CSE 127: Computer Security

Memory (un)safety

Deian Stefan

Some slides adopted from Nadia Heninger, Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, Raluca Popal, and David Wagner

Announcement

We'll try to release PAs on Wednesdays

- Due date is same
- Use Friday discussion to ask questions about it

Today

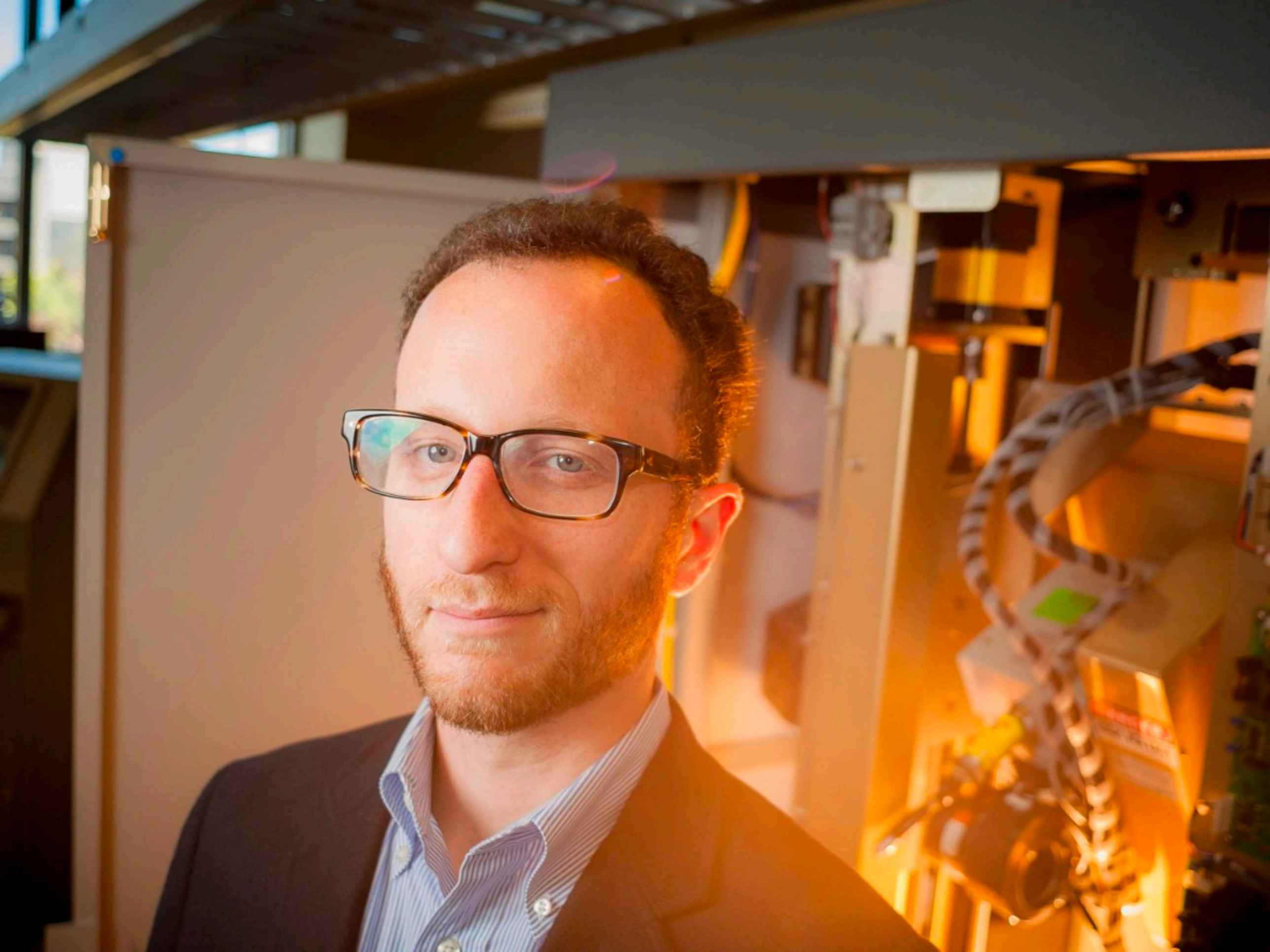
- Return oriented programming (ROP)
- Control flow integrity

