ROP Defenses: Control-Flow Integrity

With material from Mike Hicks, Dave Levin, and Michelle Mazurek

Let's say that I want to call D01F and **then** F019

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

To “set up” the attack we put 0xD01F in saved RIP

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

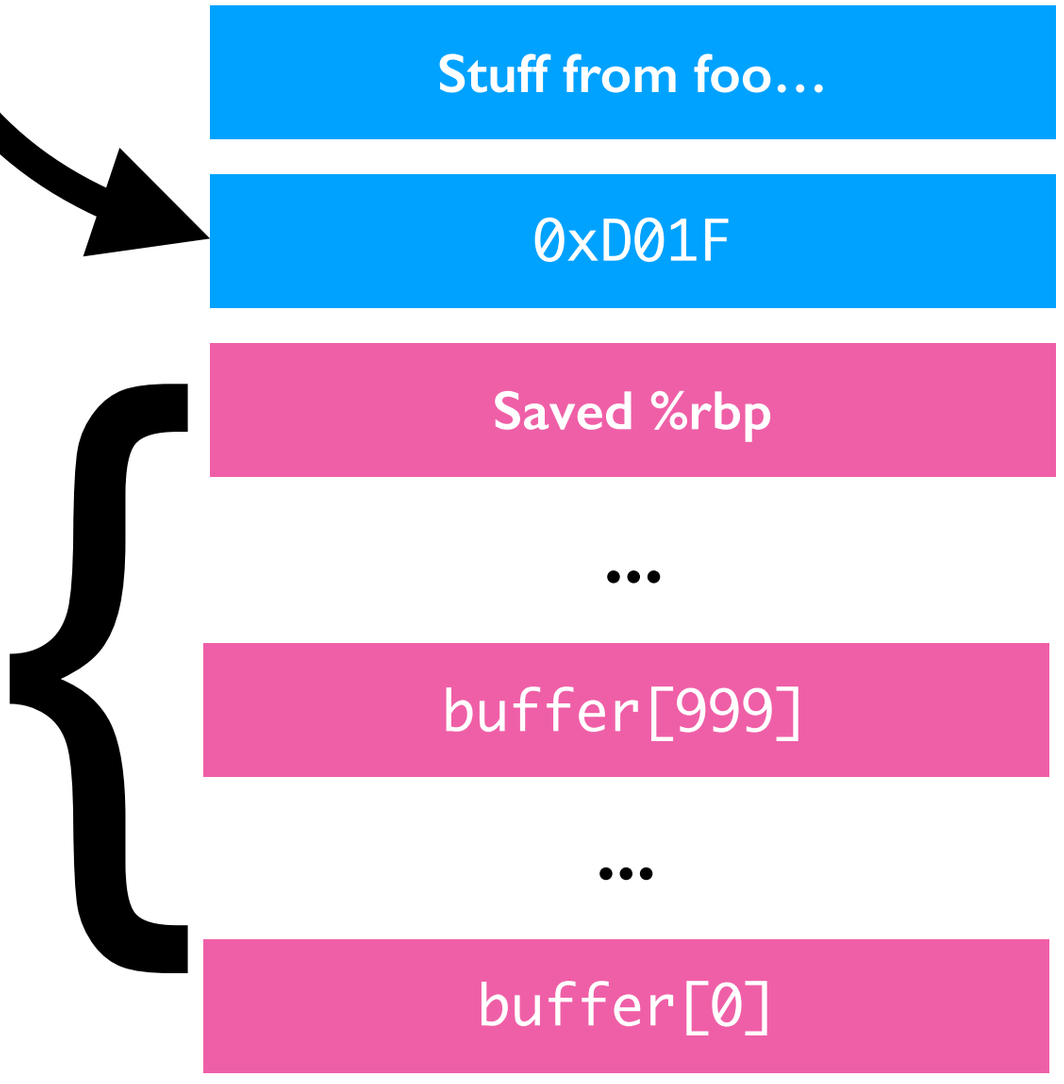
Bar’s frame

To “set up” the attack we put 0xD01F in saved RIP

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Bar’s frame

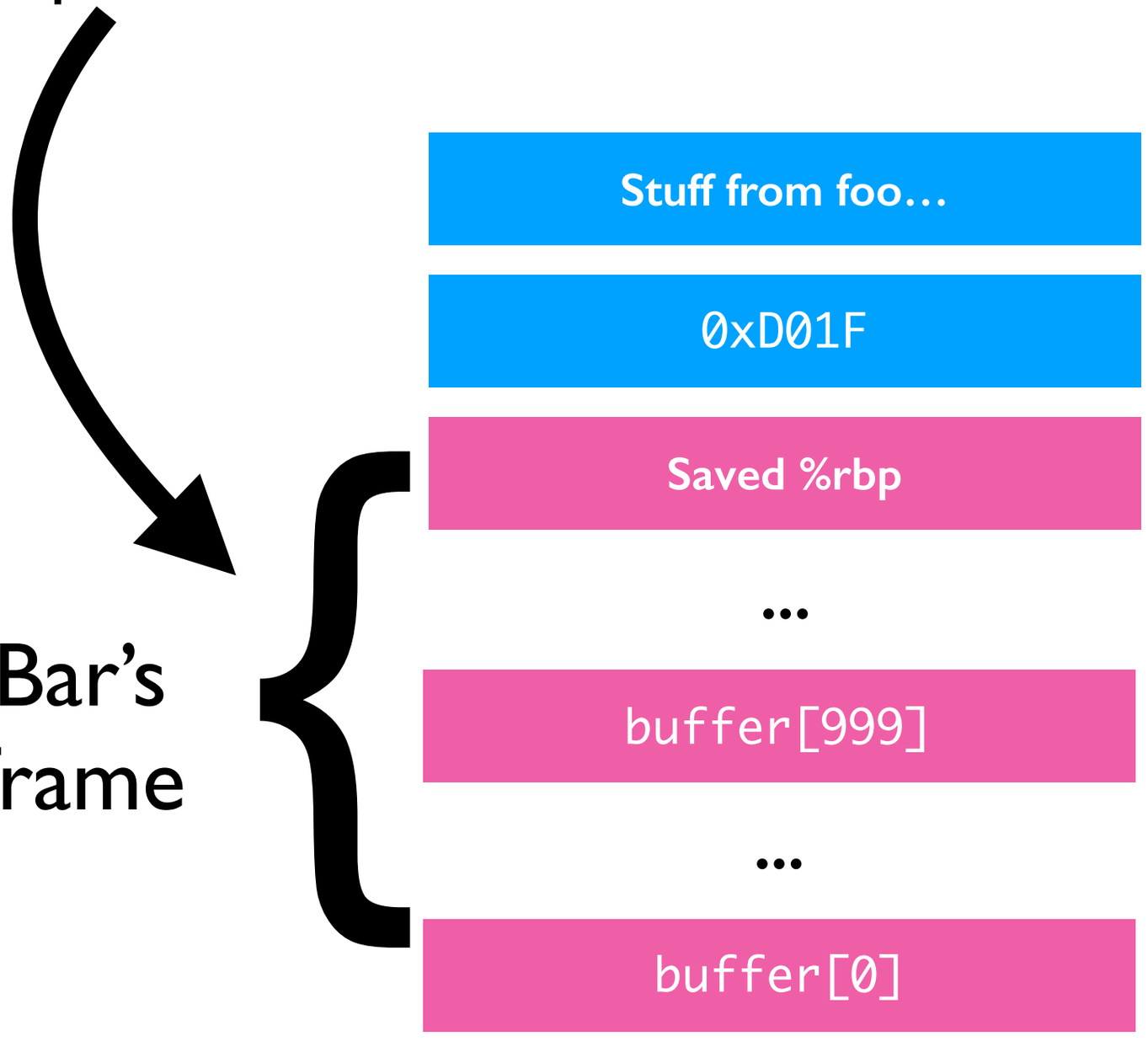


Before **foo** returns, it pops all of this stuff from the stack

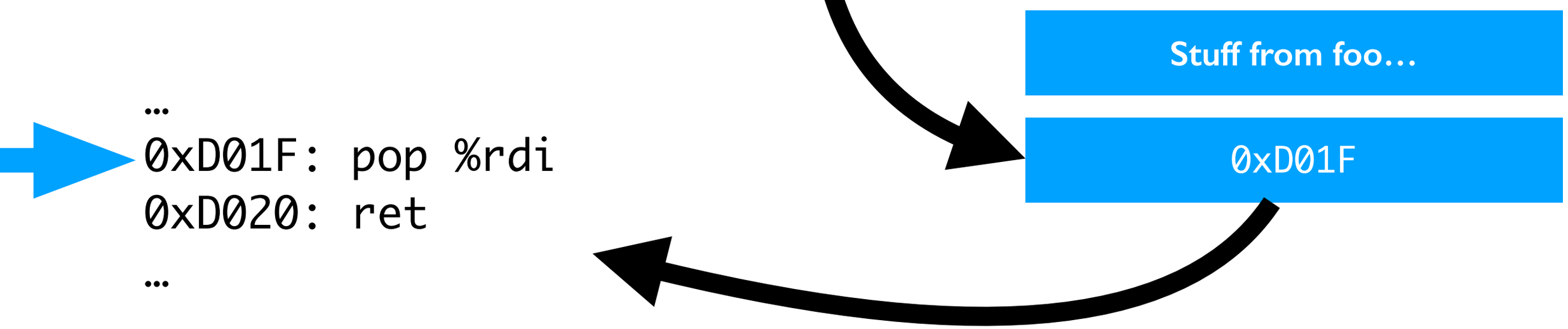
```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Bar's frame



Now it goes here




