Software Protection Research

ISSISP 2017 — Program Analysis

Christian Collberg

Department of Computer Science
University of Arizona

http://collberg.cs.arizona.edu

collberg@gmail.com

Supported by NSF grants 1525820 and 1318955 and
by the private foundation that shall not be named

What is Program Analysis?

Control Flow Analysis

Discussion

What is Program Analysis?

Program Analysis

```
int foo() {
  int x;
  int* y;
  printf(x+*y);
}
```

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

- ***Defenders**: need to analyze their program to protect it!*

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

Tigress

```
int foo() {
  … … … …
}
```

Obfuscate

```
int foo'() {
  … … … …
}
```

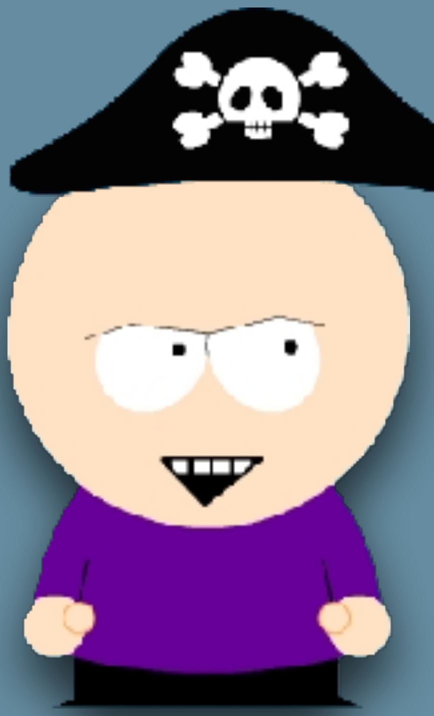
- ***Attackers**: need to analyze our program to modify it!*

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

```
int foo'() {
  … … … …
}
```

De-obfuscate

- Extract Code!
- Discover Algorithms!
- Find Design!
- Find Keys!
- Modify Code!



Two kinds of analyses:

- **static analysis:** collect information about a program by studying its code;
- **dynamic analysis:** collect information from *executing* the program.

- **static analysis:** collect information about a program by studying its code

```
int foo() {
  … … … …
}
```

Static Analysis

I.e. we analyze the source or binary code of the program itself.

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

- **dynamic analysis:** collect information from *executing* the program.

```
int foo() {
  … … … …
}
```

Program inputs

Dynamic Analysis

I.e. we analyze a *trace* of the program as it is running on some *particular input*

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

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.

Dynamic Analyses

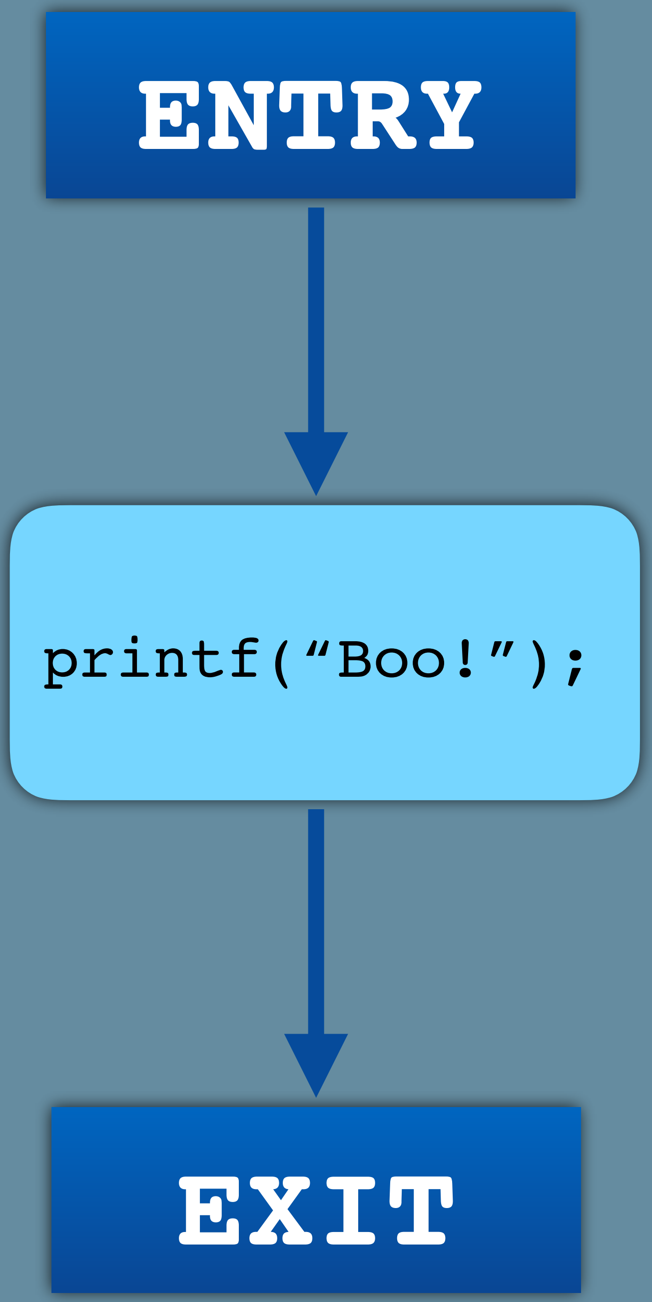
- **debugging:** what path does the program take?
- **tracing:** which functions/system calls get executed?
- **profiling:** what gets executed the most?

