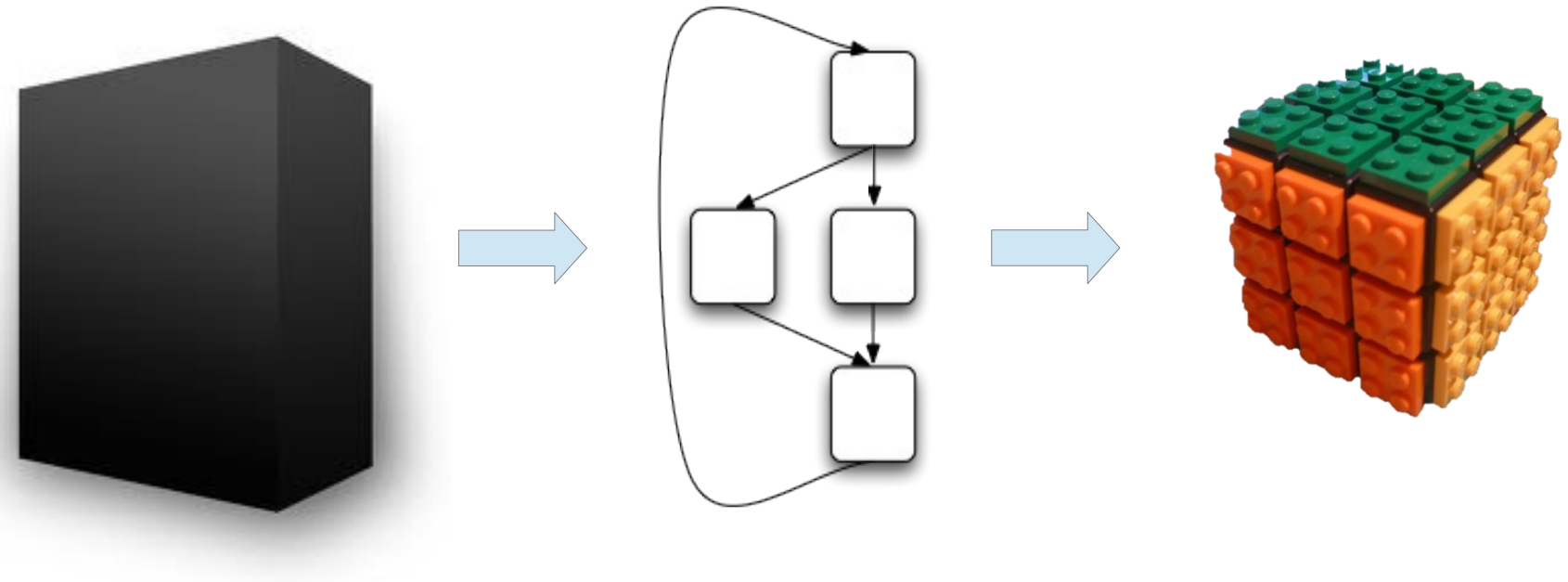
Binary Analysis in the Dyninst Toolkits

Bill Williams
Paradyn Project

Petascale Tools Workshop
Madison, Wisconsin
July 15-17, 2013

A Brief Introduction to Dyninst



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

Why Binary Analysis Matters

- Efficiency
- Accuracy
- Usability
- Abstraction

Sections, Variables and Types

```
03174AB83BC3DF1896DE
125893BAFED48893DBE4
48DDCAA1234290638957
2345728BC3448FFD4722
0012357368939375463D3
2286FFEEFFEEDDDDDDDD
00000000000000000000
BAADF00DCCCCCCCCCCCC
7FFF7FFF100000000000
00000000000000000000
00000000000000000000
```

