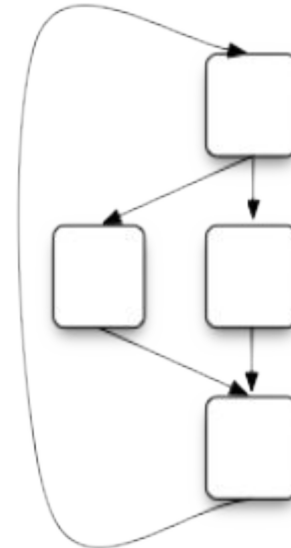
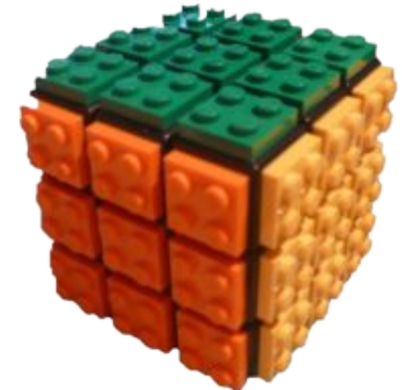
A Dyninst Primer
and
Project Updates Since July 2019

Scalable Tools Workshop
Granlibakken Resort
Lake Tahoe, California
June 20, 2022

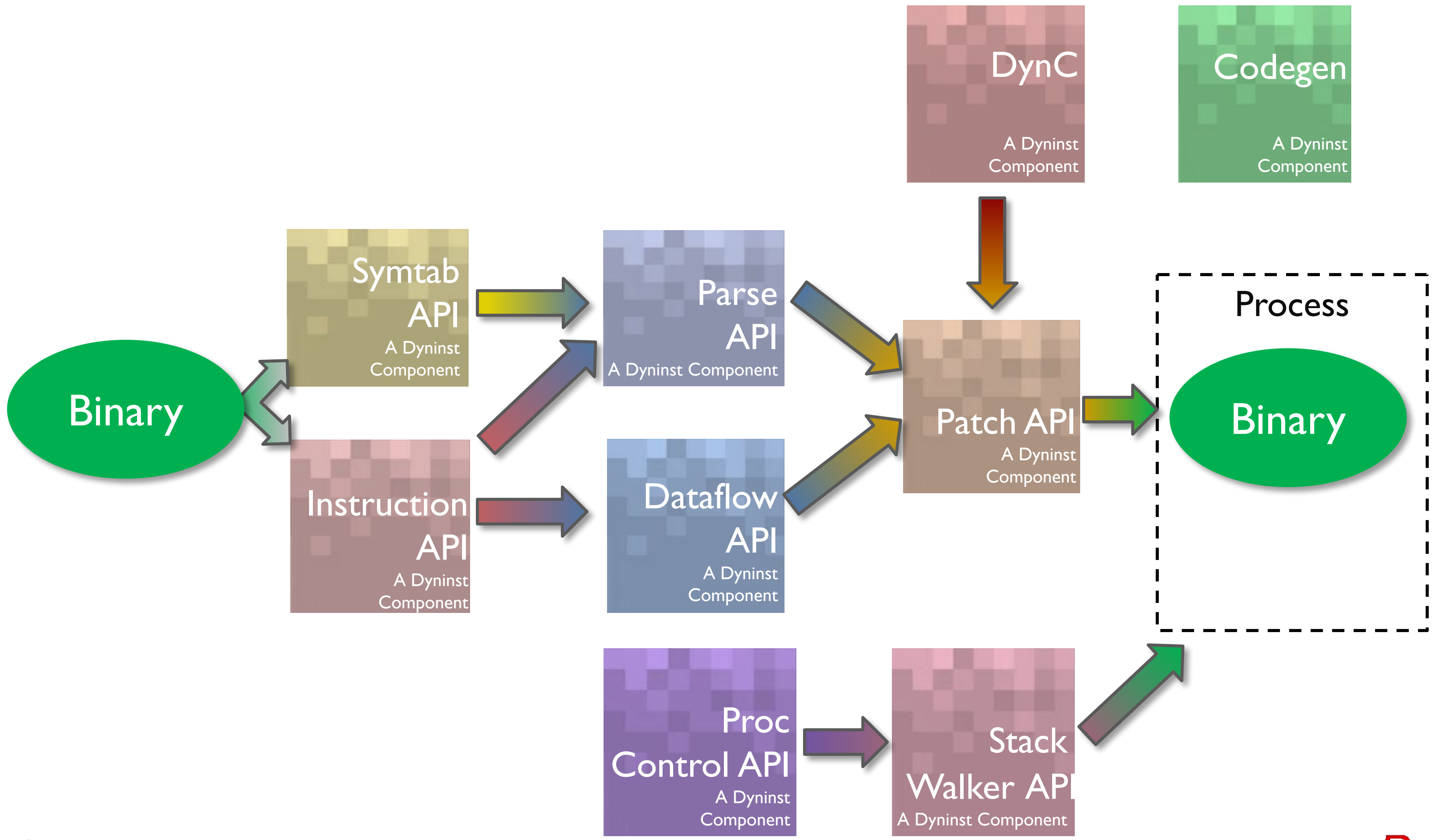
A Brief Introduction to Dyninst





Dyninst: a tool for static and dynamic binary instrumentation and modification

How is Dyninst organized?