Control Flow Analysis

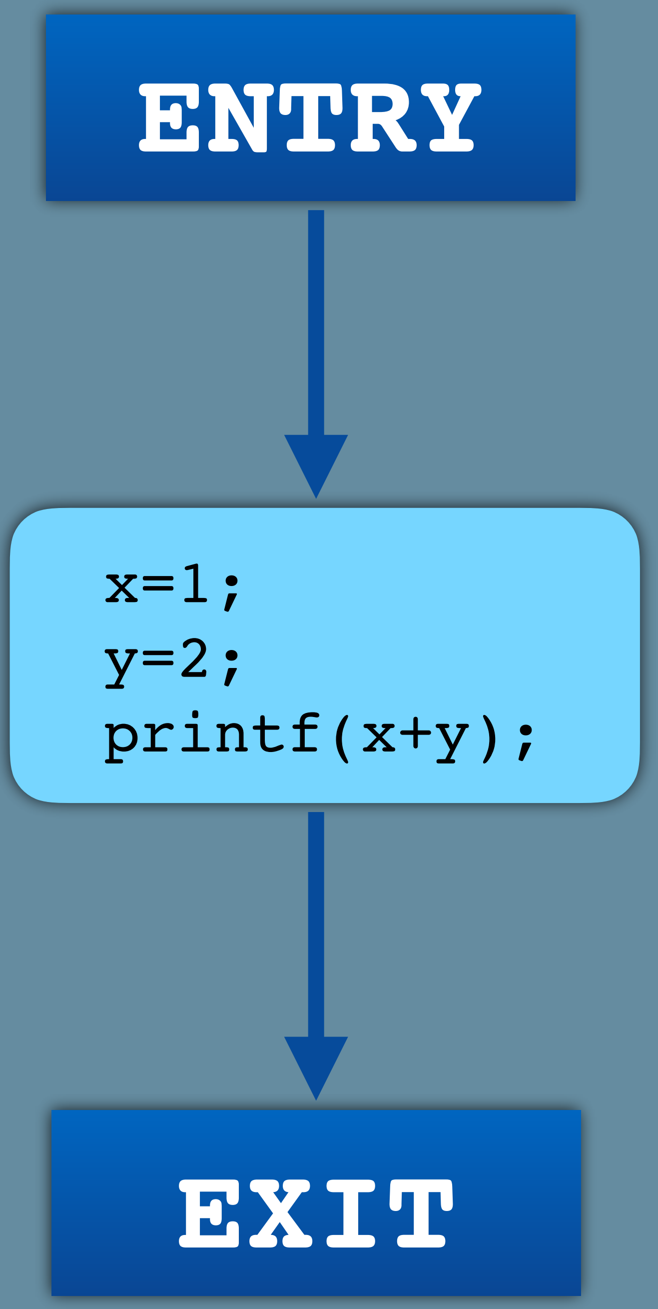
Control-Flow Graph (CFG)

- A way to represent the possible flow of control inside a function.
- **Nodes**: called **basic blocks.** Each block consists of straight-line code ending (possibly) in a branch.
- **Edges**: An edge A → B means that control could flow from A to B.
- There is one unique **entry node** and one unique **exit node**.