```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

(Rather than it's caller **foo**)

Super Critical: pops 0xD01F from stack!

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

%rsp

Stuff from foo…

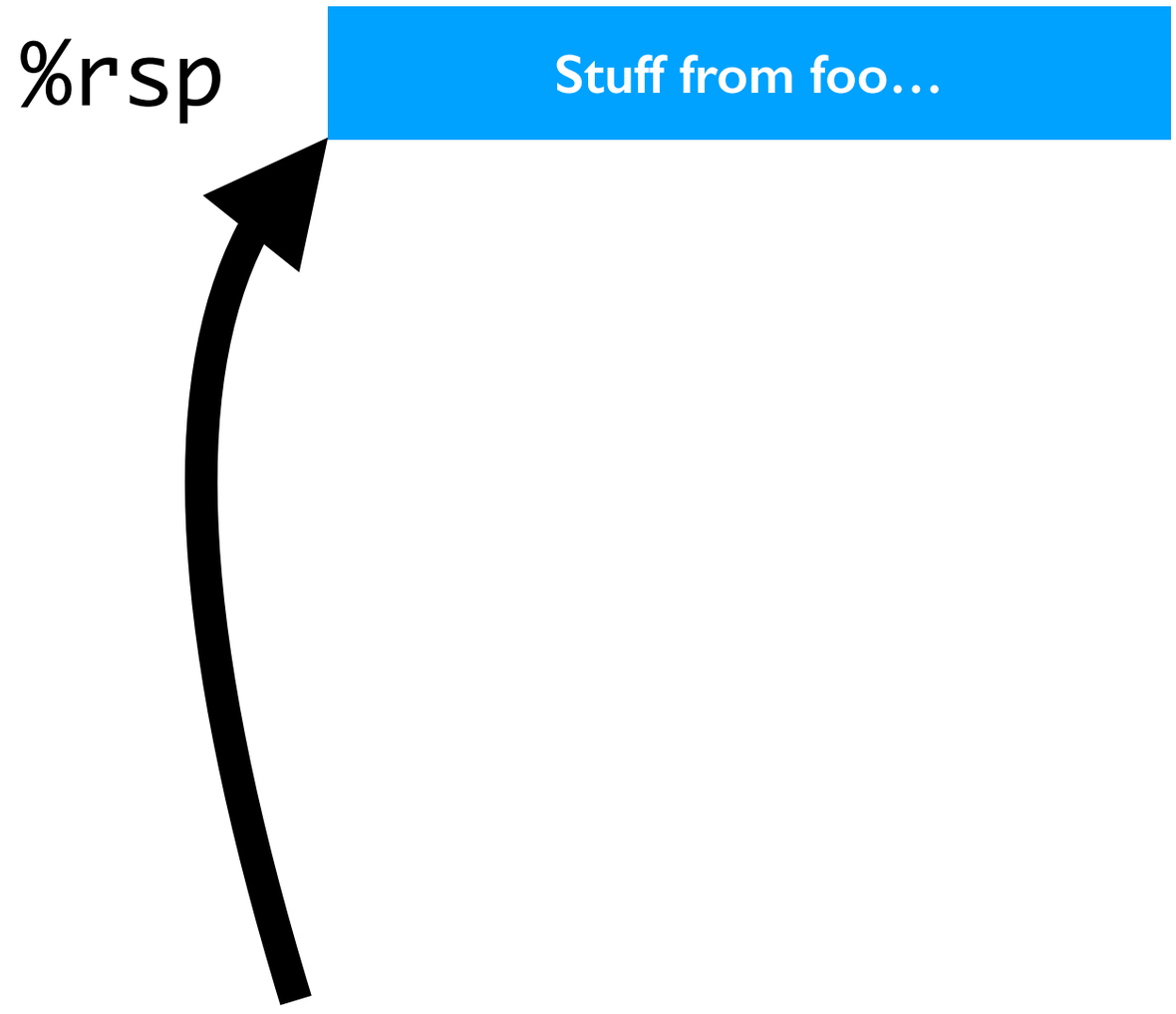
```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

So **now** whatever's on stack will be popped into %rdi

(Which is previously stuff in **foo**'s stack)

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

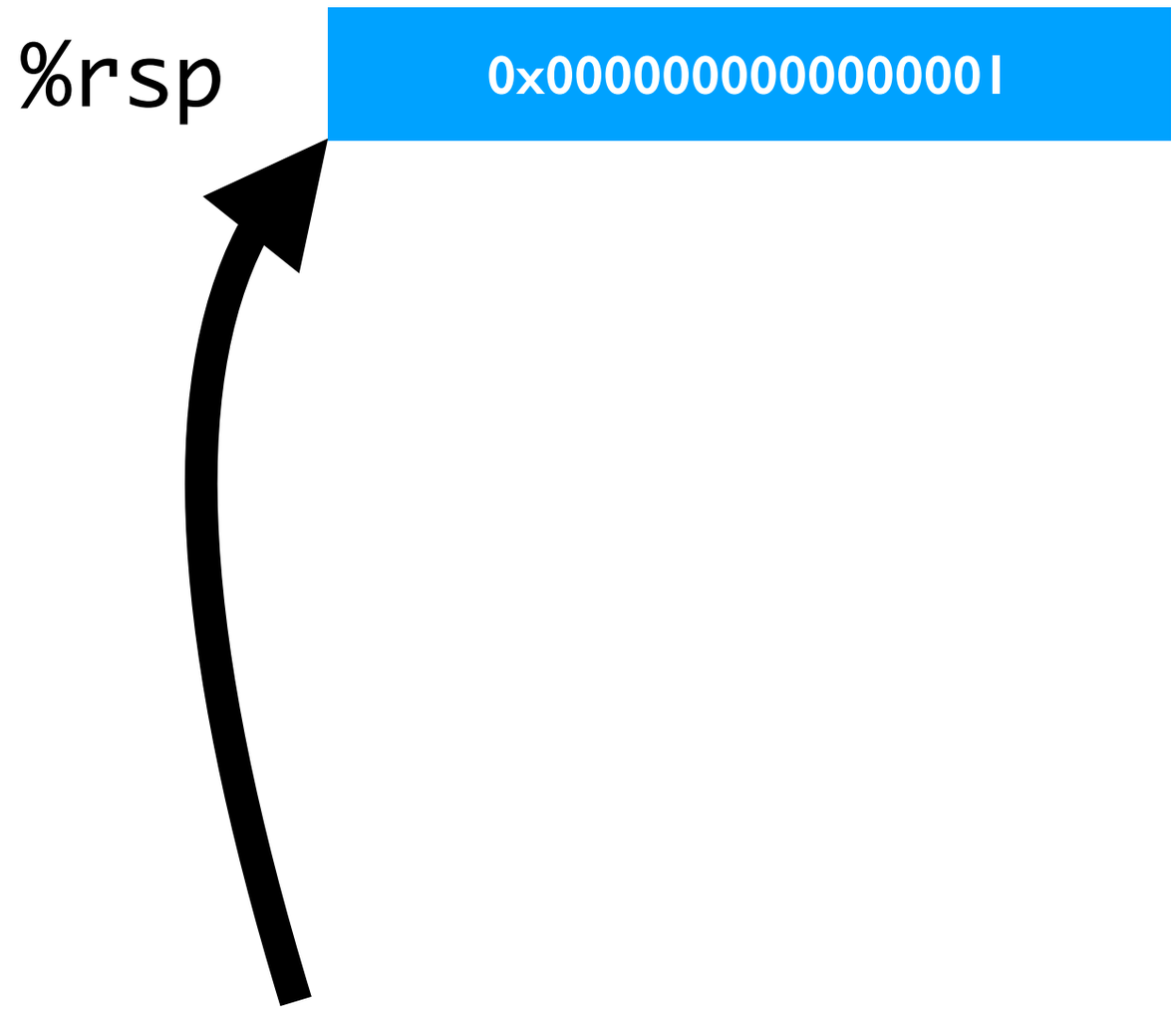


So if I want to put 1 in RDI, I put it **here**

(Which is previously stuff in **foo**'s frame)

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```



So if I want to put 1 in RDI, I put it **here**

(Which is previously stuff in **foo**'s frame)

%rsp 


