Dataflow Analysis in Dyninst

Xiaozhu Meng
Paradyn Project

Paradyn / Dyninst Week
College Park, Maryland
March 26-28, 2012

Four Analyses in Dyninst

- Liveness Analysis
  - Determine which registers are live for each instruction
- Stack Analysis
  - Calculate the height of a stack frame for each instruction
- Program Slicing
  - Identify the subset of the program that affects or is affected by a statement (or instruction)
- Symbolic Evaluation
  - Derive a symbolic expression that represents values of registers

Why are they useful?

- Liveness Analysis
  - Save fewer registers in code generation (DyninstAPI)
  - Save fewer registers when using PatchAPI snippet interface (LLNL)
- Stack Analysis
  - Return address checking (Emily Jacobson)
  - AnalysisStepper in StackwalkerAPI (ProcControlAPI Integration)
- Program Slicing and Symbolic Evaluation
  - BLR can be a return instruction or an indirect branch (ParseAPI)
  - Sensitivity resistance analysis (Andrew Bernat/Kevin Roundy)
  - Detect call stack tampering (Kevin Roundy)
  - Unstrip binaries (Emily Jacobson/Nathan Rosenblum)
  - Root-cause analysis (Ignacio Laguna@Purdue and LLNL)
  - Analyze memory access strides in loop (Xu Liu@Rice)

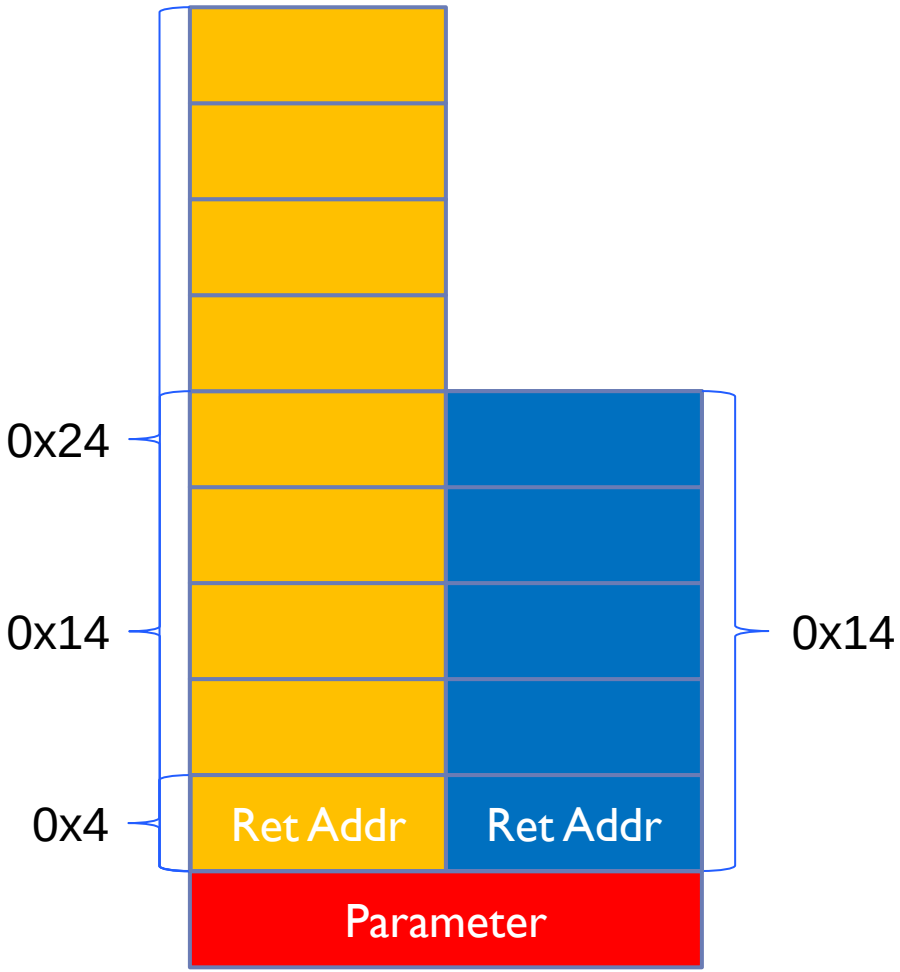
Which registers are live?

```
<factorial>
   push   %rbx
   mov    $0x1,%rax
   mov    %rdi,%rbx
   sub    $0x190,%rsp
   test   %rdi,%rdi
   je     basecase
   lea    -0x1(%rbx),%rdi
   mov    %rsp,%rsi
   callq <factorial>
   imul   %rbx,%rax
basecase:
   add    $0x190,%rsp
   pop    %rbx
   retq
```