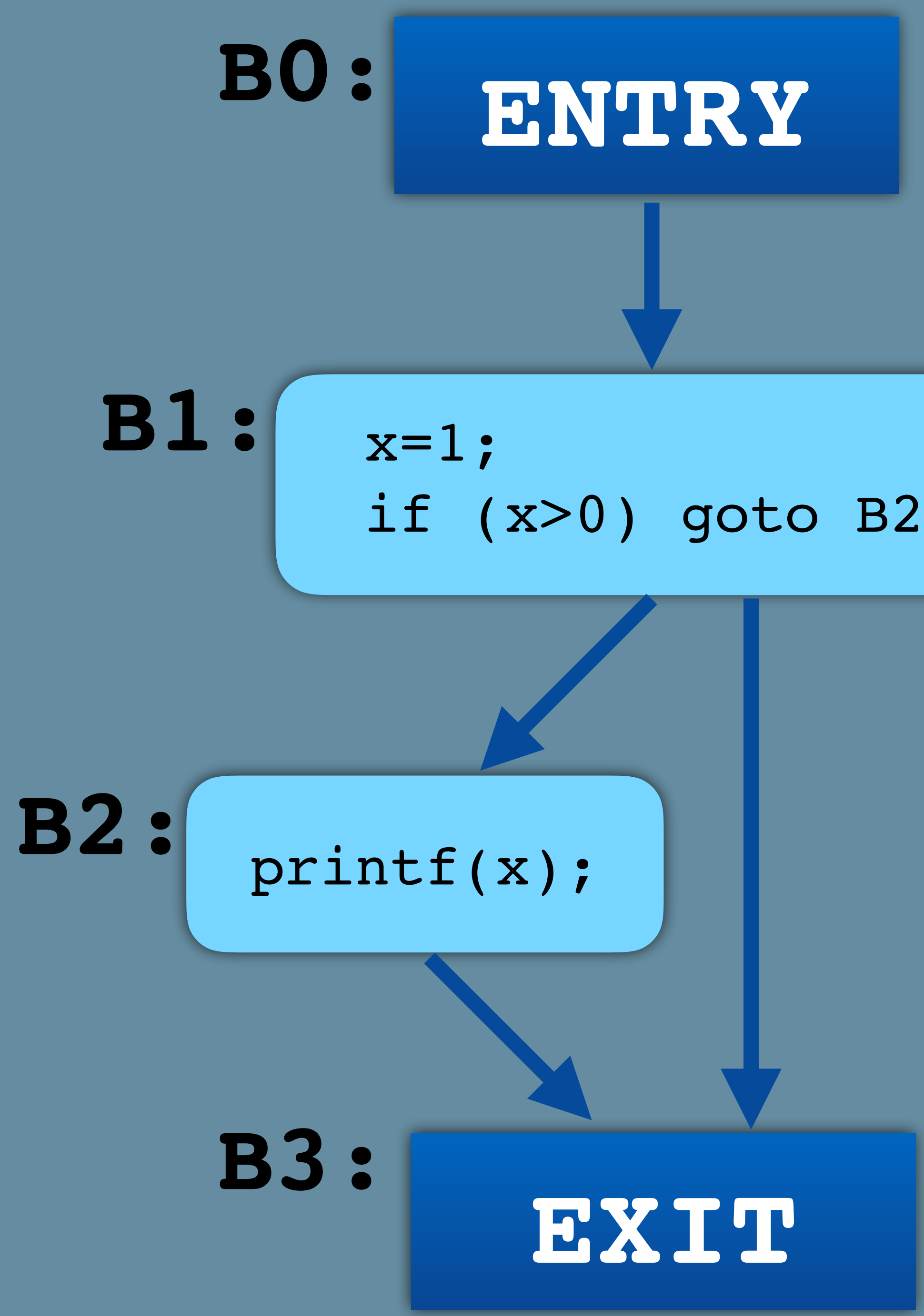
```
int foo() {
   printf("Boo!");
}
```



