CSc 466/566

Computer Security

6 : Man-At-The-End — Program Analysis

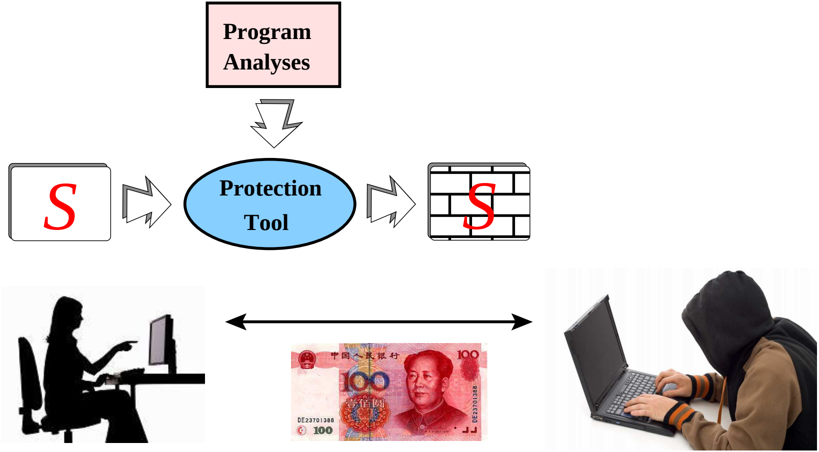
Version: 2014/09/18 14:57:46

Department of Computer Science
University of Arizona

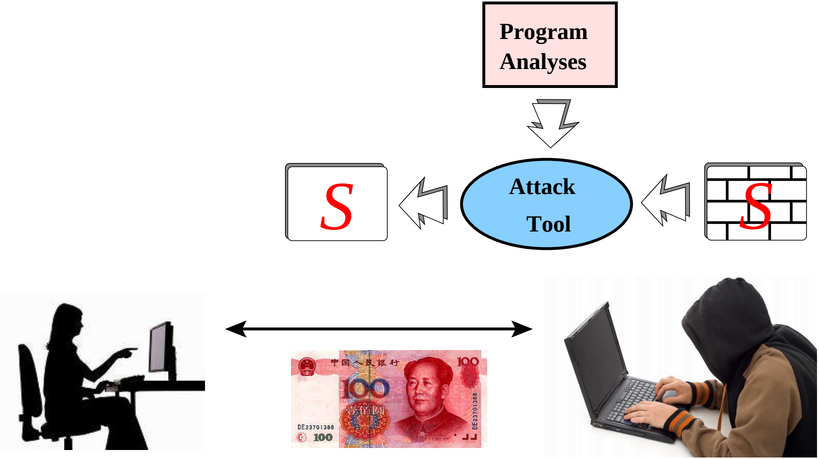
collberg@gmail.com

Christian Collberg

Program Analysis



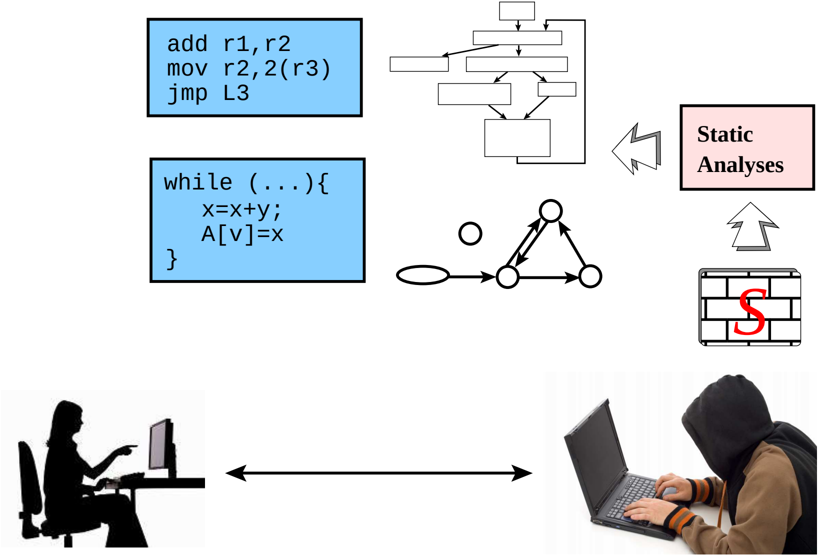
Defenders analyze their program to protect it!