```
…
0xD01F: pop %rdi
0xD020: ret
…
```

Now, when the code hits **this** point, it's going to execute a return

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Which will **yet again** go to whatever address is in %rsp

%rsp 0xF019

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Critical observation: if %rsp is **now** 0xF019, we'll get what we want

%rsp 0xF019

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Critical observation: if %rsp is **now** 0xF019, we'll get what we want

- Set %rdi to 1 (arg for exit)
- Set %rax to 60 (exit)
- Execute the “syscall” instruction

```
…
0xD01F: pop %rdi
0xD020: ret
…
```

```
…
0xF019: mov $60, %rax
0xF01B: syscall
0xF01C: ret
…
```

Critical observation: if %rsp is **now** 0xF019, we’ll get what we want

Observation: We can chain multiple sequences (that all end in ret) by setting up the stack right

Exercise

```
write(1, “Hello, world!”, 13);
```

%rax = 1 %rdi = 1 %rsi = &”Hello, world”, %rdx = 13

```
0xC110: pop %rsi
0xC112: ret
```

```
0x1029: pop %edx
0x102a: ret
```

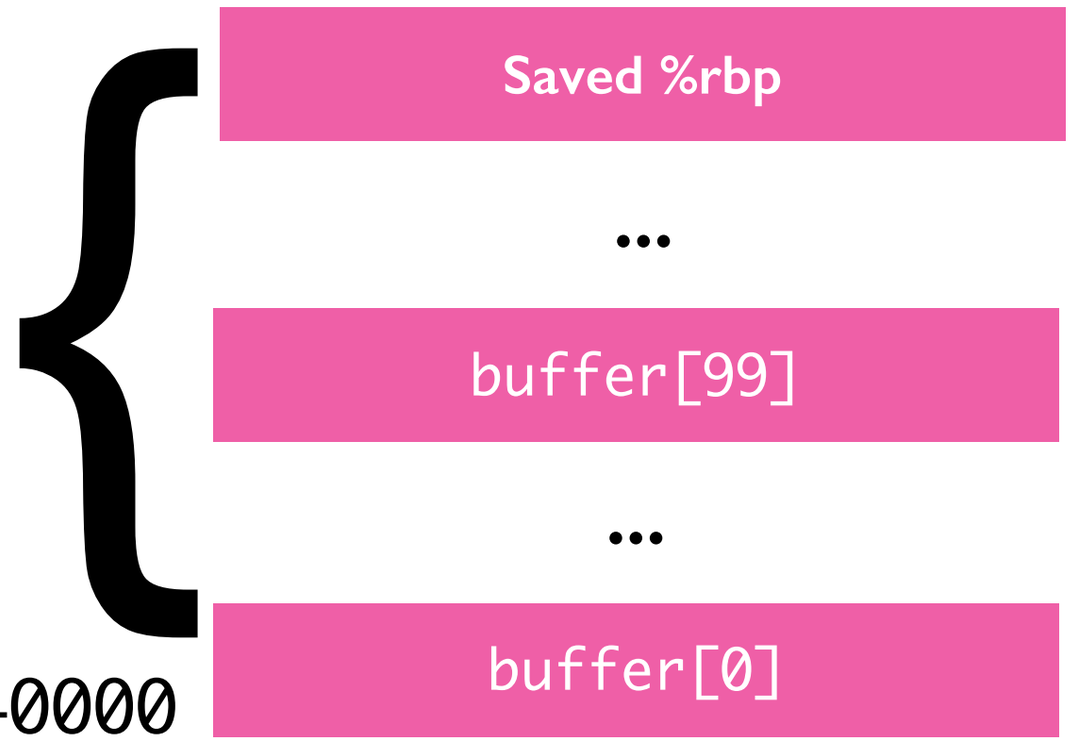
```
0xD235: xchang %rdx, %rdi
0xD238: ret
```