Live Regs	Ref	Def

What is the Height of the Stack Frame?

```
<foo>
   sub   $0x10,%esp
   movl  $0x1,0xc(%esp)
   cmpl  $0xa,0x14(%esp)
   jle   done
   sub   $0x10,%esp
   mov   0x24(%esp),%eax
   mov   %eax,0x18(%esp)
   mov   %eax,0x1c(%esp)
   add   $0x10,%esp
done:
   mov   0xc(%esp),%eax
   add   $0x10,%esp
   ret
```



Slice on an Instruction

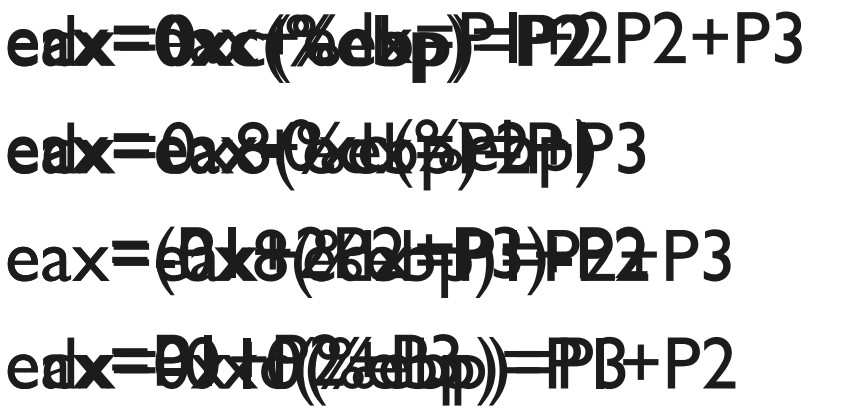
```
stwu    r1,-16(r1)
lis     r9,4097
mflr    r0
addi    r3,r9,28404
stw     r0,20(r1)
lwz     r0,28404(r9)
cmpwi   cr7,r0,0
beq-    cr7,10002d38
lis     r9,0
addi    r9,r9,0
cmpwi   cr7,r9,0
beq-    cr7,10002d38
mtctr   r9
bctrl
lwz     r0,20(r1)
addi    r1,r1,16
mtlr    r0
blr
```

```
mflr    r0

stw     r0,20(r1)

lwz     r0,20(r1)

mtlr    r0
blr
```

Symbolically Evaluate a Function

```
<sum>
mov 0xc(%ebp),%eax
mov 0x8(%ebp),%edx
lea (%edx,%eax,1),%eax
mov %eax,-0xc(%ebp)
mov 0x10(%ebp),%eax
mov 0xc(%ebp),%edx
lea (%edx,%eax,1),%eax
mov %eax,-0x8(%ebp)
mov -0x8(%ebp),%eax
mov -0xc(%ebp),%edx
lea (%edx,%eax,1),%eax
sub 0xc(%ebp),%eax
mov %eax,-0x4(%ebp)
ret
```

Variable	Symbolic Value
eax	P1+P2+P3
edx	P1+P2
0x8(%ebp)	P1
0xc(%ebp)	P2
0x10(%ebp)	P3



Dyninst Components

class LivenessAnalyzer

```
bool query(ParseAPI::Location,
           Type, bitArray&);
```

Return liveness as a bitArray for the given location

```
int getIndex(MachRegister);
```

Get the index of given MachRegister in bitArray

```
bool query(ParseAPI::Location, Type,
           MachRegister, bool&);
```

Return liveness as a bool at the given location for the given register

Saving Fewer Registers

```
<factorial>
  push   %rbx
  mov    $0x1,%rax
  mov    %rdi,%rbx
  sub    $0x190,%rsp
  test   %rdi,%rdi
  je     basecase
  lea    -0x1(%rbx),%rdi
  mov    %rsp,%rsi
  callq <factorial>
  imul   %rbx,%rax
basecase:
  add    $0x190,%rsp
  pop    %rbx
  retq
```

```
pushf
add $0x1 [601001]
popf
```