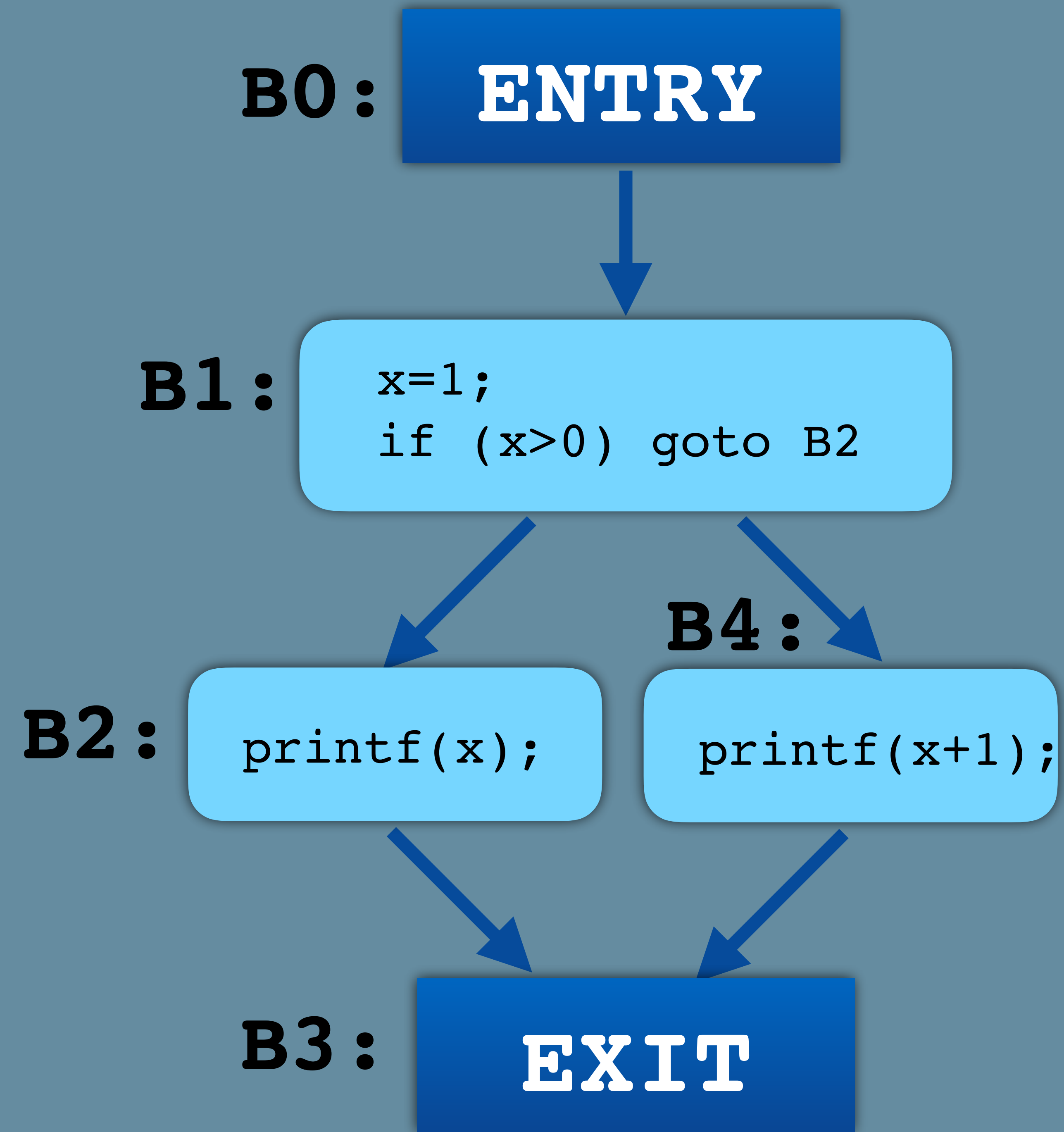
```c
int foo() {
    x=1;
    y=2;
    printf(x+y);
}
```



