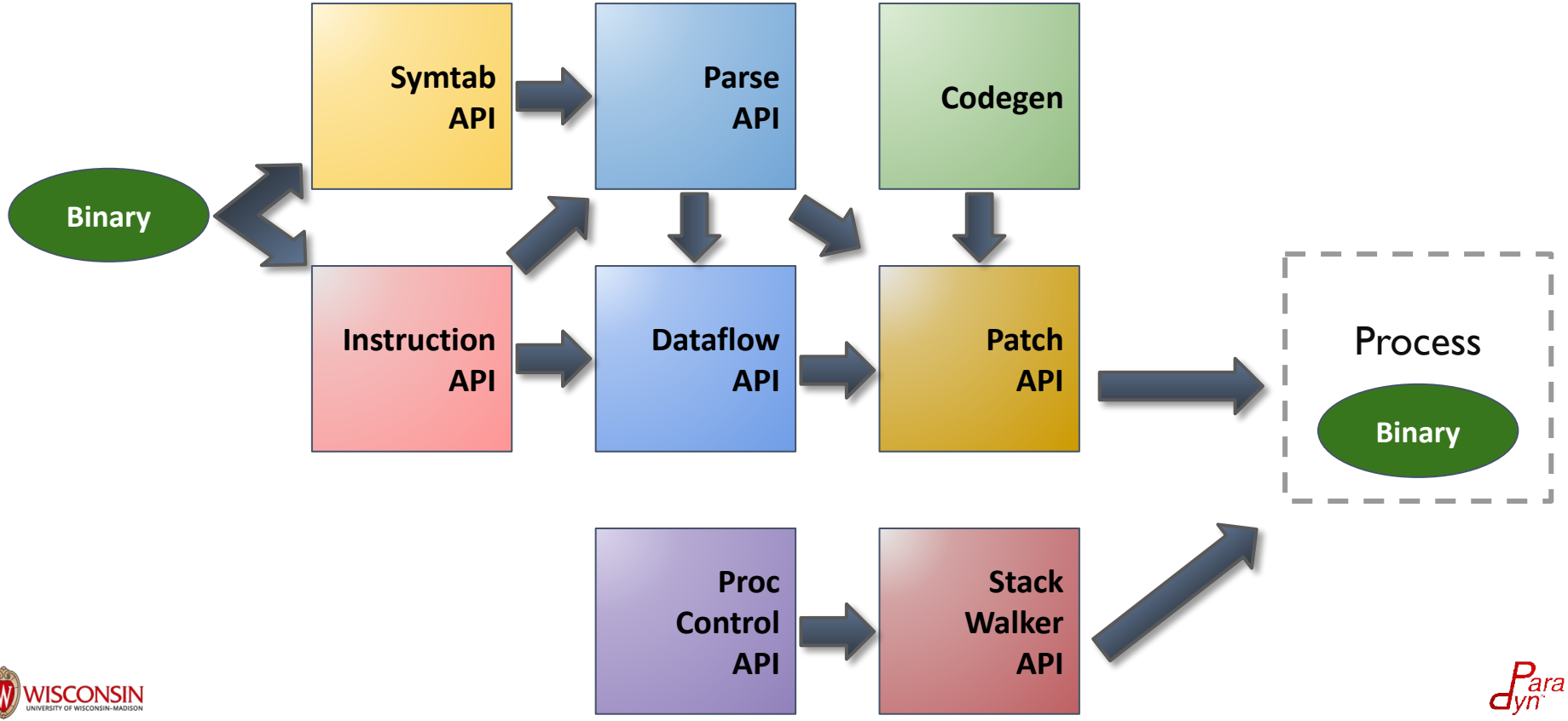
Runtime features

- 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

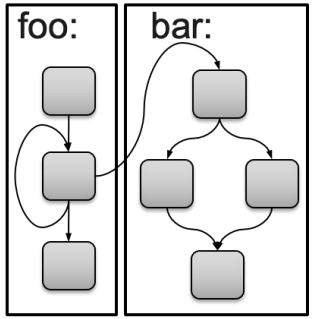
Symbols
- functions
- variables
- types
- …