```
0xF019: pop %eax
0xF01B: ret
```

```
0xB0FF: pop %rdx
0xB102: ret
```

```
0xCA2F: syscall
```

Assume this is 128

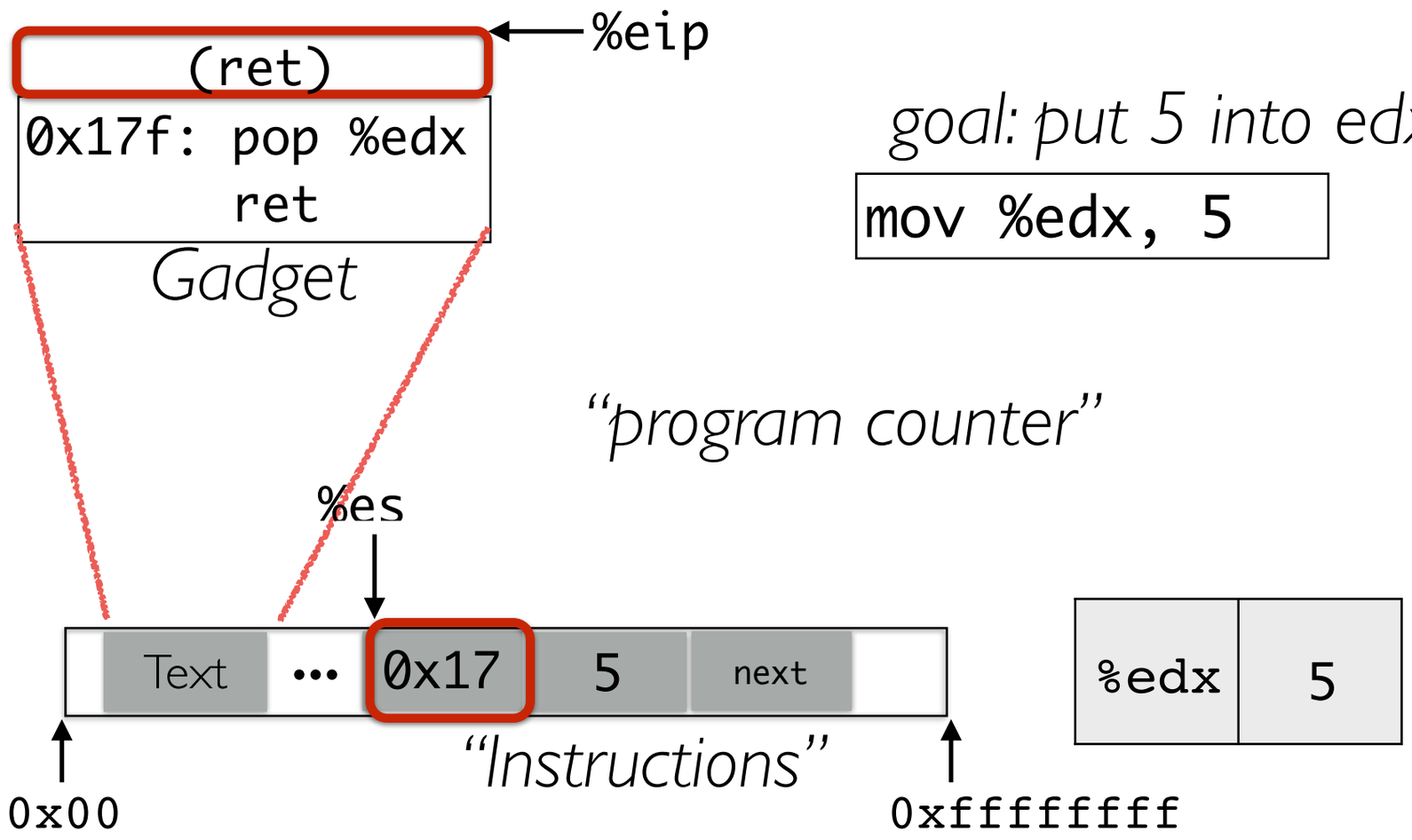


buffer = 0x40000

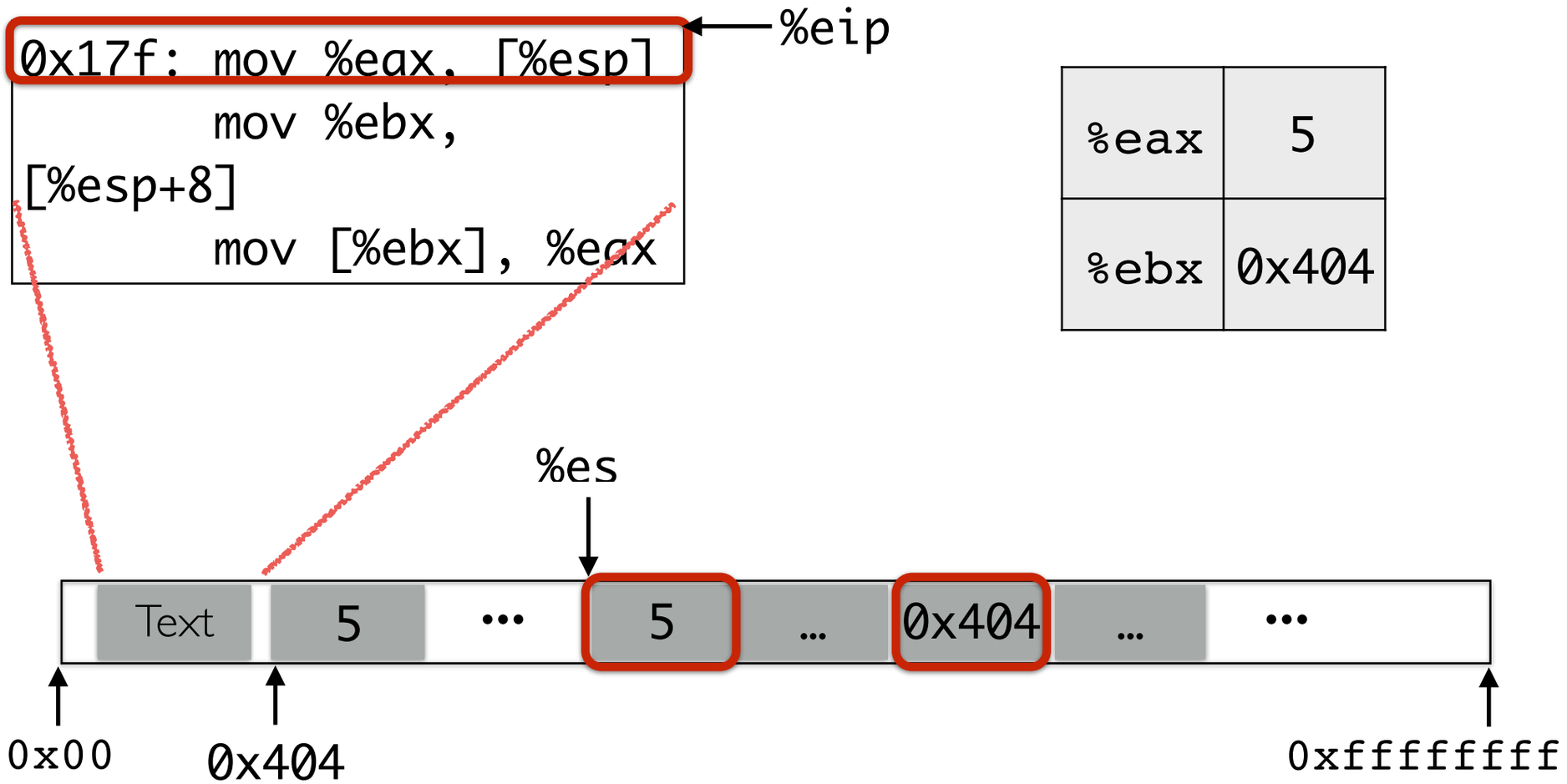
Approach

- Gadgets are instruction groups that end with **`ret`**
- Stack serves as the code
  - **`%esp`** = program counter
    - Gadgets invoked via **`ret`** instruction
    - Gadgets get their arguments via **`pop`**, etc.
      - Also on the stack

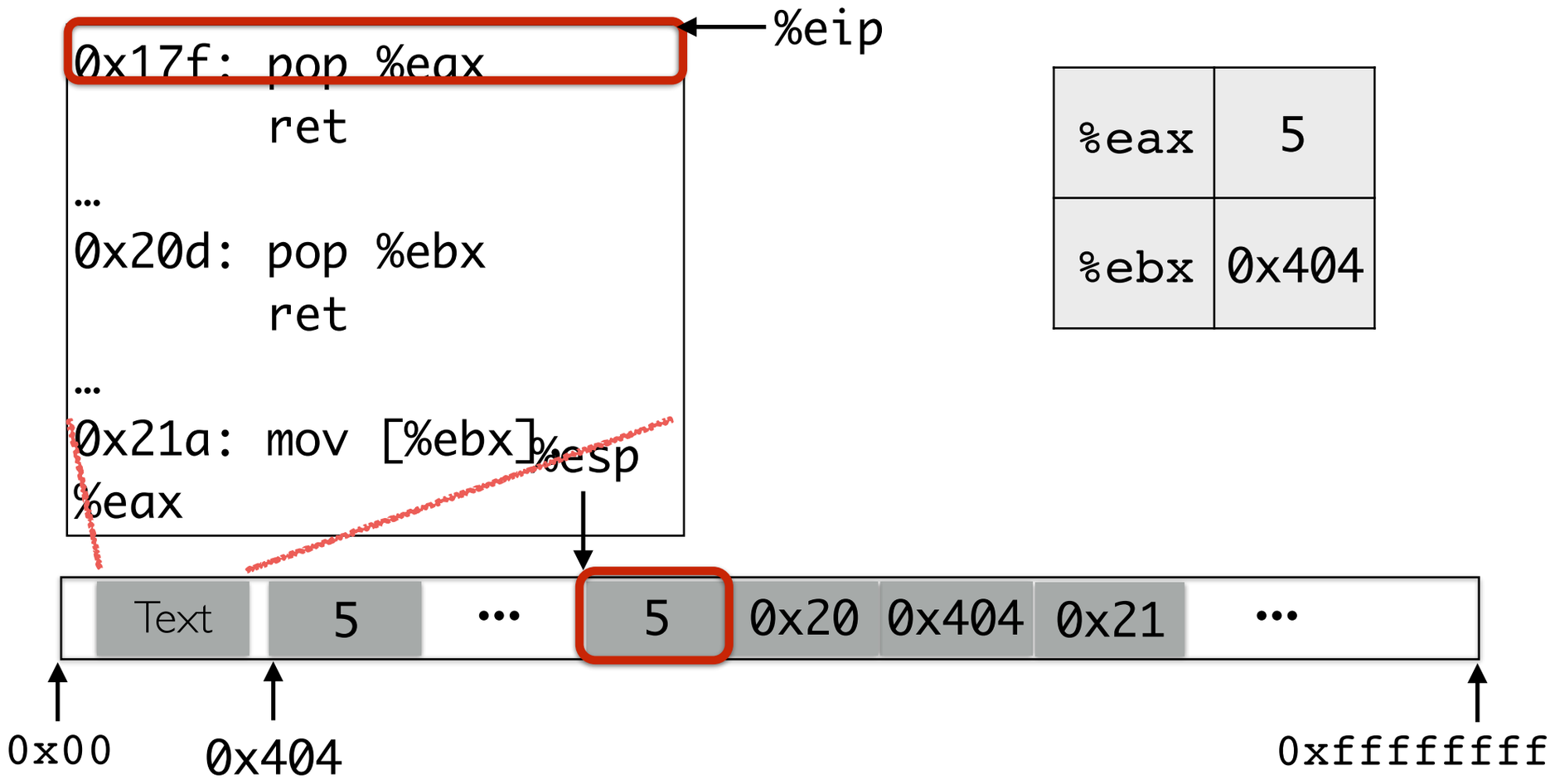
Simple example



Code sequence (no ROP)



Equivalent ROP sequence



Whence the gadgets?

- How can we find gadgets to construct an exploit?
  - Automated search: look for **`ret`** instructions, work backwards
    - Cf. https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?
  - For significant codebases (e.g., libc), **Turing complete**
    - Especially true on x86's dense instruction set
  - Schwartz et al. (USENIX Sec'11) automated gadget shellcode creation, Turing complete not required

Blind ROP

- **Defense: Randomizing the location of the code** (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response**: **Blind ROP**

- If server restarts on a crash, but does not re-randomize:

  1. Read the stack to **leak canaries and a return address**