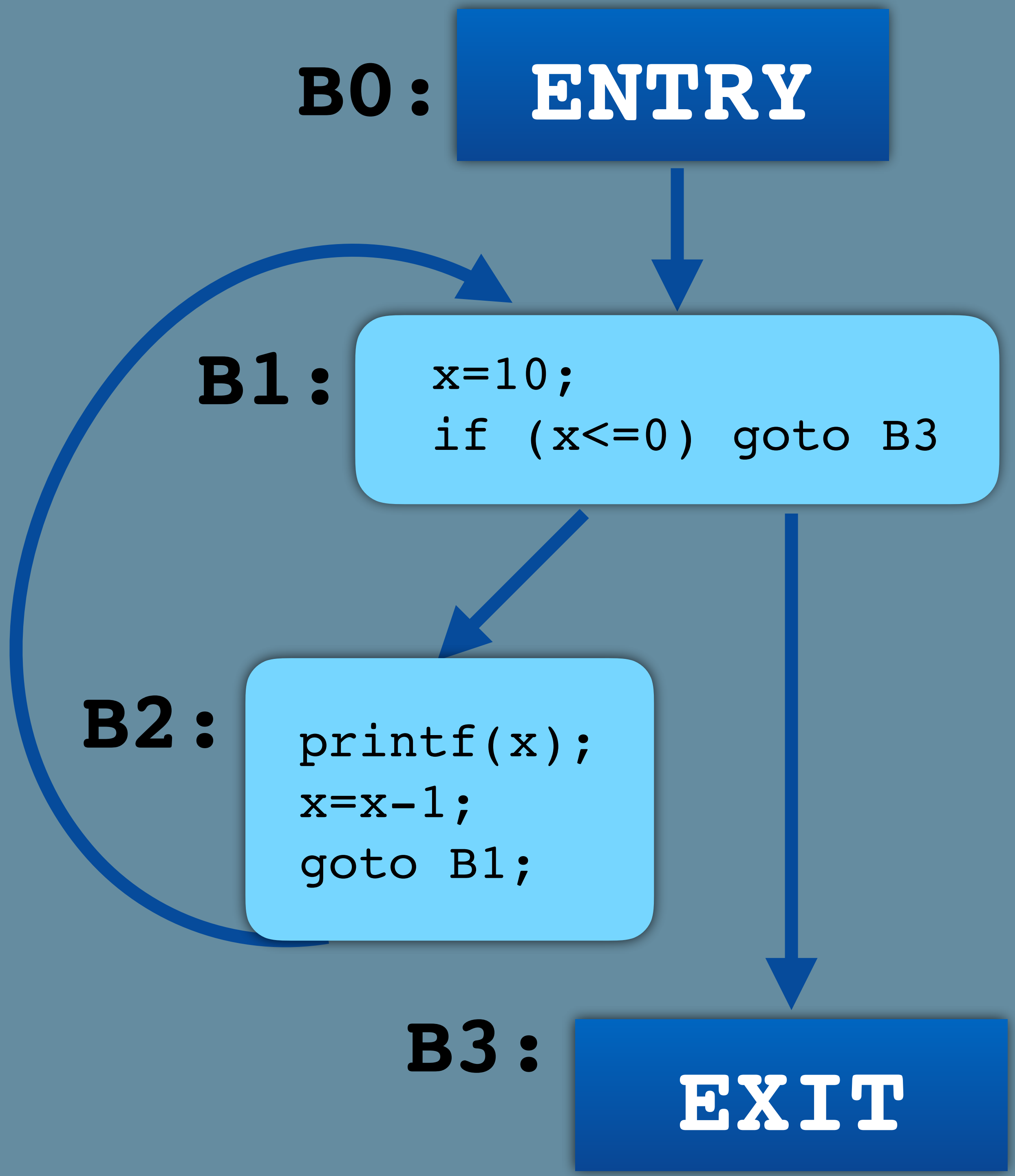
```
int foo() {
  read(x);
  if (x>0)
      printf(x);
}
```



```c
int foo() {
  read(x);
  if (x>0)
    printf(x);
  else
    printf(x+1);
}
```



```
int foo() {
  x=10;
  while (x>0){
    printf(x);
    x=x-1;
  }
}
```