Binary Properties
- segments
- sections
- ELF properties
- …

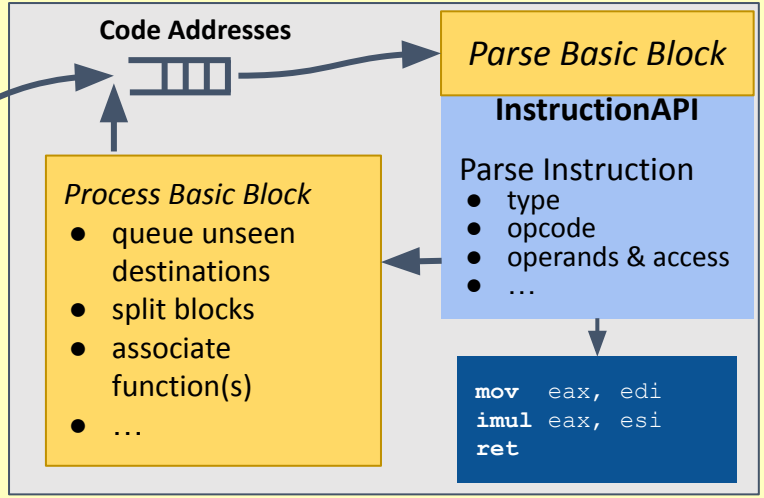
DataFlowAPI

- 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

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

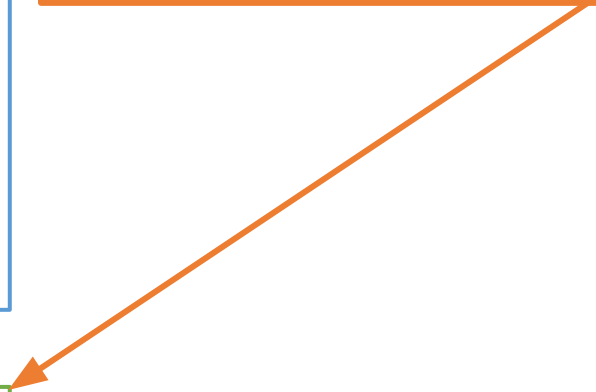
```
XXX <Trace>:
      …
…: // trace functionality
      …
…: retq
```

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);
```

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

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,…);
```