Some of Dyninst's Capabilities

- Analysis of executables and libraries
- Instrumentation
  - Rewrite binaries
  - Modify running programs
- Process control
- High level abstractions allow for platform/architecture independence
- Can also extract machine-specific information

What you can do with Dyninst

- find all functions
- find all global variables
- find local variables
- find all used variables
- find all basic blocks
- analyze control graphs
- create call graphs
- perform forward slicing
- perform backward slicing
- analyze instructions
  - totals number
  - total by instruction
  - total by type
- find all memory accesses
- tracing function calls
- code coverage
- performance time/counts
- peephole optimizations
- change program behavior
- fix bugs via patching
- examine call stack
- disassembly
- analyze w/o debug symbols
- analyze stripped binaries
- and more…

Dyninst - Analysis

- Symbols
  - functions
  - variables
  - local variables
  - types
- Code
  - instruction decoding (low-level platform specific)
  - instruction abstraction (high-level platform independent)
  - functions
  - basic blocks
  - control flow graphs
  - loop analysis
  - forward/backward slicing

Dyninst - Instrumentation

- replace instructions (low-level)
- snippet abstraction - list of abstract operations
  - read/write memory, registers, variables
  - basic math
  - call functions
  - conditional branches
  - jumps
  - ...
- location point abstraction - places to modify
  - find using high-level criteria
    - function entry/exit
    - basic block entry/exit
    - memory writes
    - ...
  - snippet insertion

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

Paradyn

Dyninst - Process Control

- Debugger-like functionality
- Attach to running process
- Launch existing or modified binary
- Suspend/resume process
- Use Dyninst's capabilities to allow
  - analysis
  - instrumentation
- Allows dynamic analysis - instrument based on runtime criteria

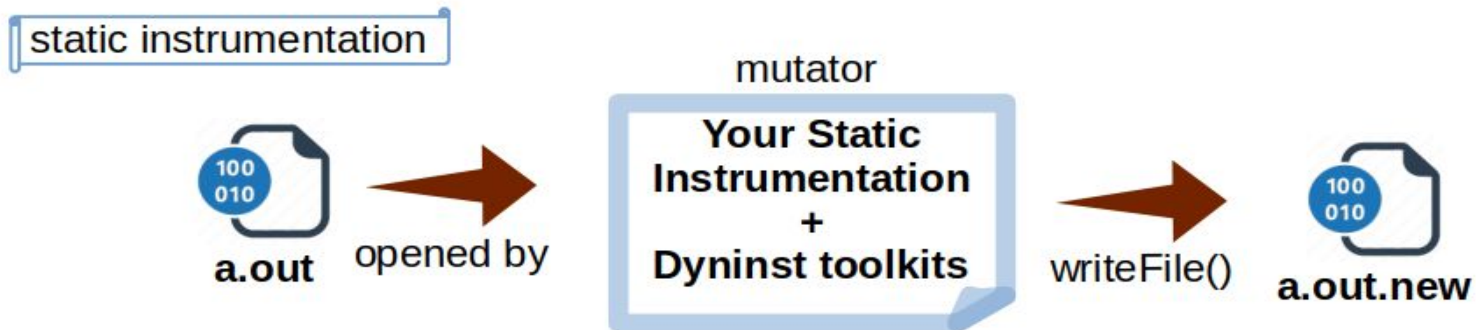
Instrumentation Techniques

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	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

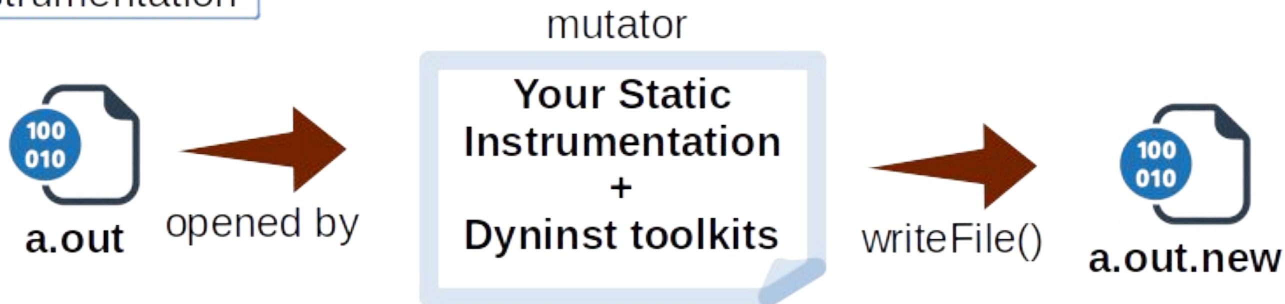
WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