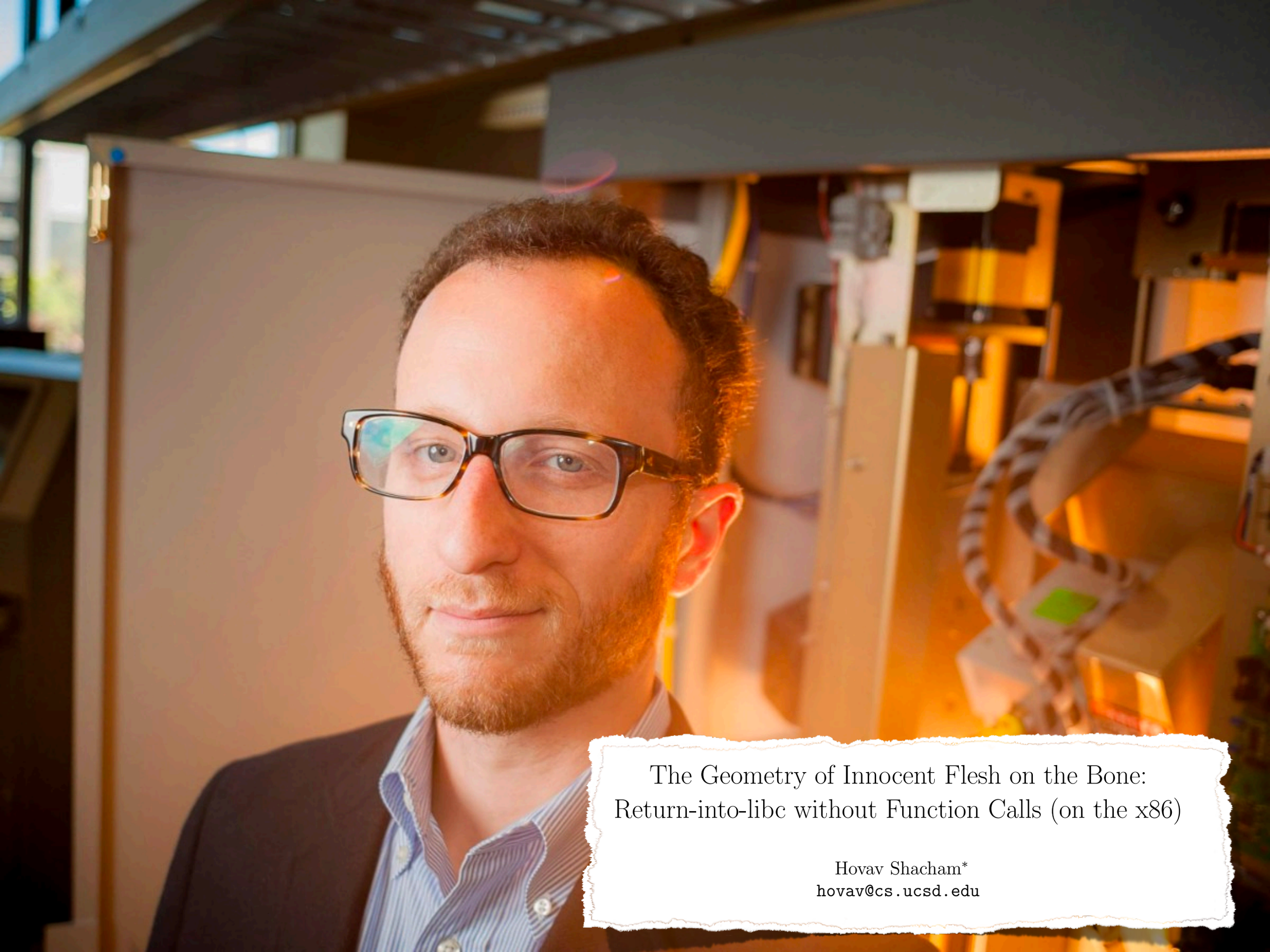
Last time: return-to-libc

- **Defense:** W^X makes the stack not executable
  - Prevents attacker data from being interpreted as code
- What can we do (as the attacker)?
  - Reuse existing code (either program or libc)
  - E.g., use `system("/bin/sh")`
  - E.g., use mprotect() to mark stack executable

Return-to-libc is great, but....

what if there is no function that does what we want?