Attackers analyze our program to modify it!

Program Analysis

- Attackers: need to analyze our program to modify it!
- Defenders: need to analyze our program to protect it!
- Two kinds of analyses:
  1. *static analysis tools* collect information about a program by studying its code;
  2. *dynamic analysis tools* collect information from executing the program.


add r1,r2
mov r2,2(r3)
jmp L3
while (...){
x=x+y;
A[v]=x
}
Static Analyses
S

Static Analyses

- control-flow graphs: representation of (possible) control-flow in functions.

Static Analyses

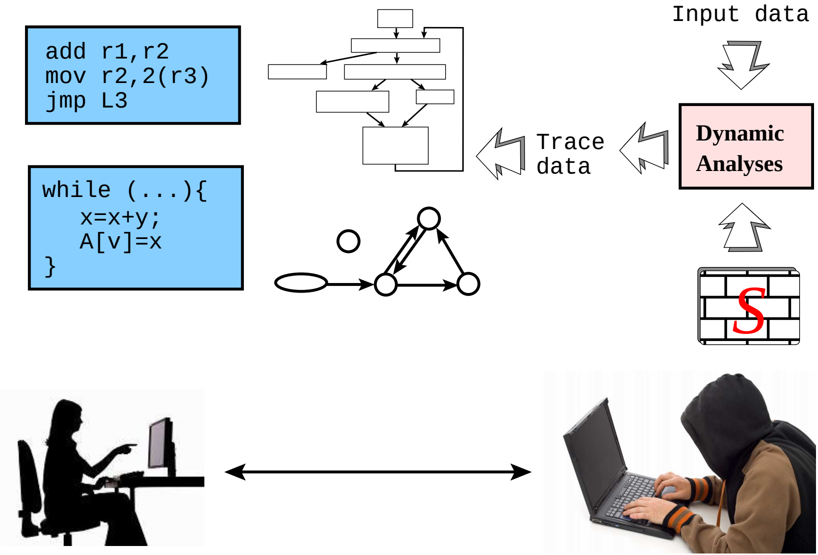
- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.

Static Analyses

- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.
- disassembly: turn raw executables into assembly code.

Static Analyses

- control-flow graphs: representation of (possible) control-flow in functions.
- call graphs: representation of (possible) function calls.
- disassembly: turn raw executables into assembly code.
- decompilation: turn raw assembly code into source code.


add r1,r2
mov r2,2(r3)
jmp L3
while (...){
x=x+y;
A[v]=x
}
Input data
Trace data
Dynamic Analyses
S

Dynamic Analyses

- debugging: what path does the program take?

Dynamic Analyses

- debugging: what path does the program take?
- tracing: which functions/system calls get executed?

Dynamic Analyses

- debugging: what path does the program take?
- tracing: which functions/system calls get executed?
- profiling: what gets executed the most?

Control-Flow Graphs

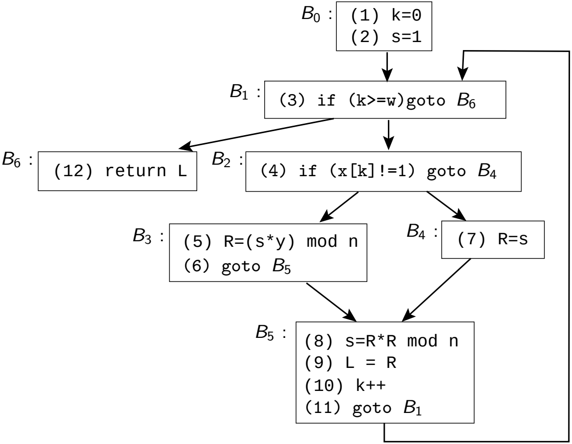
Control-flow Graphs (CFGs)