  2. Find a few gadgets (at run-time) to **effect call to `write`**
  3. **Dump binary to find gadgets for shellcode**

http://www.scs.stanford.edu/brop/

Blind ROP, continued

- Able to **completely automatically**, only **through remote interactions**, develop a **remote code exploit for nginx**, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization
- Conclusion: Are avoidance defenses hopeless?
- Put another way: **Memory safety is really useful!**

Today

- Finish up memory safety:
  - Finish CFI
  - Rules for secure coding in C
- Move on to malware
  - Viruses
  - Worms
  - Case studies
  - “Modern” malware

Control Flow Integrity

Behavior-based detection

- Stack canaries, non-executable data, ASLR make standard attacks harder / more complicated, but may not stop them

- Idea: **observe** the program's **behavior** — **is it doing what we expect it to?**
  - If not, might be compromised

- Challenges
  - Define "expected behavior"
  - Detect deviations from expectation efficiently
  - Avoid compromise of the detector

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

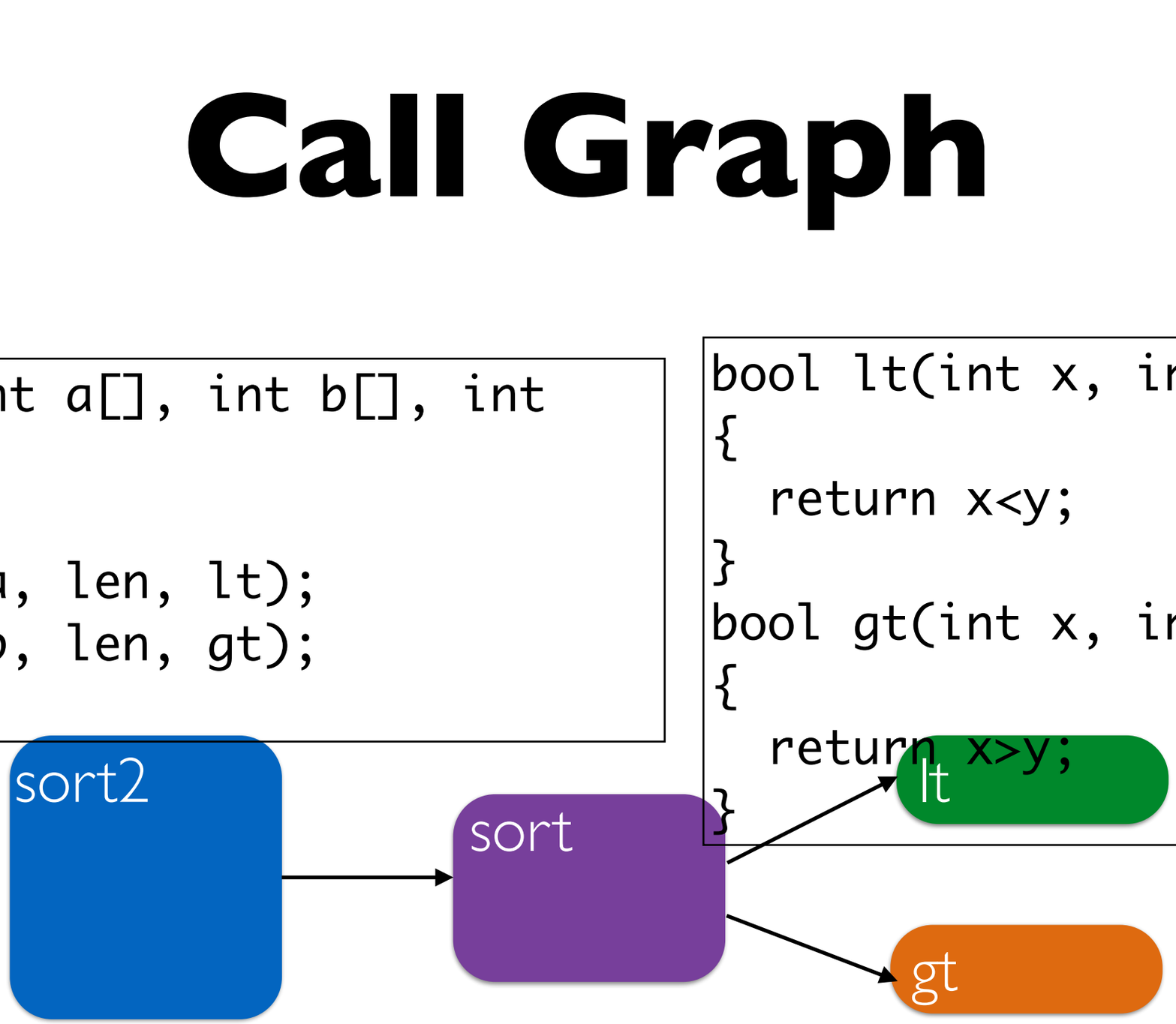
- *Detect deviations from expectation efficiently*

- *Avoid compromise of the detector*

Call Graph