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

Exercise!

```
X ← 20;
while (X<10){
    X←X-1;
    A[X]←10;
    if (X=4)
        X←X-2;
};
Y←X+5;
```

←

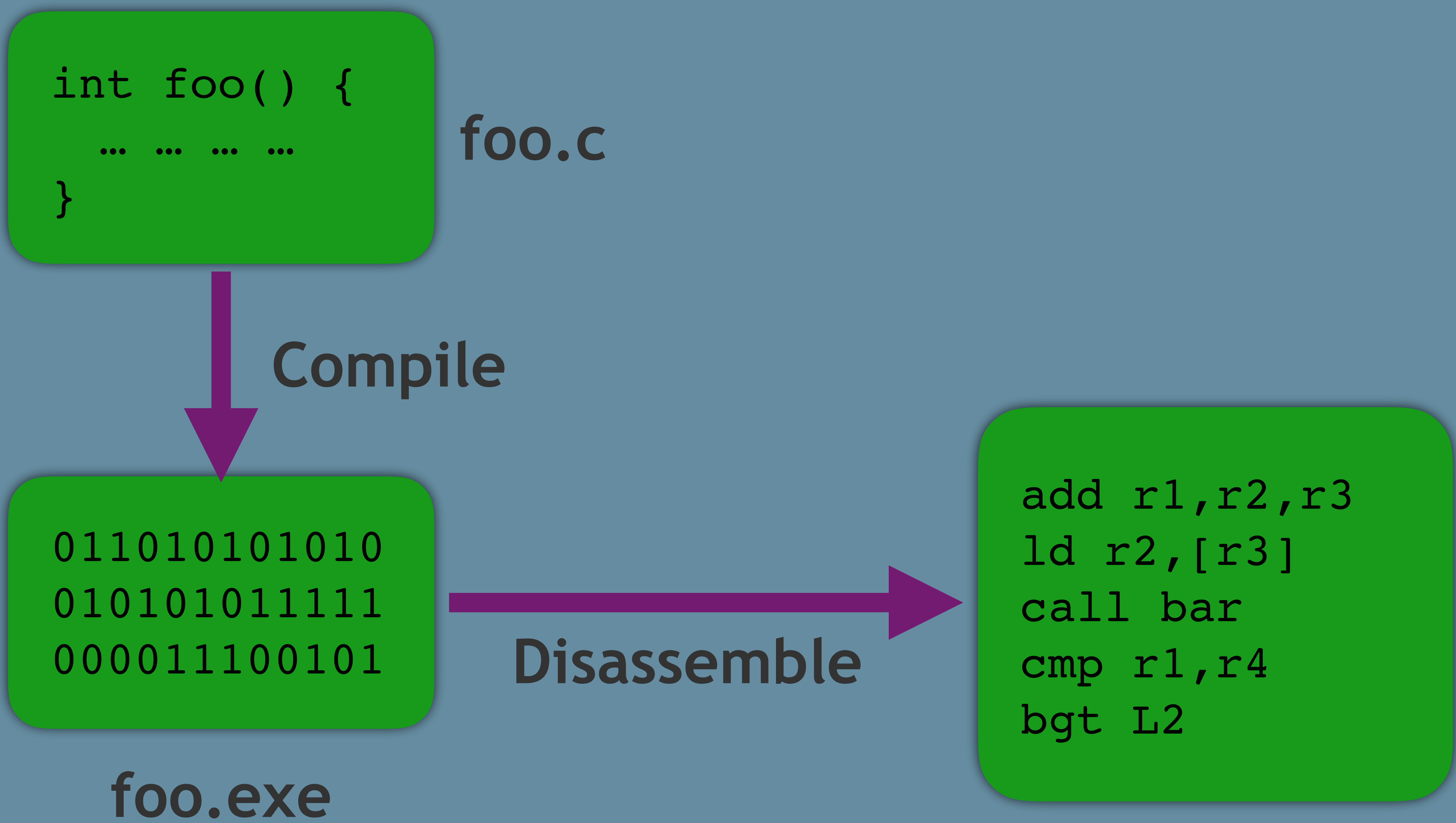
```
1: X←20
2: if X>=10 goto (8)
3: X←X-1
4: A[X]←10
5: if X!=4 goto (7)
6: X←X-2
7: goto (2)
8: Y←X+5
```