Live Regs
rbx,rdi

We only need to save registers that are both written by snippets and live

Liveness Analysis Code Example

```
ParseAPI::Location loc(FuncEntry(func));
LivenessAnalyzer live(ADDR_LEN_X86_64);
bool flagLive;
if (live.query(loc, ParseAPI::Before,
                x86_64::flags, flagLive)){
    if (flagLive){
        SaveFlag();
    }
}
```

class StackAnalysis

Height represents integer values or "unknown"

```
Height findSP(ParseAPI::Location,
              Type);
```

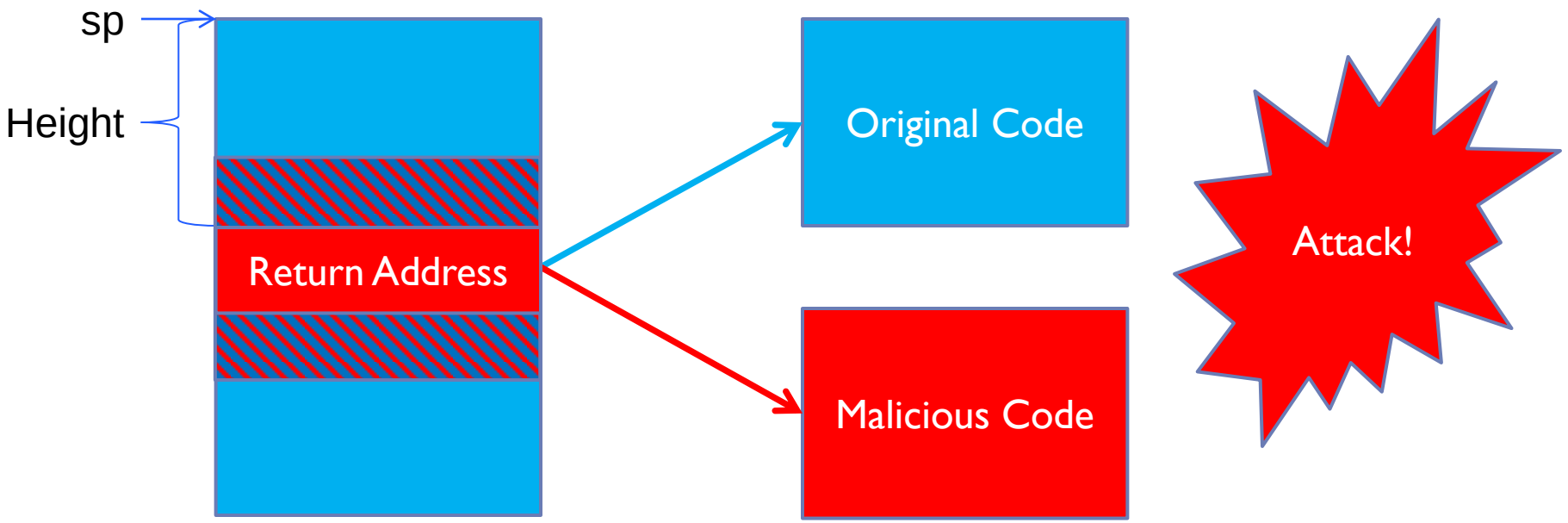
Get the height of the stack pointer

```
Height find(ParseAPI::Location,
            Type, MachRegister);
```

Get the height of an alias of the stack pointer

Checking Return Address

- Some attacks change the return address to force programs to return to malicious code.
- Stack analysis identifies the return address, enabling other analyses to check its validity



Checking Return Address

```
StackAnalysis sa(func);
ParseAPI::Location loc(func, block, currAddr);
StackAnalysis::Height height;
height=sa.findSP(loc, ParseAPI::Before);
if (height.isBottom()) return;
Address retAddr;
retAddr=getStackContent(getSP()+height.val());
if (isNotCallerReturnAddr(retAddr))
      reportAttack();
```

class Slicer

The Predicates class tunes slicing behavior, determining

- Slicing end points
- Whether to slice into a caller or callee
- When to "widen"

```
Graph::Ptr forwardSlice(Predicates);
```

Return a Graph representation of forward slicing

```
Graph::Ptr backwardSlice(Predicates);
```