- A way to represent the possible flow of control inside a function.
- Nodes are called basic blocks.
- Each block consists of straight-line code ending (possibly) in a branch.
- An edge $A \rightarrow B$: control could flow from A to B.

```
int modexp(int y, int x[],
           int w, int n) {
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      if (x[k] == 1)
         R = (s*y) % n;
      else
         R = s;
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```

```
 (1) k=0
 (2) s=1
 (3) if (k>=w) goto (12)
 (4) if (x[k]!=1) goto (7)
 (5) R=(s*y)%n
 (6) goto (8)
 (7) R=s
 (8) s=R*R%n
 (9) L=R
(10) k++
(11) goto (3)
(12) return L
```

The resulting graph



Step 1: Generate Three-Address Statements

- Compile the function into a sequence of simpler statements:
  - `x = y + z`
  - `if (x < y) goto L`
  - `goto L`
- These are called three-address statements.
- Other representations are possible, for example expression trees.

Step 2: Build the graph

BuildCFG(F):

1. Mark every instruction which can start a basic block as a *leader*:
   - the first instruction is a leader;
   - any target of a branch is a leader;
   - the instruction following a conditional branch is a leader.
2. A basic block consists of the instructions from a leader up to, but not including, the next leader.
3. Add an edge $A \to B$ if A ends with a branch to B or can fall through to B. $\square$

In-Class Exercise I

```
int gcd(int x, int y) {
    int temp;
    while (true) {
        if (x%y == 0) break;
        temp = x%y;
        x = y;
        y = temp;
    }
}
```

1. Turn this function into a sequence of three-address statements.
2. Turn the sequence of simplified statements into a CFG.

In-Class Exercise II

```
X := 20;
WHILE X < 10 DO
    X := X-1;
    A[X] := 10;
    IF X = 4 THEN
        X := X - 2;
    ENDIF;
ENDDO;
Y := X + 5;
```

⇨

```
(1)    X := 20
(2)    if X>=10 goto (8)
(3)    X := X-1
(4)    A[X] := 10
(5)    if X<>4 goto (7)
(6)    X := X-2
(7)    goto (2)
(8)    Y := X+5
```

1. Construct the corresponding CFG.

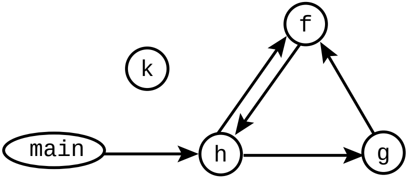
Call Graphs

Interprocedural control flow

- *Interprocedural analysis* also considers flow of information between functions.
- Call graphs are a way to represent possible function calls.
- Each node represents a function.
- An edge $A \rightarrow B$: A might call B.

Building call-graphs

```
void h();

void f(){ h(); }

void g(){ f(); }

void h() { f(); g(); }

void k() {}

int main() {
    h();
    void (*p)() = &k;
    p();
}
```

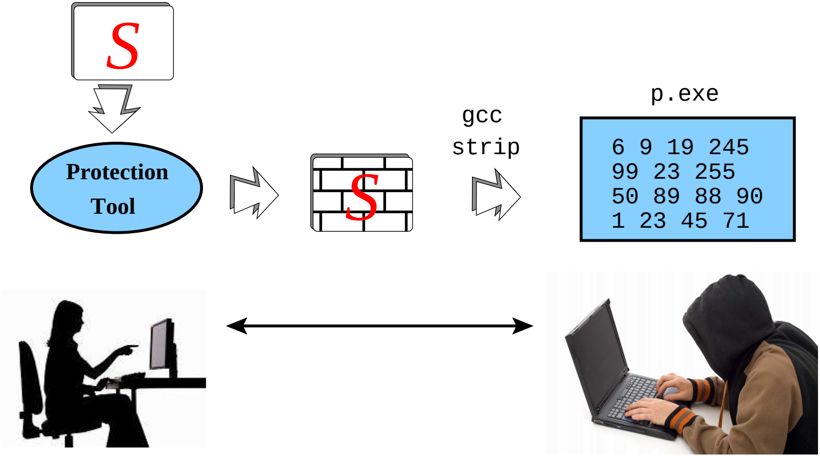


In-Class Exercise

- Build the call graph for the Java program on the next slide.