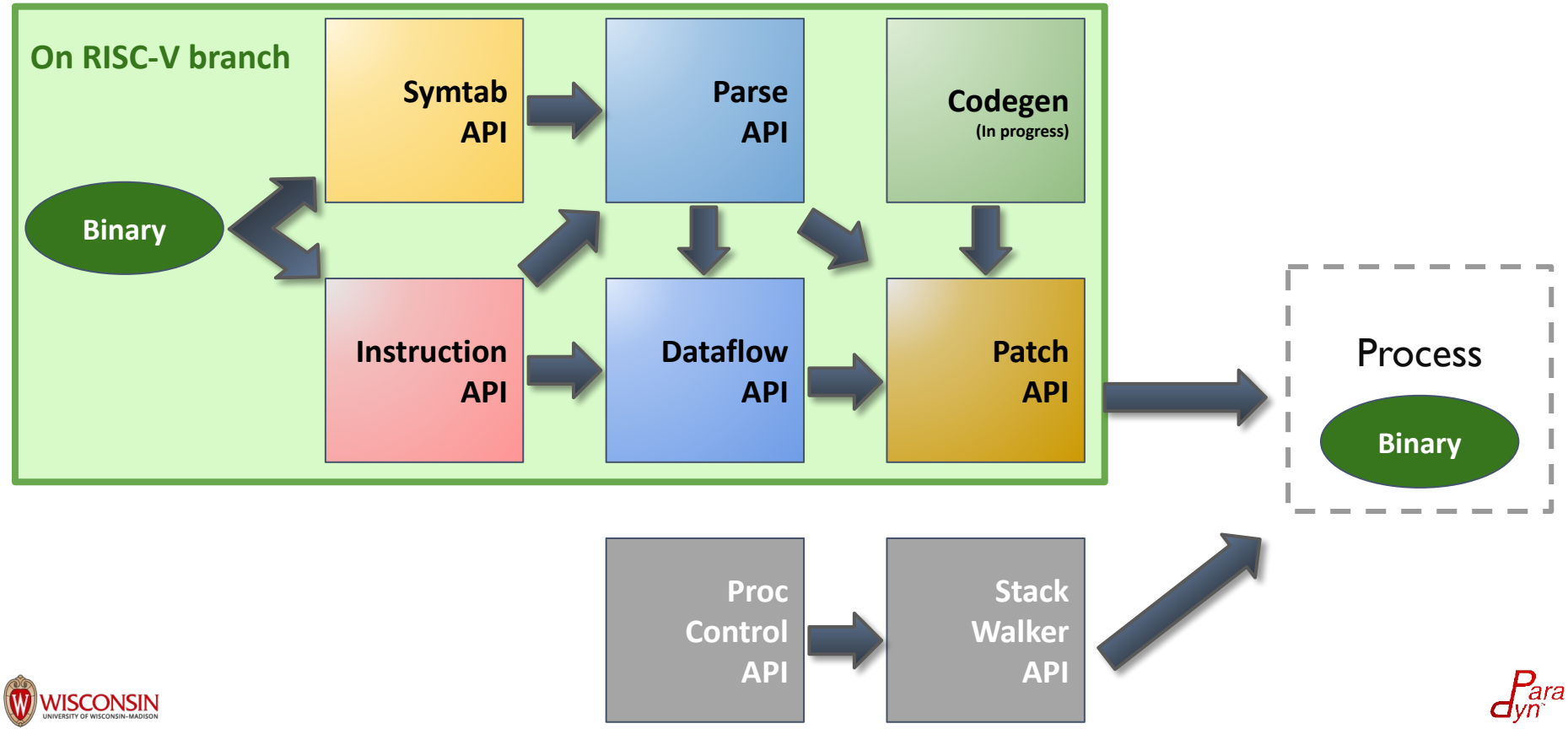
7. Insert snippets

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

What is new since August 2024?

Parsing	• Improve register support to add missing registers and correctly determine base registers • Correct support for syscalls and interrupts on all architectures • Correct instruction semantics and add missing instructions (mainly ARM & PPC)
LineInfo	• Correct LineInformation to support multiple entries mapping to same address
Clean up	• Code Cleanup, bug fixes and new compiler support
Testing	• Correct testsuite tests on PPC and ARM • Github CI improvements - more platforms and tests
GPU	• Updated AMD GPU instructions with new data from AMD • Fixed AMD GPU issues discovered by HPCtoolkit team • Added support to build Dyninst to instrument an architecture different from the host
PE	• Added support to open/parse Windows PE files on Linux
WIP	• Work in progress: GPU analysis and instrumentation & RISC-V port

RISC-V - Working Components

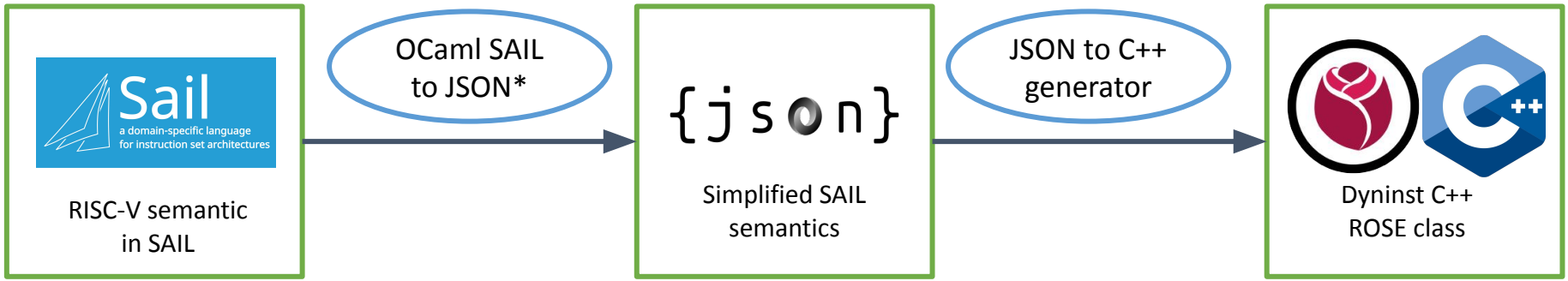


RISC-V Progress

- SymtabAPI
  - RISC-V ELF specific section (e.g. .riscv_attribute)
  - RISC-V specific relocation
  - Gcc specific RISC-V dynamic relocation table (.rela.dyn)
- InstructionAPI
  - Support RISC-V 64 bit (rv64imafdc)
  - Capstone instruction decoder (6.0.0-Alpha1 and above)
  - We added the operand read/write information to Capstone and got them accepted into Capstone 6.0.0-Alpha1