The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

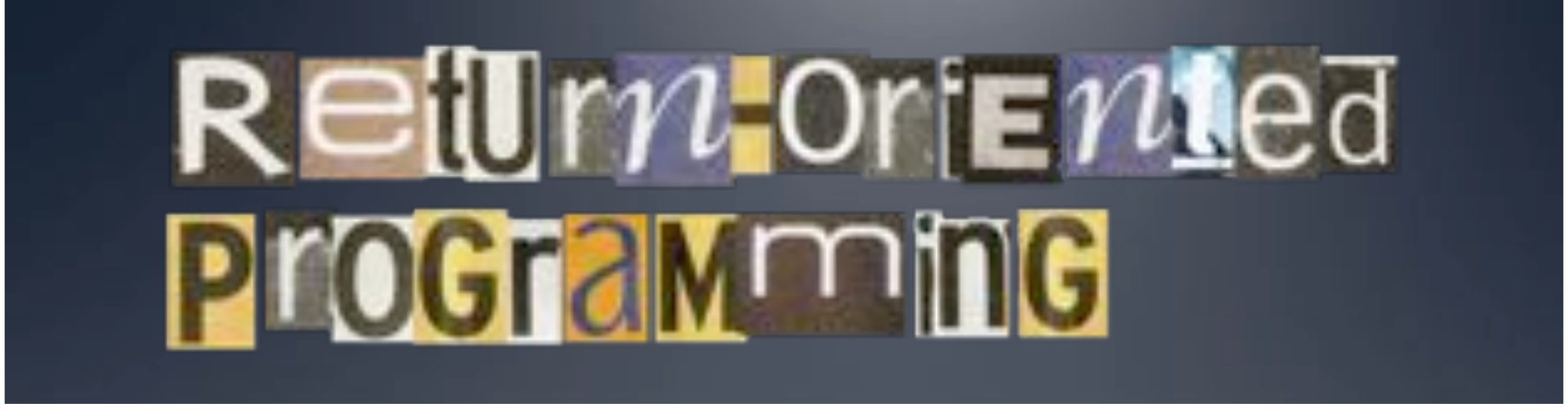
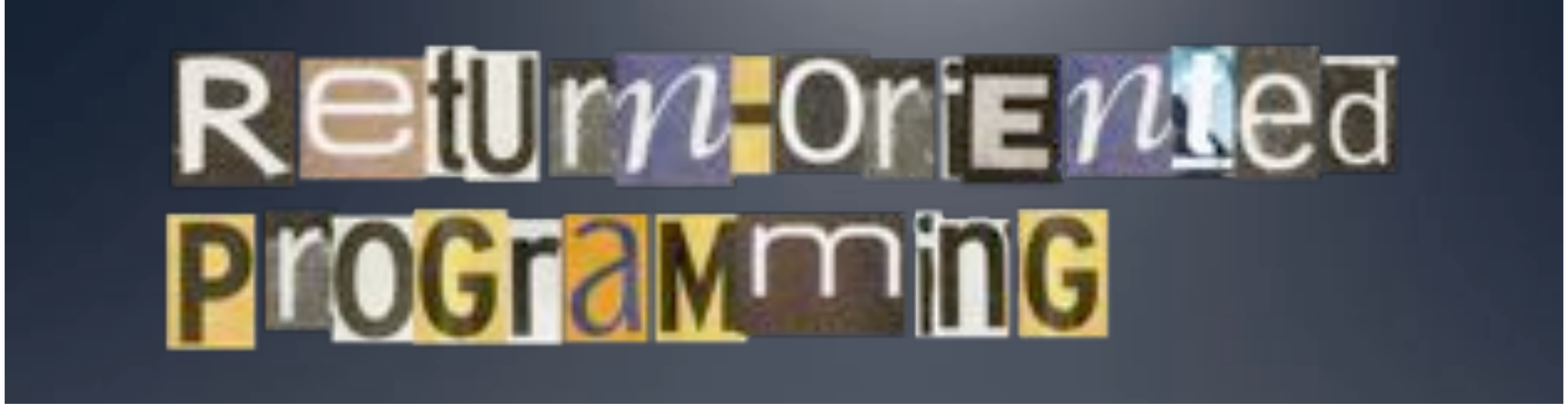
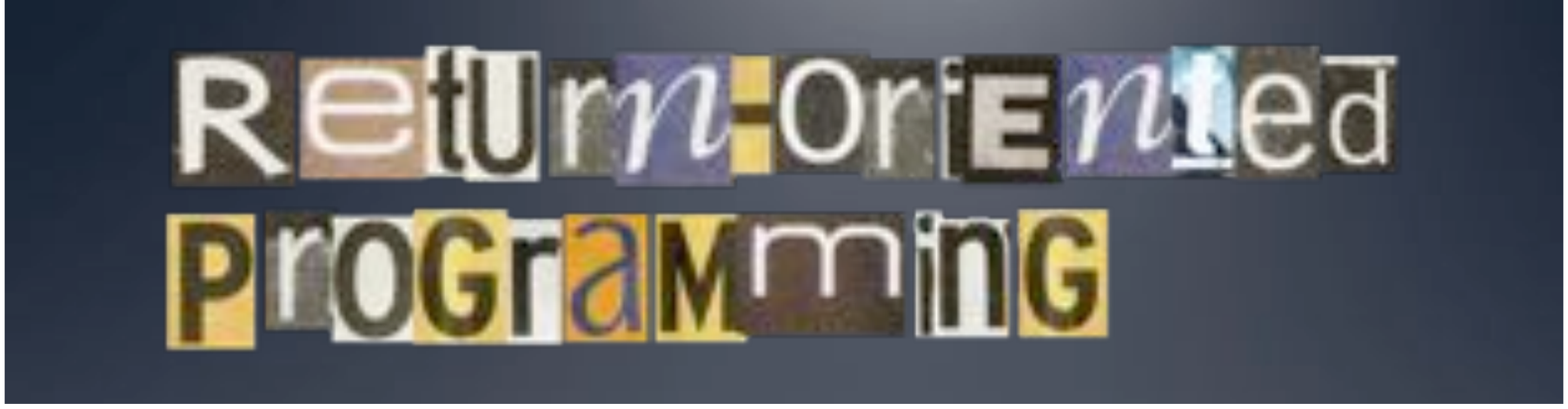