```java
class M {
    public void a () {System.out.println("hello");}
    public void b () {}
    public void c () {System.out.println("world!")}
}
class N extends M {
    public void a () {super.a();}
    public void b () {this.b(); this.c();}
    public void c () {}
}
class Main {
    public static void main (String args[]) {
        M x = (args.length > 0)? new M() : new N();
        x.a();
        N y = new N(); y.b();
    }
}
```

Disassembly


S
Protection Tool
S
gcc
strip
p.exe
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71

p.as

```
mov R1,R2
add R2,R2
jmp L3
L4: call Q
```

p.exe

```
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71
```

Instruction Set

- On the next slide you will see an instruction set for a small architecture.
- All operators and operands are one byte long.
- Instructions can be 1-3 bytes long.

Instruction set 1

opcode	mnemonic	operands	semantics
0	`call`	*addr*	function call to *addr*
1	`calli`	*reg*	function call to address in *reg*
2	`brg`	*offset*	branch to $\mathrm{pc}+$*offset* if flags for $>$ are set
3	`inc`	*reg*	*reg* $\leftarrow$ *reg* $+ 1$
4	`bra`	*offset*	branch to $\mathrm{pc} +$ *offset*
5	`jmpi`	*reg*	jump to address in *reg*
6	`prologue`		beginning of function
7	`ret`		return from function

Instruction set 2

opcode	mnemonic	operands	semantics
8	`load`	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	`loadi`	reg, imm	$reg \leftarrow imm$
10	`cmpi`	reg, imm	compare *reg* and *imm* and set flags
11	`add`	reg_1, reg_2	$reg_1 \leftarrow reg_1 + reg_2$
12	`brge`	*offset*	branch to $\mathrm{pc}+$*offset* if flags for $\geq$ are set
13	`breq`	*offset*	branch to $\mathrm{pc}+$*offset* if flags for $=$ are set
14	`store`	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$

Disassembly — example

```
6 0 10 9 0 43 1 0 7 0 6 9 0 1 10
0 1 2 26 9 1 30 11 1 0 8 2 1 5 2
32 37 9 1 3 4 7 9 1 4 4 2 7 6 9 0
3 7 6 9 0 1 7 42 2 4 3 1 7 4 3 4 1
```

- Next few slides show the results of different disassembly algorithms.
- Correctly disassembled regions are in pink.

```
main: # ORIGINAL PROGRAM
0: [6]        prologue
1: [0,10]     call      foo
3: [9,0,43]   loadi     r0,43
6: [1,0]      calli     r0
8: [7]        ret
9: [0]        .align     2
foo:
10:[6]        prologue
11:[9,0,1]    loadi     r0,1
14:[10,0,1    cmpi      r0,1
17:[2,26]     brg       26
19:[9,1,30]   loadi     r1,30
22:[11,1,0]   add       r1,r0
25:[8,2,1]    load r2,(r1)
28:[5,2]      jmpi      r2
30:[32]       .byte     32
31:[37]       .byte     37
32:[9,1,3]    loadi     r1,3
35:[4,7]      bra       7
37:[9,1,4]    loadi     r1,4
40:[4,2]      bra       2
42:[7]        ret
```

```
bar:
43:[6]        prologue
44:[9,0,3]    loadi     r0,3
47:[7]        ret
baz:
48:[6]        prologue
49:[9,0,1]    loadi     r0,1
52:[7]        ret
life:
53:[42]       .byte     42
fred:
54:[2,4]      brg       4
56:[3,1]      inc       r1
58:[7]        ret
59:[4,3]      bra       3
61:[4,1]      bra       1
```

```
# LINEAR SWEEP DISASSEMBLY
0:  [6]       prologue
1:  [0,10]    call      10
3:  [9,0,43]  loadi     r0,43
6:  [1,0]     calli     r0
8:  [7]       ret
9:  [0,6]     call      6
11:[9,0,1]    loadi     r0,1
14:[10,0,1]   cmpi      r0,1
17:[2,26]     brg       26
19:[9,1,30]   loadi     r1,30
22:[11,1,0]   add       r1,r0
25:[8,2,1]    load   r2,(r1)
28:[5,2]      jmpi      r2
30:[32]       ILLEGAL   32
31:[37]       ILLEGAL   37
32:[9,1,3]    loadi     r1,3
35:[4,7]      bra       7
37:[9,1,4]    loadi     r1,4
40:[4,2]      bra       2
42:[7]        ret
```