Return a Graph representation of backward slicing

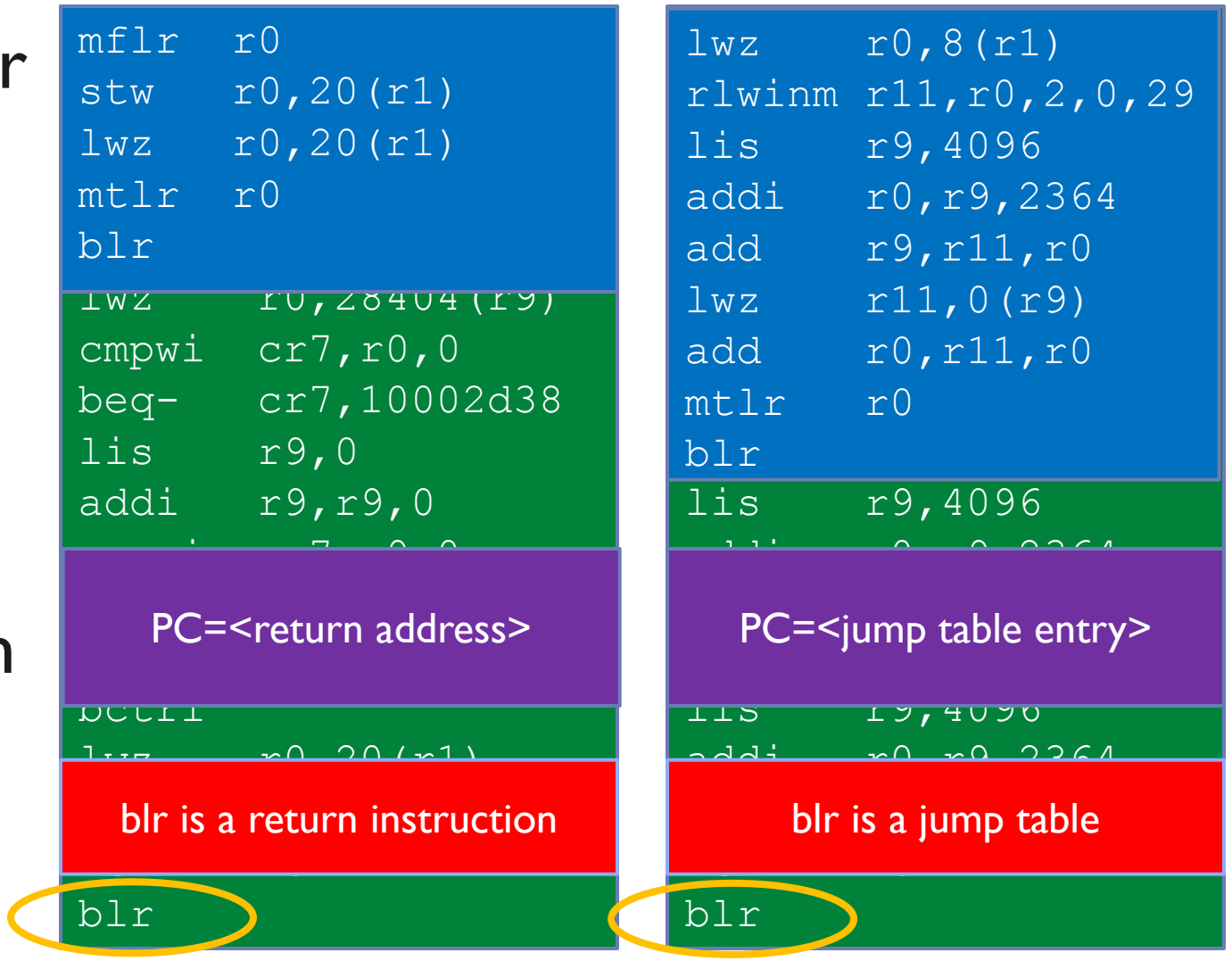
class SymEval

```
static bool expand(Graph::Ptr,
                   SymEval::Result&
                   );
```

Symbolically evaluate the provided slice

Return or Jump Table?

1. Slice on blr
2. Symbolic evaluate remaining code
3. Check the expression of PC



Summary

- Dataflow analysis is an effective method to extract hidden information in programs.
- Dataflow API is a tool box containing four strong dataflow analysis tools.
- Currently in beta status.