```
sort2(int a[], int b[], int
len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y)
{
  return x<y;
}
bool gt(int x, int y)
{
  return x>y;
}
```

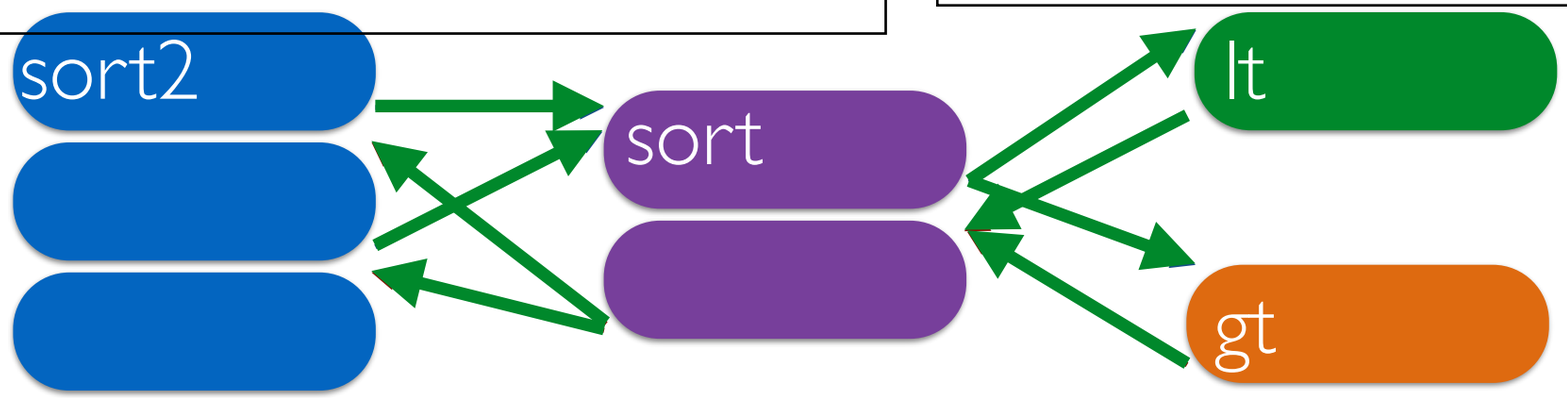


Which functions call other functions

Control Flow Graph

```
sort2(int a[], int b[], int
len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Break into ***basic blocks***

Distinguish ***calls*** *from* ***returns***

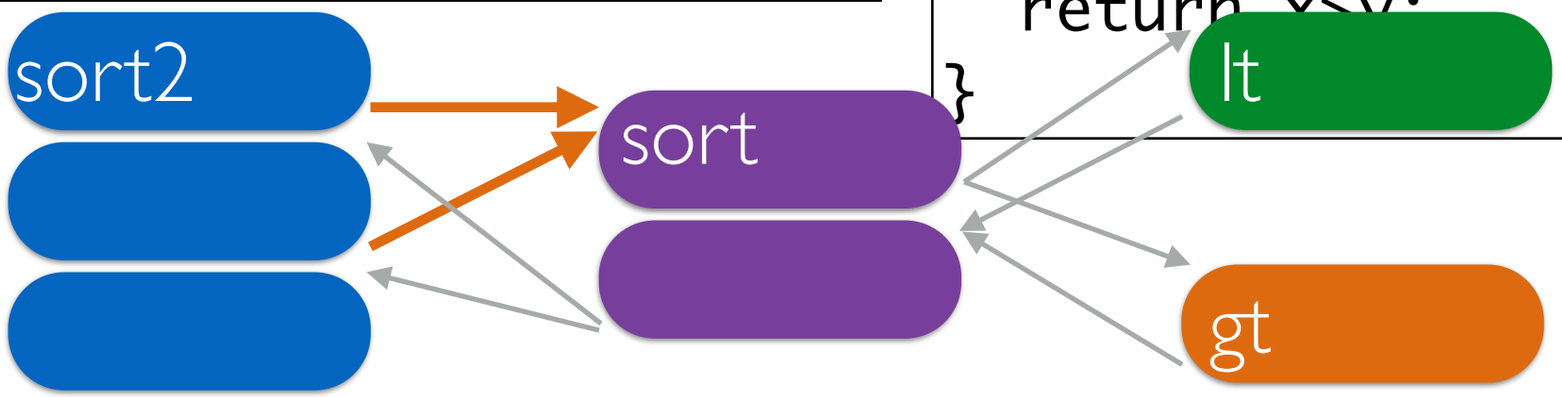
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
  - During compilation, or from the binary
- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG
- Observation: **Direct calls** need not be monitored
  - Assuming the code is immutable, the target address cannot be changed
- Therefore: **monitor only indirect calls**
  - `jmp`, `call`, `ret` with non-constant targets