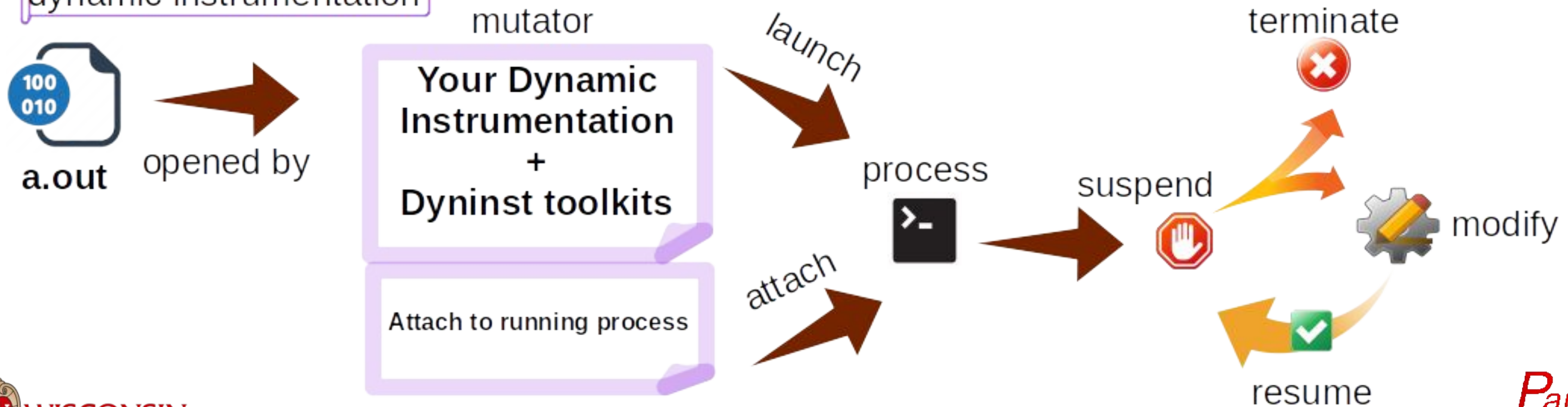
Paradyn

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

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

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

```
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
```

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Paradyn

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");
```

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");
```

WISCONSIN UNIVERSITY OF WISCONSIN-MADISON

Paradyn

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");
```

WISCONSIN UNIVERSITY OF WISCONSIN-MADISON

Paradyn

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

```
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,…);
```

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

```
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

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

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 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

```
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 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);
```