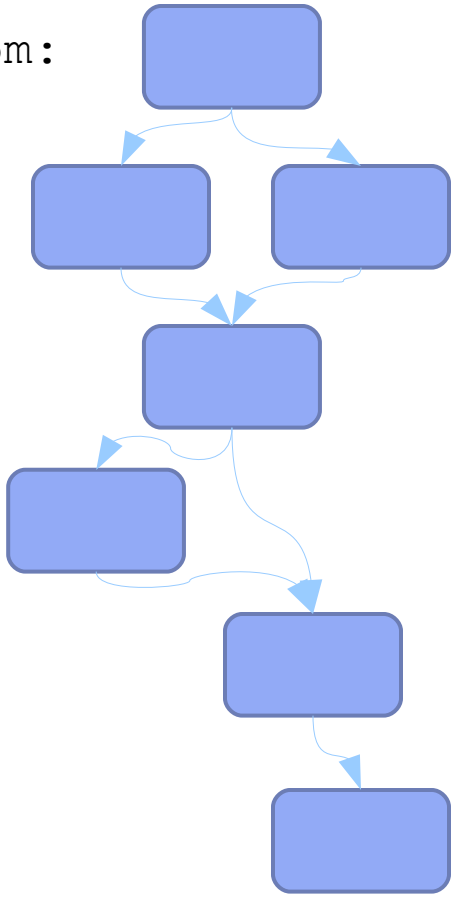
```
Data:
Void* p = 0x03174AB8
Object* this =
    0x7FFF7FFF
          …
```

Functions

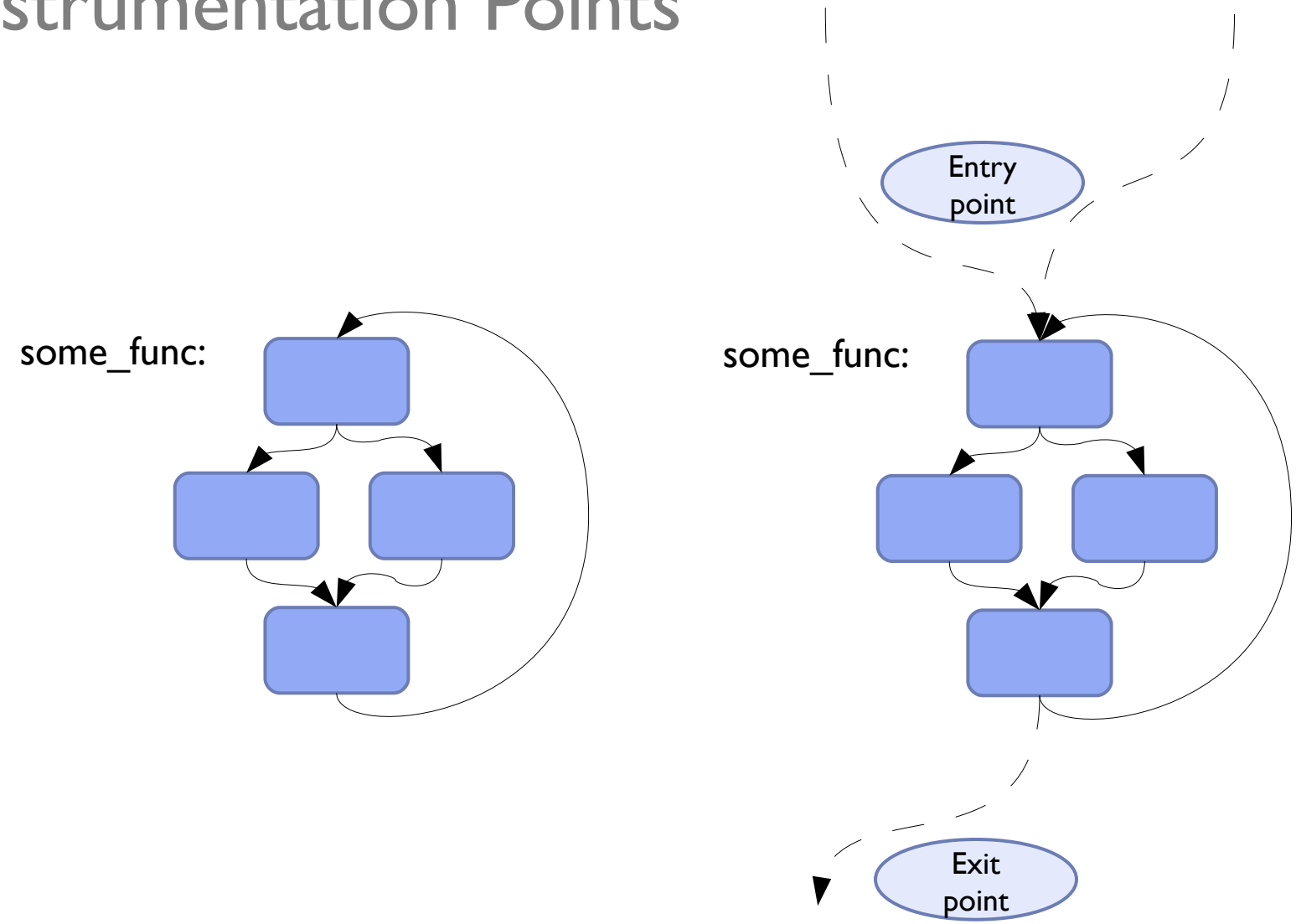
`random:`

```
5531C089E5B901000000
5383EC0CE8FC7FFEFF81
C3D144160065833D0C00
0000007401F00FB18B24
1100000F85BF0100008D
45F8894424048D833803
0000890424E81A030000
65833D0C000000007401
F083AB24110000010F85
A20100008B45F883C40C
5B5DC3
```