Control Flow Graph

```
sort2(int a[], int b[], int
len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y)
{
  return x<y;
}
bool gt(int x, int y)
{
  return x>y;
}
```

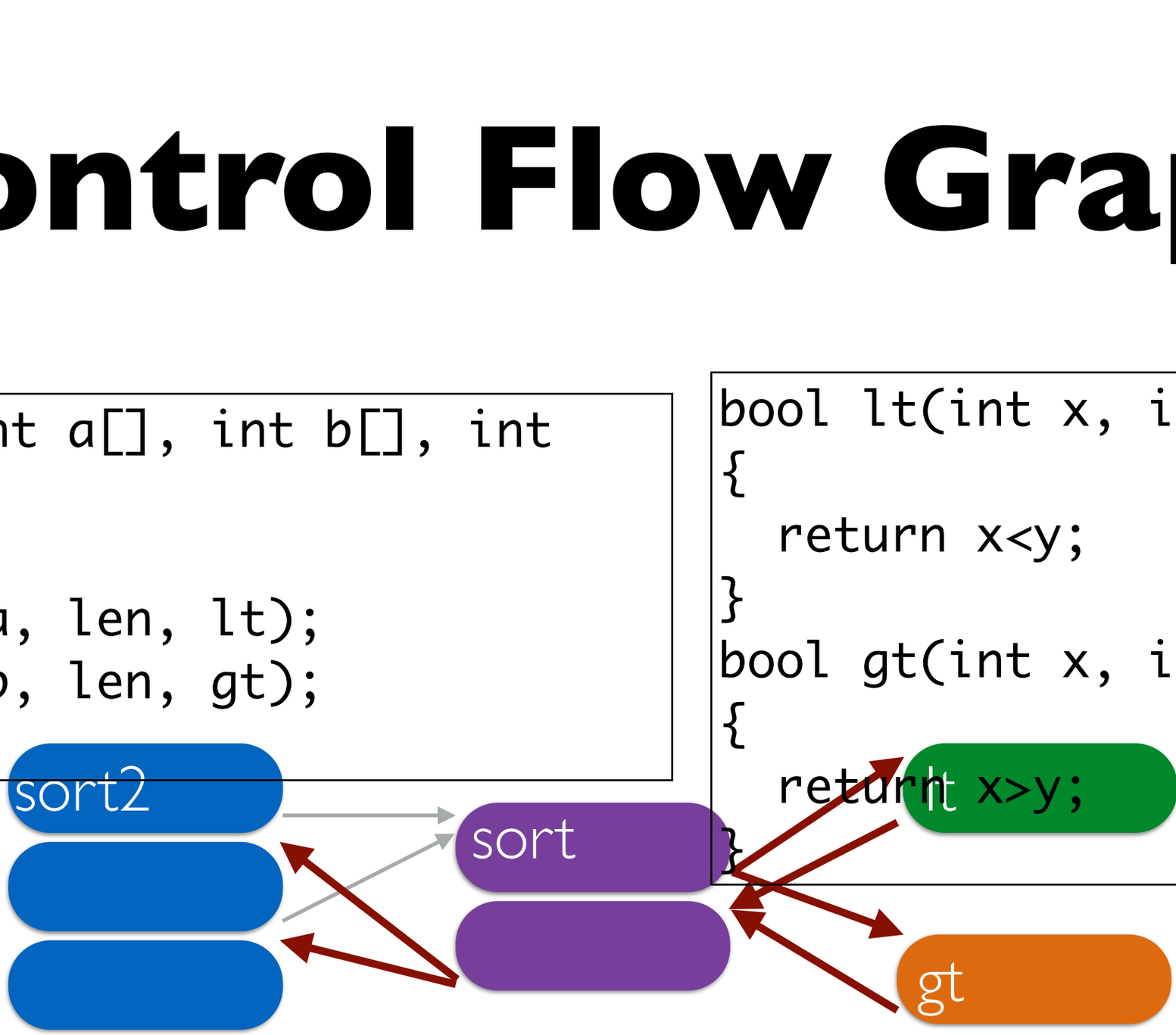


Direct calls *(always the same target)*

Control Flow Graph

```
sort2(int a[], int b[], int
len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y)
{
  return x<y;
}
bool gt(int x, int y)
{
  return x>y;
}
```



Indirect transfer (`call` *via register, or* `ret`)

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

In-line Monitor

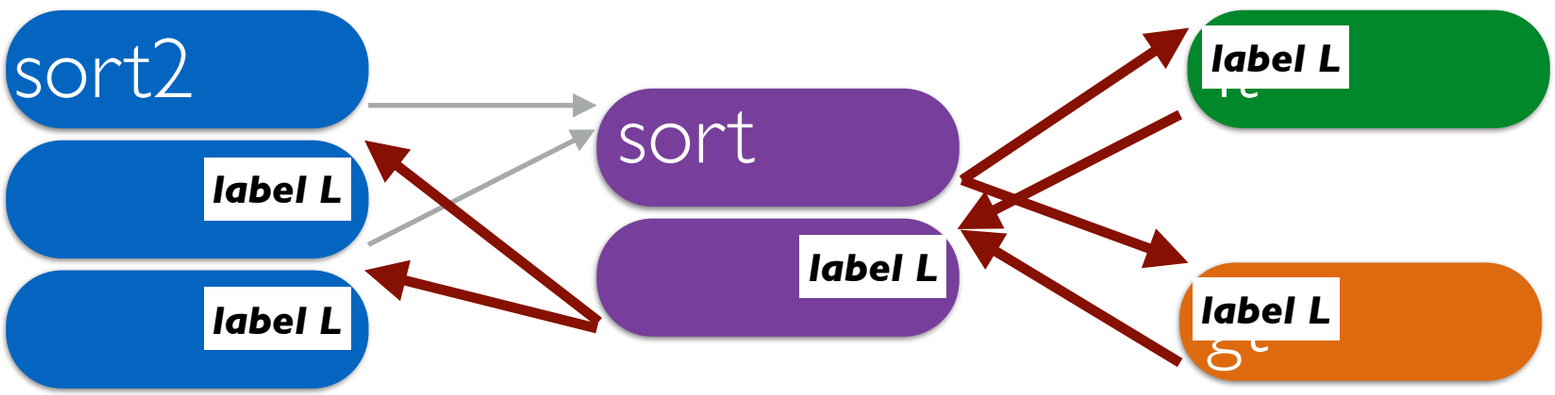
- Implement the monitor in-line, as a **program transformation**

- Insert a **label just before the target address** of an indirect transfer

- Insert **code to check the label of the target** at each indirect transfer
  - Abort if the label does not match