image by Dino Dai Zovi



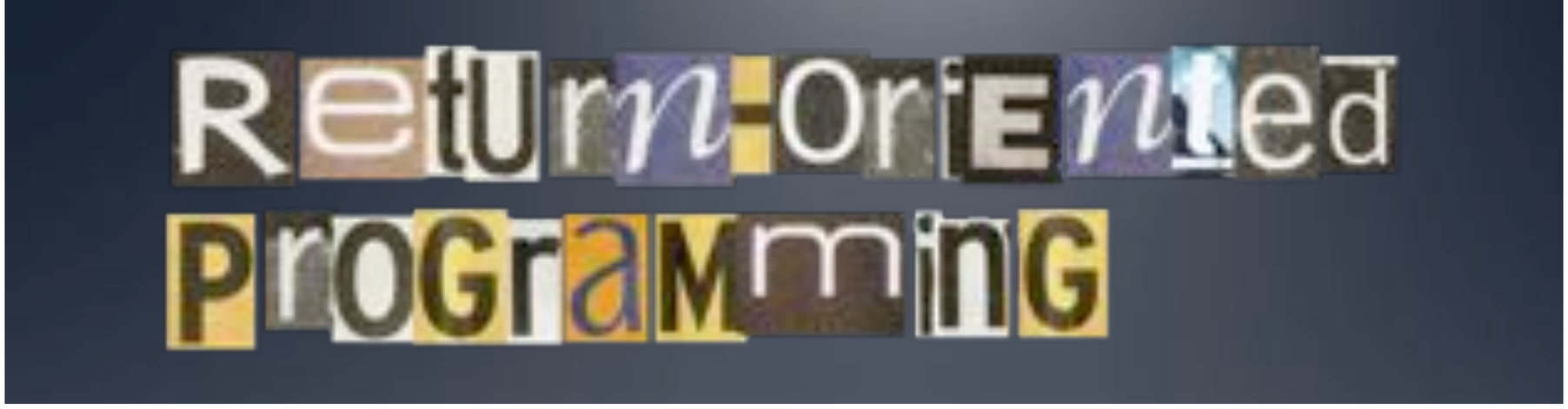
- Idea: make shellcode out of existing code gadgets
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to point to first gadget, then second gadget, etc.