RISC-V Progress

- ParseAPI
  - Added recognition for RISC-V return and call idioms
- DataflowAPI
  - Added instruction semantics to Dyninst ROSE classes
    - Use official RISC-V SAIL instruction specifications



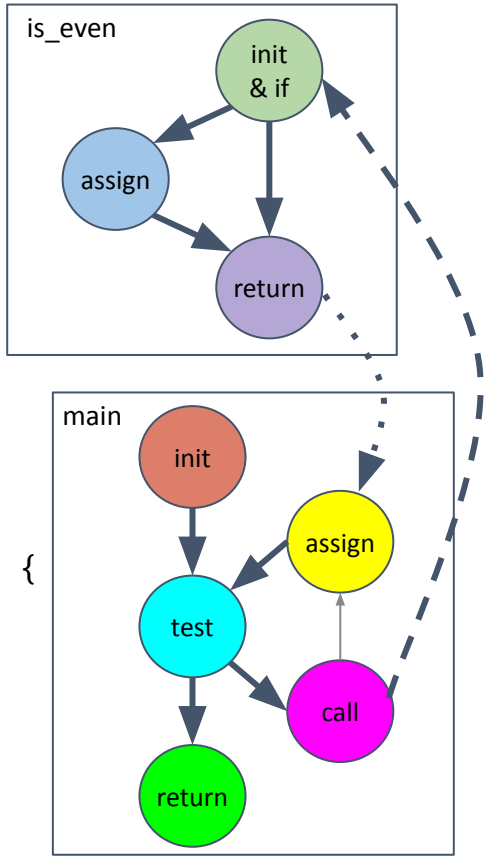
RISC-V Progress

- DyninstAPI
  - Code generation is work in progress
  - Many examples in the Dyninst example repository work
- Future work
  - More testing, working on getting the test suite and all examples in Dyninst to work
  - Complete the proccontrol and stackwalker
  - RISC-V RVA23 (New standardized RISC-V set of features that adds additional extensions like vector)

Demo Code - Count Even Mutatee

```
int is_even(int num) {
    int value = 0;
    if (num % 2 == 0) {
        value = 1;
    }
    return value;
}
```

```
int main() {
    int count = 0;
    for (int i = 0; i < 10; i++) {
        count += is_even(i);
    }
    return count;
}
```



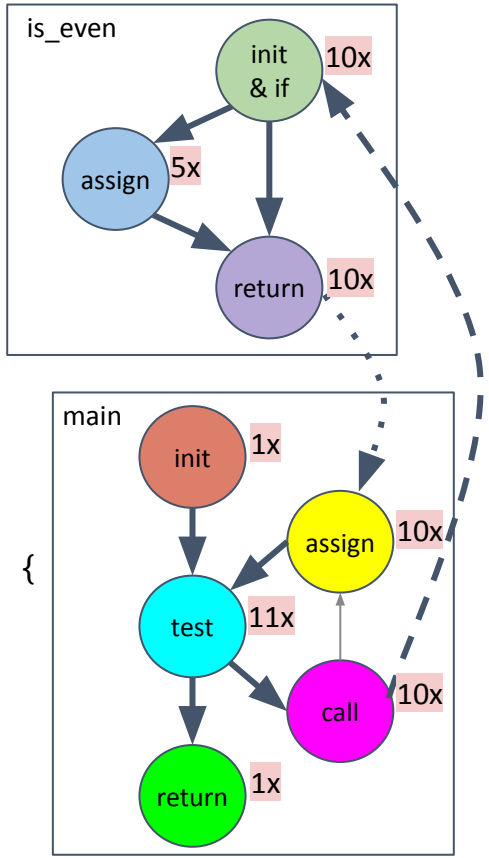
Mutator

- Create two counters
  - Basic block entry counter
  - Function entry counter
- Instrumentation
  - main entry, clear counters
  - Basic block entry, count
  - Function entry, count
  - main end, print counters

Demo Code - Result

```
int is_even(int num) {
    int value = 0;
    if (num % 2 == 0) {
        value = 1;
    }
    return value;
}
```

```
int main() {
    int count = 0;
    for (int i = 0; i < 10; i++) {
        count += is_even(i);
    }
    return count;
}
```




```
$ gcc count_even.c
$ ./mutator a.out mutated
$ ./mutated
Function count: 10
Block count: 58
$
```

Questions?

https://github.com/dyninst/dyninst