Convert to CFG! First simplify!

Work with your friends!!!

Disassembly

- ***Attackers**: prefer looking at assembly code than machine code*

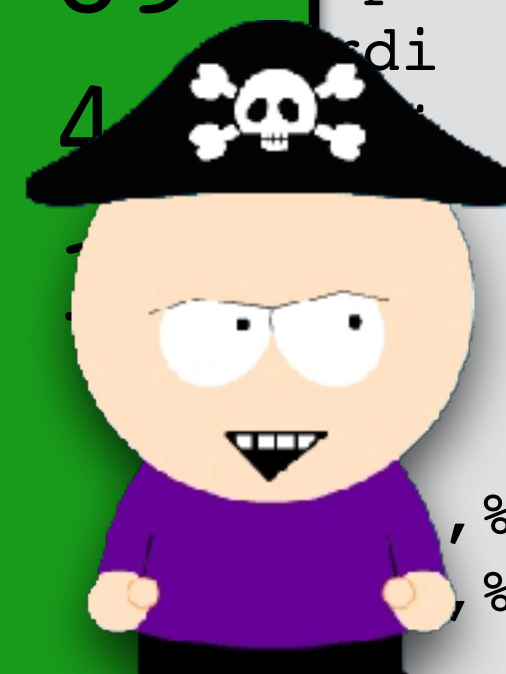


Static Disassembly

```
objdump -d i/bin/ls | less
```

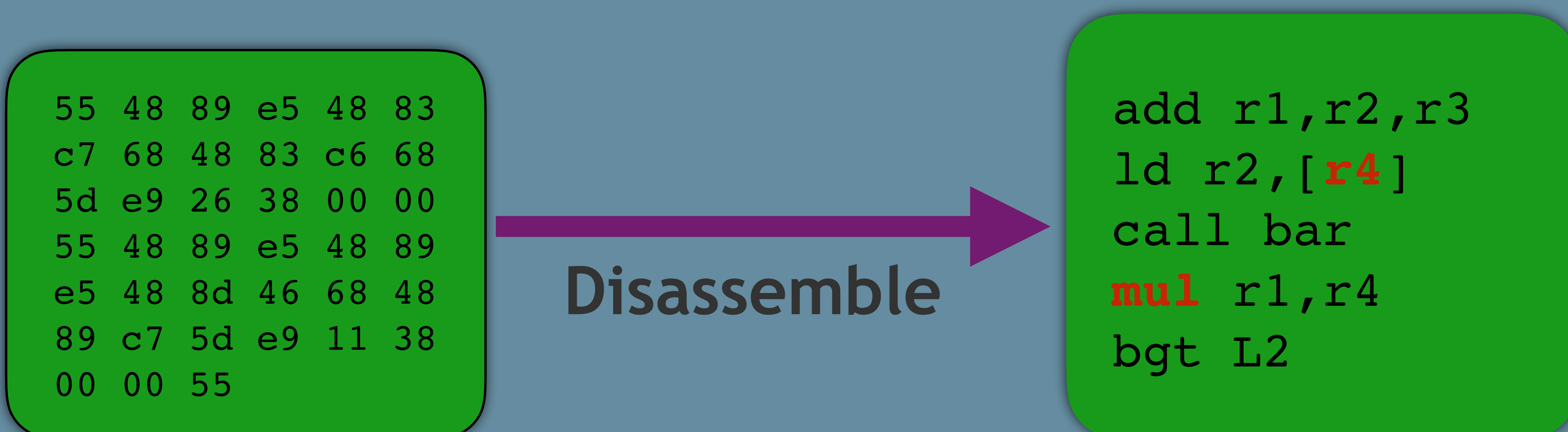
Address Code bytes Assembly

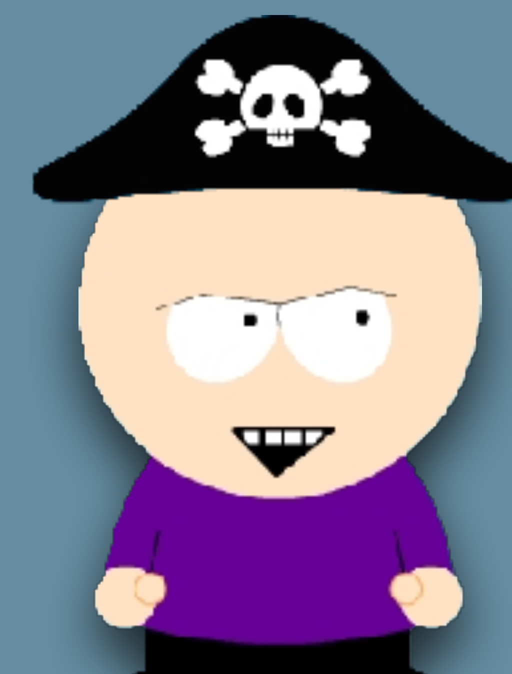
```
55 48 89 e5 48 83 c7
68 48 83 c6 68 5d e9
26 38 00 00 55 48 89
e5 48 89 e5 48 8d 4
68 48 89 c7 5d e9 1
38 00 00 55
```



```
13.  0xd9a:  e9 11 38 00 00   jmpq   1000045b0
14.  0xd9f:  55               push   %rbp
```

- *Disassembly is hard! And sometimes disassemblers get it wrong!*





  - *In general, this is always the case: program analysis is more or less precise.*

- *There are two general algorithm ideas for disassembly:*
    1. *Linear Sweep Traversal*
    2. *Recursive Traversal*
- *At times, both with fail.*
- *We typically add heuristics to improve precision.*

```
55 48 89 e5 48 83 c7 68
48 83 c6 68 5d e9 26 38