`random:`



Instrumentation Points



Introduction to Dataflow Analysis

- What is dataflow analysis?
- How is it different from the analyses we've been talking about?
- How do we use it internally?
- How can users apply it?

Stack Slots and Function Contexts

Assembly code

```
PUSH %EBP
MOV %ESP, %EBP
SUB $0X08, %ESP
          …
ADD %EAX, -0X4(%EBP)
SUB %EBX, 0X4(%ESP)
```

Stack analyzed pseudo-code

```
STACK(-4) = CALLER EBP
ESP = &STACK(-4)
EBP = &STACK(-4)
ESP = &STACK(-12)
          …
STACK(-8) += EAX
STACK(-8) -= EBX
```

Register Liveness and ABIs

Basic block

```
MOV %EAX, 0X4(%ESP)
CMP %EBX, $0
JNE <LABEL>
```

Liveness-based instrumentation

```
PUSH %EAX
CALL get_block_ctr
INC *%EAX
POP %EAX
MOV %EAX, 0X4(%ESP)
CMP %EBX, $0
JNE <LABEL>
```

Future Work

- Ways to make other tools' analysis work on Dyninst-altered binaries/processes
- Ways to use analysis more effectively

Emit Better Binaries

Current

```
.dyninstInst:
<DATA>
          …
<RELOCS>
printf:
          …
<TEXT>
          …
MOV printf, %EBX
CALL %EBX
```

Improved

```
.dyninstInst:
<RELOCS>
printf@plt:
          …
<TEXT>
          …
CALL printf@plt
<DATA>
          …
```

Updating debugging information

Current

```
Foo: jmp Foo_dyninst

Foo_dyninst:
<nothing GDB understands>
```

No source line information in Foo_dyninst
No local variable information in Foo_dyninst
No information about instrumentation locals
Exception handlers not moved to Foo_dyninst

Improved

```
Foo: jmp Foo_dyninst

Foo_dyninst:
<original code>
<state saves>
<instrumentation>
<state restores>
<original code>
```

Add source lines for original code
Add locals for original code
Add locals for instrumentation
Move exception handlers

Improve Types

Current

```
char* foo;
void* bar;
std::string baz;
assignExpr(foo, bar);
assignExpr(baz, foo);
```

No warning
Not possible

Improved

```
char* foo;
void* bar;
std::string baz;
assignExpr(foo, bar);
assignExpr(baz, foo);
```