```
43:[6]        prologue
44:[9,0,3]    loadi     r0,3
47:[7]        ret
48:[6]        prologue
49:[9,0,1]    loadi     r0,1
52:[7]        ret
53:[42]       ILLEGAL   42
54:[2,4]      brg       4
56:[3,1]      inc       r1
58:[7]        ret
59:[4,3]      bra       3
61:[4,1]      bra       1
```

```
f0: # RECURSIVE TRAVERSAL
0:  [6]          prologue
1:  [0,10]       call      10
3:  [9,0,43]     loadi     r0,43
6:  [1,0]        calli     r0
8:  [7]          ret

9:  [0]          .byte     0

f10:
10:[6]           prologue
11:[9,0,1]       loadi     r0,1
14:[10,0,1]      cmpi      r0,1
17:[2,26]        brg       26
19:[9,1,30]      loadi     r1,30
22:[11,1,0]      add       r1,r0
25:[8,2,1]       load   r2,(r1)
28:[5,2]         jmpi      r2
30:[32]          .byte     32
31:[37]          .byte     37
```

```
32:[9,1,3]       loadi     r1,3
35:[4,7]         bra       7
37:[9,1,4]       loadi     r1,4
40:[4,2]         bra       2
42:[7]           ret
43:[6]           prologue
44:[9,0,3]       loadi     r0,3
47:[7]           ret

48:[6]           .byte     6
49:[9]           .byte     9
50:[0]           .byte     0
51:[1]           .byte     1
52:[7]           .byte     7
53:[42]          .byte     42
54:[2]           .byte     2
        ..........
59:[4]           .byte     4
60:[3]           .byte     3
61:[4]           .byte     4
62:[1]           .byte     1
```

Exercise

1. Disassemble this binary instruction sequence:

```
6 0 4 7
6 9 0 1 10 0 1 2 26 9 1 30
11 1 0 8 2 1 5 2 32 37 9 1 3
4 7 9 0 38 1 0 7 99
6 9 0 3 7
```

opcode	mnemonic	operands	semantics
0	`call`	*addr*	function call to *addr*
1	`calli`	*reg*	function call to address in *reg*
2	`brg`	*offset*	branch to $\mathrm{pc}+$*offset* if flags for $>$ are set
3	`inc`	*reg*	*reg* $\leftarrow$ *reg* $+1$
4	`bra`	*offset*	branch to $\mathrm{pc}+$ *offset*
5	`jmpi`	*reg*	jump to address in *reg*
6	`prologue`		beginning of function
7	`ret`		return from function

opcode	mnemonic	operands	semantics
8	`load`	$reg_1, (reg_2)$	$reg_1 \leftarrow [reg_2]$
9	`loadi`	reg, imm	$reg \leftarrow imm$
10	`cmpi`	reg, imm	compare *reg* and *imm* and set flags
11	`add`	reg_1, reg_2	$reg_1 \leftarrow reg_1 + reg_2$
12	`brge`	*offset*	branch to $\mathrm{pc}+offset$ if flags for $\geq$ are set
13	`breq`	*offset*	branch to $\mathrm{pc}+offset$ if flags for $=$ are set
14	`store`	$(reg_1), reg_2$	$[reg_1] \leftarrow reg_2$

Why is disassembly hard?

- Variable length instruction sets — overlapping instructions.

Why is disassembly hard?

- Variable length instruction sets — overlapping instructions.
- Mixing data and code — misclassify data as instructions.

Why is disassembly hard?

- Variable length instruction sets — overlapping instructions.
- Mixing data and code — misclassify data as instructions.
- Indirect jumps — must assume that *any* location could be the start of an instruction!

Why is disassembly hard?

- Variable length instruction sets — overlapping instructions.
- Mixing data and code — misclassify data as instructions.
- Indirect jumps — must assume that *any* location could be the start of an instruction!
- Find the beginning of functions if all calls are indirect.

Why is disassembly hard. . . ?

- Finding the end of fuctions — if no dedicated return instruction.

Why is disassembly hard. . . ?

- Finding the end of fuctions — if no dedicated return instruction.
- Handwritten assembly code — won't conform to the standard calling conventions.

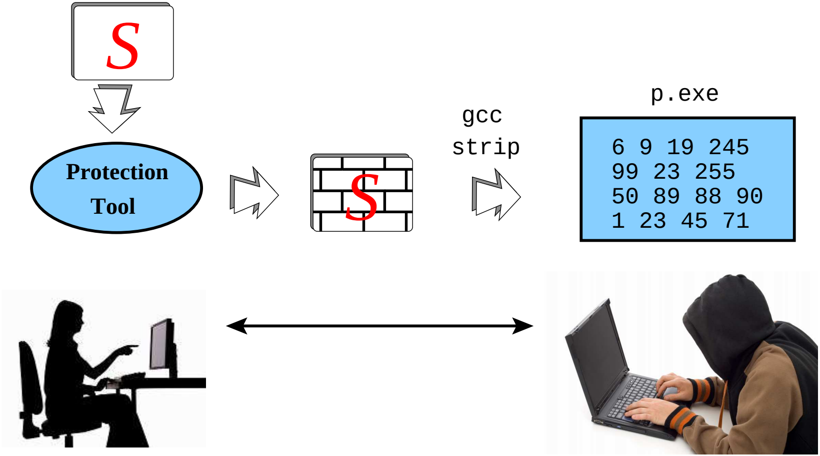
Why is disassembly hard...?

- Finding the end of fuctions — if no dedicated return instruction.
- Handwritten assembly code — won't conform to the standard calling conventions.
- code compression — the code of two functions may overlap.

Why is disassembly hard...?

- Finding the end of fuctions — if no dedicated return instruction.
- Handwritten assembly code — won't conform to the standard calling conventions.
- code compression — the code of two functions may overlap.
- Self-modifying code.

Decompilation


S
Protection
Tool
S
gcc
strip
p.exe
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71

p.as

```
mov R1,R2
add R2,R2
jmp L3
L4: call Q
```

p.exe

```
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71
```

p.c

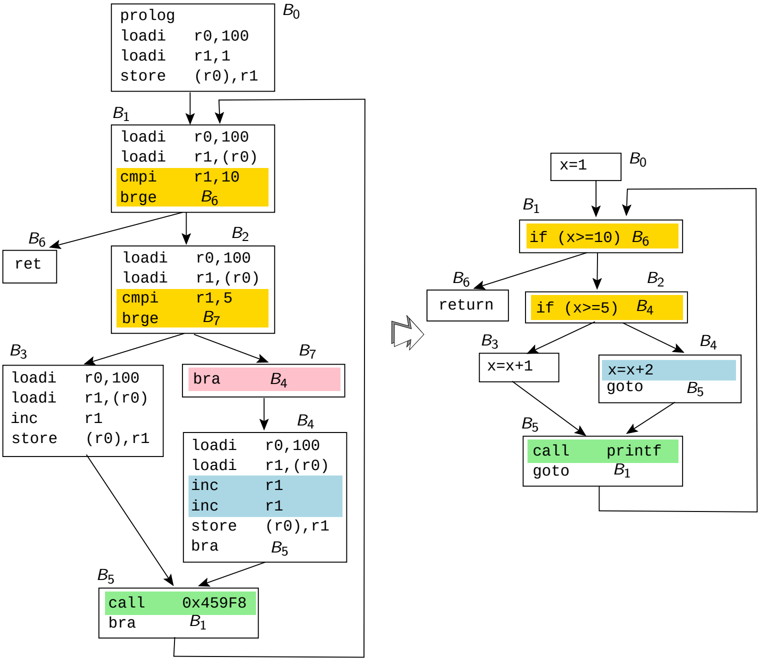
```
while (...){
   a += b;
   Q();
}
```

p.as

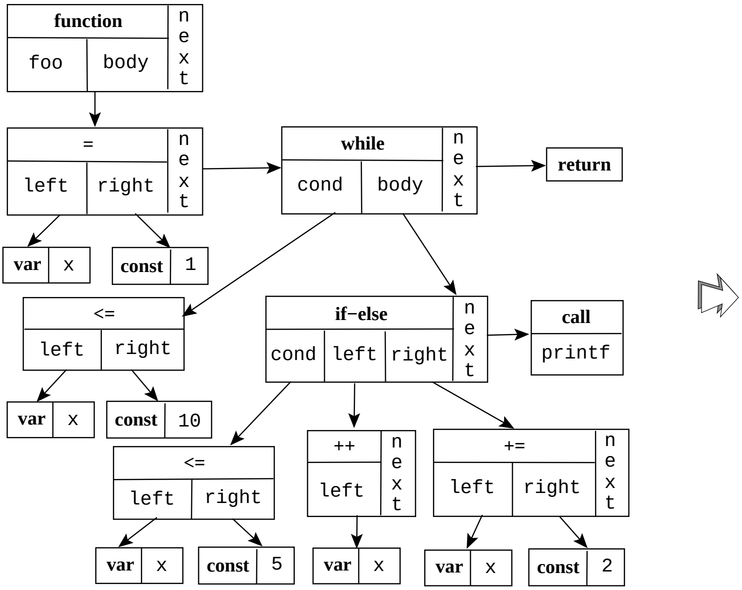
```
mov R1,R2
add R2,R2
jmp L3
L4: call Q
```

p.exe