- The **labels are determined by the CFG**

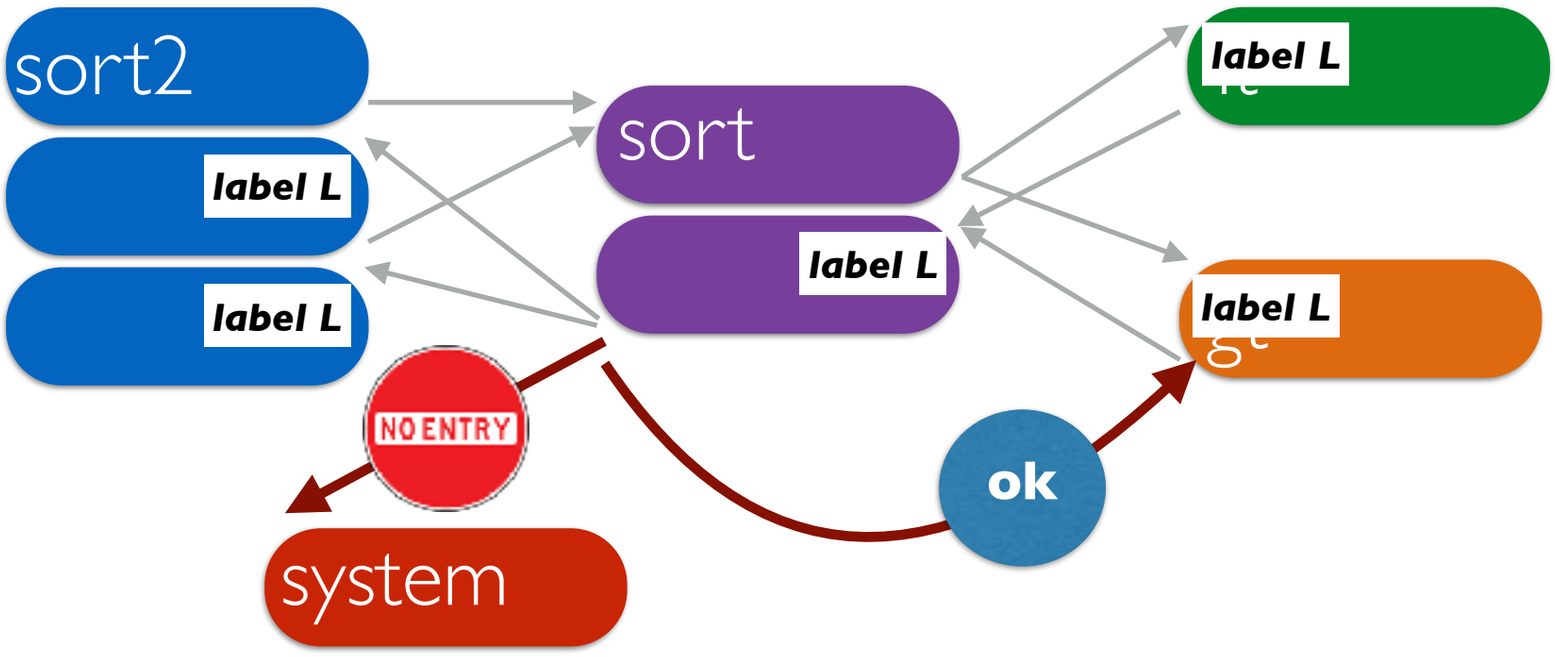
Simplest labeling



Use the same label at all targets: *label just means it's OK to jump here.*

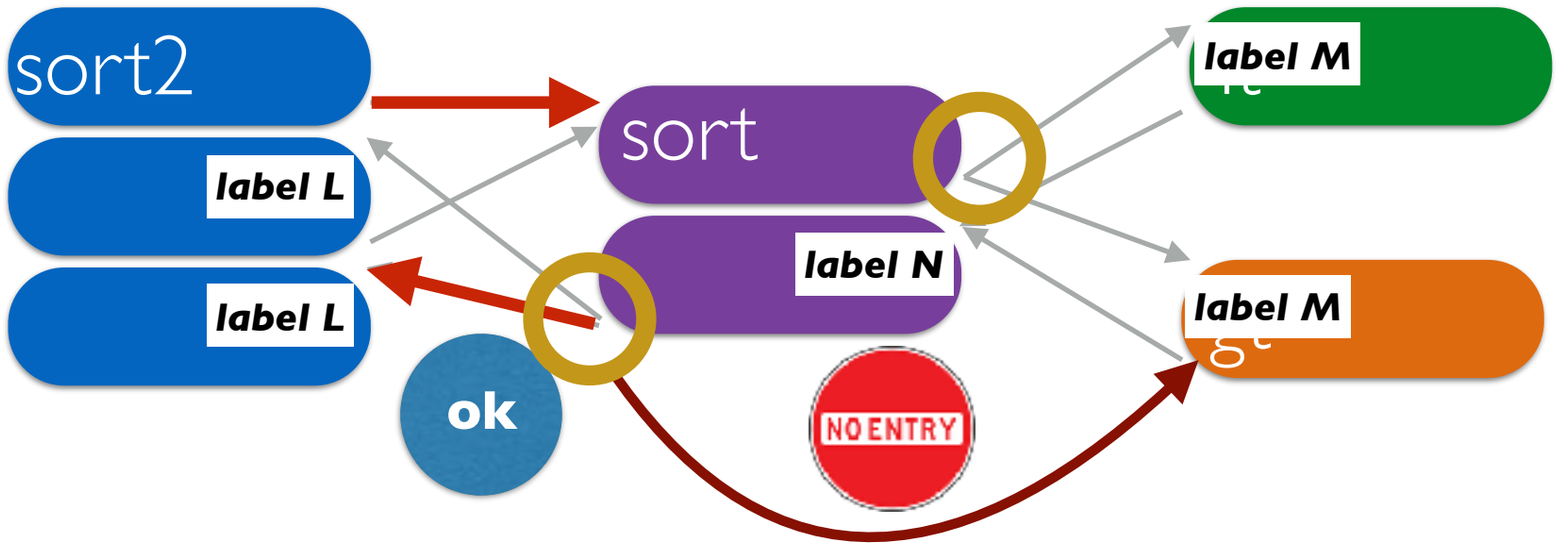
What could go wrong?

Simplest labeling



- Can't return to functions that aren't in the graph
- **Can** return to the right function in the wrong order

Detailed labeling



- All potential destinations of **same source** must match
  - Return sites from calls to `sort` must share a label (*L*)
  - Call targets `gt` and `lt` must share a label (*M*)
  - Remaining label unconstrained (*N*)

Prevents more abuse than simple labels,

but still permits call from site A to return to site B

Classic CFI instrumentation

Before CFI

```
FF 53 08                    call  [ebx+8]              ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

After CFI

```
8B 43 08                    mov   eax, [ebx+8]         ; load pointer into register
3E 81 78 04 78 56 34 12     cmp   [eax+4], 12345678h   ; compare opcodes at destination
75 13                       jne   error_label          ; if not ID value, then fail
FF D0                       call  eax                  ; call function pointer
3E 0F 18 05 DD CC BB AA     prefetchnta [AABBCCDDh]    ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 0C 24                    mov   ecx, [esp]           ; load address into register
83 C4 14                    add   esp, 14h             ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA     cmp   [ecx+4], AABBCCDDh   ; compare opcodes at destination
75 13                       jne   error_label          ; if not ID value, then fail
FF E1                       jmp   ecx                  ; jump to return address
```