WARNING: reinterpret_casting!
Use char* constructor for std::string

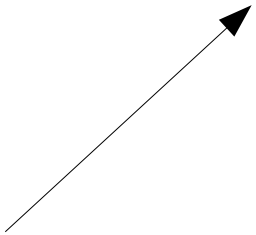
Improve Parsing

- Indirect control flow
- Tail calls
- Incorporate SD-Dyninst techniques

Improve Code Generation

Block entry

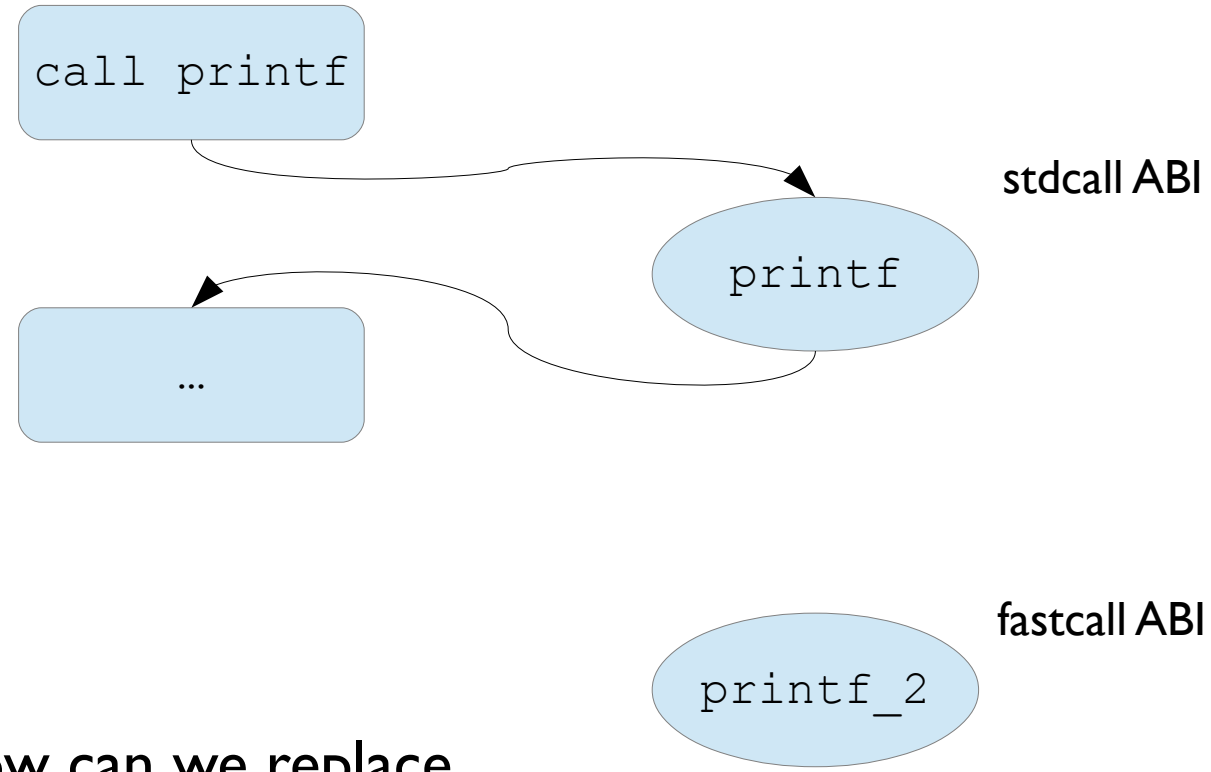
```
PUSH %EAX
CALL get_block_ctr
INC *%EAX
POP %EAX
MOV %EAX, 0X4(%ESP)
CMP %EBX, $0
JNE <LABEL>
```



Better location

```
MOV %EAX, 0X4(%ESP)
~~PUSH %EAX~~
CALL get_block_ctr
INC *%EAX
~~POP %EAX~~
CMP %EBX, $0
JNE <LABEL>
```

Improve Modification Abstractions



How can we replace printf with printf_2?

Conclusions

- Binary analysis isn't just dataflow
- Dyninst currently does a lot of analysis…
- …but there's still plenty of room to improve