```
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71
```



B0
prolog
loadi r0,100
loadi r1,1
store (r0),r1
B1
loadi r0,100
loadi r1,(r0)
cmpi r1,10
brge B6
B6
ret
B2
loadi r0,100
loadi r1,(r0)
cmpi r1,5
brge B7
B3
loadi r0,100
loadi r1,(r0)
inc r1
store (r0),r1
B7
bra B4
B4
loadi r0,100
loadi r1,(r0)
inc r1
inc r1
store (r0),r1
bra B5
B5
call 0x459F8
bra B1
B0
x=1
B1
if (x>=10) B6
B6
return
B2
if (x>=5) B4
B3
x=x+1
B4
x=x+2
goto B5
B5
call printf
goto B1


function
foo
body
next
=
left
right
next
var
x
const
1
while
cond
body
next
return
<=
left
right
var
x
const
10
if−else
cond
left
right
next
call
printf
<=
left
right
var
x
const
5
++
left
next
var
x
+=
left
right
next
var
x
const
2

```
void foo() {
    x=1;
    while (x<10} {
        if (x<5)
            x++;
        else
            x+=2;
        printf();
    }
}
```

What's so hard about decompilation?

- Disassembly — first step of any decompiler!

What's so hard about decompilation?

- Disassembly — first step of any decompiler!
- Target language — assembly code may not correspond to any legal source code.

What's so hard about decompilation?

- Disassembly — first step of any decompiler!
- Target language — assembly code may not correspond to any legal source code.
- Standard library functions — (`call printf()` $\Rightarrow$ `call foo96()`).

What's so hard about decompilation?

- Disassembly — first step of any decompiler!
- Target language — assembly code may not correspond to any legal source code.
- Standard library functions — (`call printf()` ⇒ `call foo96()`).
- Idioms of different compilers (`xor r0,r0`⇒`r0=0`).

What's so hard about decompilation?

- Artifacts of the target architecture (unnecessary jumps-to-jumps).

What's so hard about decompilation?

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow — from mess of machine code branches.

What's so hard about decompilation?

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow — from mess of machine code branches.
- Compiler optimizations — undo loop unrolling, shifts and adds $\Rightarrow$ original multiplication by a constant.

What's so hard about decompilation?

- Artifacts of the target architecture (unnecessary jumps-to-jumps).
- Structured control-flow — from mess of machine code branches.
- Compiler optimizations — undo loop unrolling, shifts and adds $\Rightarrow$ original multiplication by a constant.
- Loads/stores – $\Rightarrow$ operations on arrays, records, pointers, and objects.