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

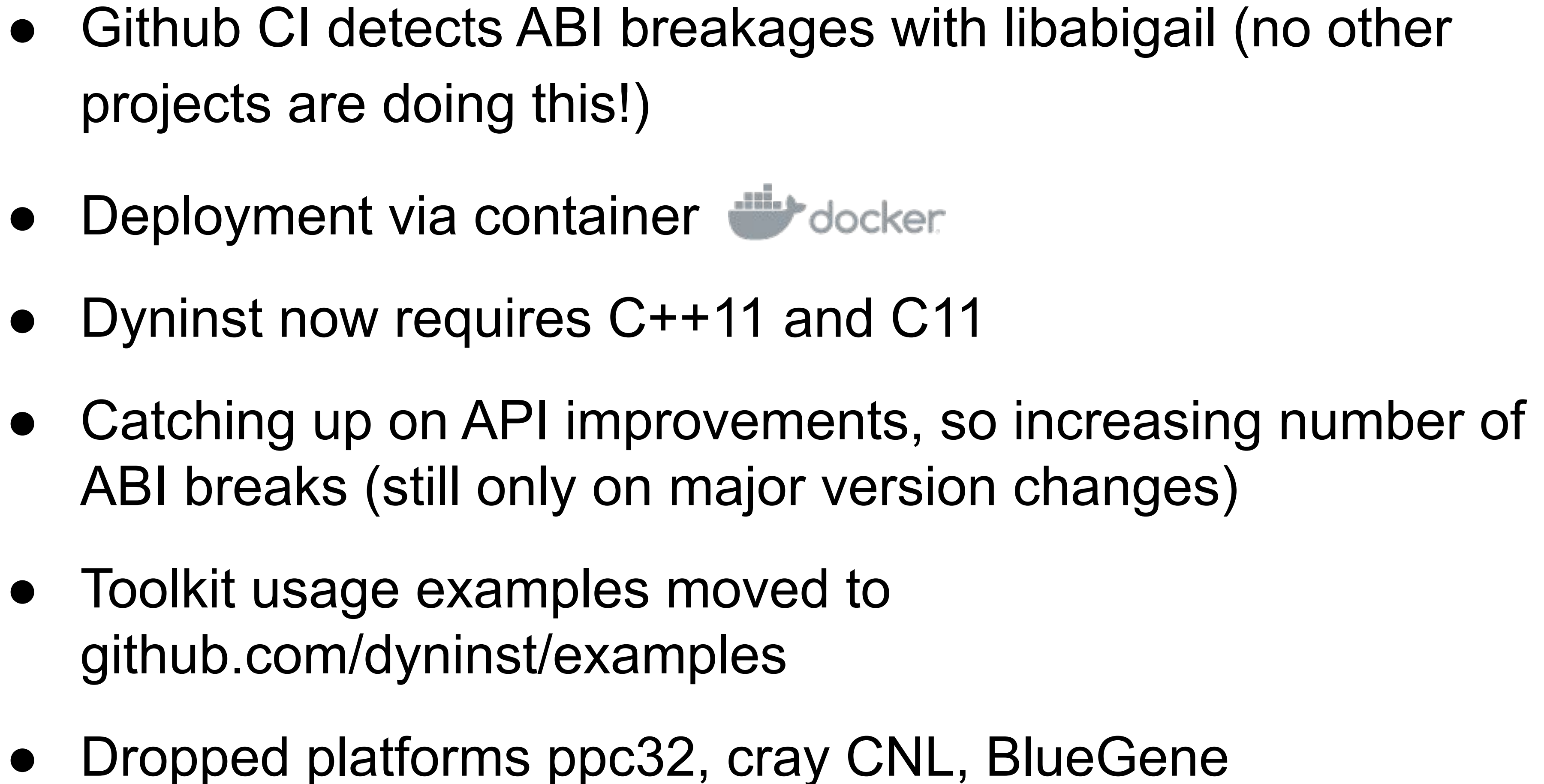
What is new since July 2019?

- Improvements to Dyninst Builds & CI
- GPU Enhancements
- Code Cleanup

Enhancements

- Support PIE binaries (now default on modern Linuxes)
- Correct symbol demangling
  - consistent handling of clone function names and symbol versions
- Support DWARF separate debug files (DWZ)
- ARM support for arbitrary number of function parameters
- Rewritten binaries can now be debugged

Enhancements to Builds & CI



WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Paradyn

Enhancements - NVIDIA

- **NVIDIA**
  - Enhanced line map information (requires elfutils ≥ 1.86)
  - Handle custom ELF parsing needed for CUDA ≥ 11.6
  - Code slicing

Enhancements - AMD GPUs

On GPU code objects extracted from the AMD fat binary

- MI25 (Vega)
  - Instruction Parsing
  - Symbol Parsing
  - Direct & Indirect Control Flow Analysis
- MI200 (CDNA2)
  - Instruction Parsing
  - Symbol Parsing
  - Direct Control Flow Analysis

Code Cleanup

- Compiler warnings cleanup – **no compiler warnings** is our new policy.
- Modernize code to support C++ 11, 14, 17 and 20; and C 11 and 17
- Dyninst libraries built with enhanced DWARF (-g3 -Og)
- Compiler Warning enhancements
  - Enabled 37 gcc warning options (100’s of warning types)
  - Updated code to eliminate warnings – fixed 100’s of real issues and 1000’s of potential issues
- Dropped support for STABs debug information – now DWARF-only.

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Paradyn

Future Work

- Callsite parsing
- AMD GPU binary rewriting and instrumentation
- Support for more GPUs and CPUs
- Update instruction parsing backend
  - Use Capstone for x86 (ARM and PPC in future)
  - Add user-defined handling of unknown instructions
- Improve memory utilization
- Move to modern CMake to make building against Dyninst easier
- Migrate documentation to readthedocs.io

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)
- 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
- Open|SpeedShop
- stat (LLNL)
- ATP (Cray)
- Common Tools Interface (Cray)

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!

Who uses Dyninst?

Some of our Dyninst users:

- HPCToolkit (Rice)
- SystemTap (Red Hat)
- AMD
- Open|SpeedShop
- stat (LLNL)
- ATP (Cray)
- Common Tools Interface (Cray)

Questions?

https://github.com/dyninst/dyninst

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!