image by Dino Dai Zovi



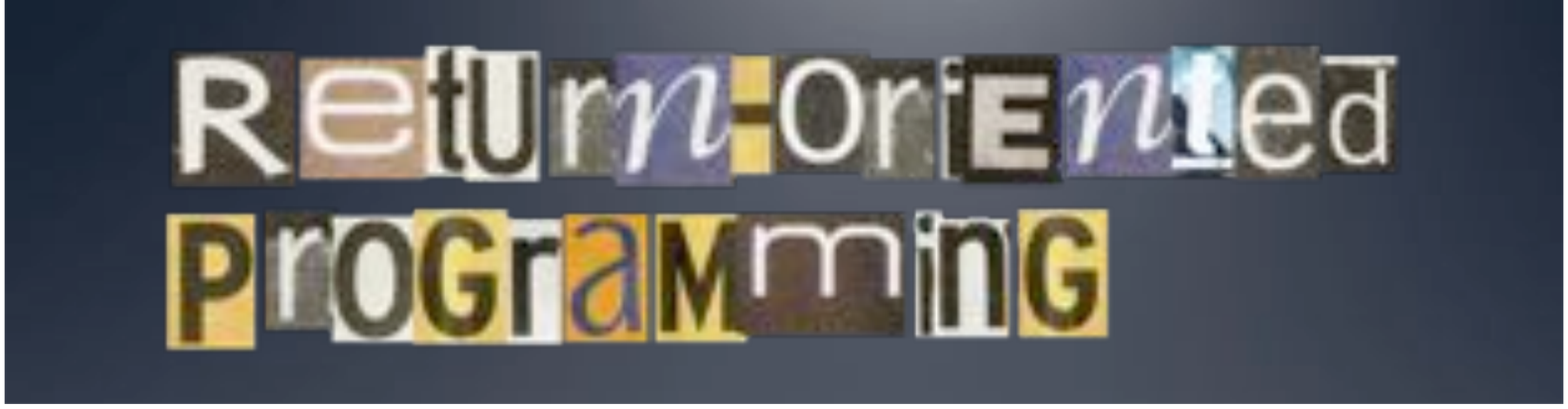
- Idea: make shellcode out of existing code gadgets
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to point to first gadget, then second gadget, etc.
- Where do you often find ret instructions?

image by Dino Dai Zovi



- Idea: make shellcode out of existing code gadgets
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to point to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function

image by Dino Dai Zovi



- Idea: make shellcode out of existing code gadgets
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to point to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function
  - Any sequence of executable memory ending in 0xc3