00 00 55
```

1. push   %rbp
2. mov   %rsp,%rbp
3. add $0x68,%rdi
4. add $0x68,%rsi
5. pop   %rbp
6. jmpq   1000045b0
7. push   %rbp

Linear sweep disassembly

```
55 48 89 e5 48 83 c7 68
48 83 c6 68 5d e9 26 38
00 00 55
```

```
1. 0xd78   push %rbp
2. 0xd79:  mov  %rsp,%rbp
3. 0xd7c:  add  $0x68,%rdi
4. 0xd80:  add  $0x68,%rsi
5. 0xd84:  pop  %rbp
6. 0xd85:  jmpq 0x45b0
7. 0xd8a   push %rbp
```

Recursive traversal disassembly

Stack

0xd8a

Exercise!

```
0xd78: push %rbp
0xd79: mov  %rsp,%rbp
0xd7c: add  $0x68,%rdi
0xd80: add  $0x68,%rsi
0xd84: pop  %rbp
0xd85: jmpq 0x45b0
0xd8a: .byte 0x55
0xd8b: mov  %rdi,%rbp
```

`0x55 ≡ push %rbp!!!`

- How would a **linear sweep** disassembly handle this code?

Exercise!

```
1. 0xd78: push %rbp
2. 0xd79: mov  %rsp,%rbp
3. 0xd7c: add  $0x68,%rdi
4. 0xd80: add  $0x68,%rsi
5. 0xd84: pop  %rbp
6. 0xd85: jmpr %rdi
7.0xd8b: mov  %rdi,%rbp
```

Indirect jump!

- How would a **recursive traversal** disassembly handle this code?



Questions?