image by Dino Dai Zovi

```
$otool -t /bin/ls |grep c3
0000000100000f70        39 48 38 7f 07 b8 ff ff ff ff 7d 02 5d c3 48 83
0000000100000fc0        00 00 7d 02 5d c3 48 83 c6 68 49 83 c0 68 48 89
0000000100001010        c3 48 83 c7 68 48 83 c6 68 5d e9 6b 35 00 00 55
0000000100001050        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 49 83 c0
00000001000010a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 34
00000001000010e0        48 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001120        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 58 34
0000000100001150        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 48 83 c1
00000001000011a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 33
00000001000011e0        58 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001870        c0 09 c8 8a 0d ab 3c 00 00 89 c3 81 cb 80 00 00
0000000100001b70        5d d4 89 de e8 57 29 00 00 48 89 c3 48 85 db 0f
0000000100001c30        03 39 00 00 01 e9 52 01 00 00 0f b7 c0 83 f8 0d
0000000100001dd0        c3 48 8d 35 91 2d 00 00 eb 07 48 8d 35 c0 2d 00
0000000100001e20        36 0f b7 56 58 83 fa 07 75 02 5d c3 44 0f b7 c9
0000000100001ec0        00 48 8d 3d e2 2c 00 00 e8 21 26 00 00 48 89 c3
0000000100001f70        34 48 83 c3 02 80 f9 3a 75 19 80 7b fe 3a 75 13
0000000100001fa0        c3 84 c9 75 d0 44 89 b5 78 fb ff ff 45 89 e6 80
00000001000023b0        fb ff ff 74 5c 8b 78 74 e8 ef 20 00 00 48 89 c3
00000001000023e0        00 00 48 89 c3 48 85 db 0f 84 9a 04 00 00 48 89
0000000100002520        66 18 4d 8b 7e 20 41 8b 5e 30 48 63 c3 48 8d 34
0000000100002560        20 49 63 4e 30 41 89 04 8f 41 8b 5e 30 ff c3 41
0000000100002870        38 05 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 48
0000000100002970        c3 48 8d 3d 9e 22 00 00 48 8d 35 a1 22 00 00 48
0000000100002a30        ed 48 83 c3 68 48 89 df e8 4f 0b 00 00 89 c3 45
0000000100002a90        0f b7 7c 24 04 e8 28 0b 00 00 01 c3 89 d8 48 83
0000000100002aa0        c4 08 5b 41 5c 41 5d 41 5e 41 5f 5d c3 55 48 89
0000000100002c30        4f 28 48 8b 46 08 eb 0f 85 c0 45 8b 4f 38 48 8b
0000000100003200        00 48 83 c3 18 48 81 fb a8 01 00 00 75 84 bb 10
0000000100003260        45 89 fd 4c 8d bd b0 f7 ff ff 48 83 c3 18 48 83
00000001000032e0        5b 41 5c 41 5d 41 5e 41 5f 5d c3 48 8d 35 c5 18
0000000100003350        48 83 c4 08 5b 5d c3 48 8d 3d c6 1b 00 00 31 c0
00000001000034a0        c4 70 5b 41 5e 5d c3 e8 a0 0f 00 00 55 48 89 e5
0000000100003550        00 89 d8 48 83 c4 08 5b 5d c3 66 90 7e ff ff ff
00000001000035f0        00 00 00 5d c3 81 c1 00 60 00 00 81 e1 00 f0 00
00000001000036a0        75 06 48 83 c3 10 eb 69 48 8d 7b 68 e8 f7 0e 00
0000000100003740        5e 41 5f 5d c3 55 48 89 e5 41 57 41 56 41 55 41
0000000100003930        5c f0 ff ff 89 c3 48 8d 05 1b 1d 00 00 8b 08 85
0000000100003970        7c 04 85 c9 75 40 41 89 d4 89 c3 48 8d 05 b6 1c
0000000100003990        45 f8 e8 c3 0b 00 00 42 8d 04 2b 23 45 c8 44 89
00000001000039f0        83 c4 38 5b 41 5c 41 5d 41 5e 41 5f 5d c3 31 ff
0000000100003ac0        00 00 48 89 c3 8a 04 1a 88 45 d6 48 83 ca 01 48
0000000100003b00        f8 80 f9 30 75 36 83 c3 d0 41 89 1f 66 bb 01 00
0000000100003b40        9f 80 f9 07 77 08 83 c3 9f 41 89 1f eb 4e 89 c1
0000000100003b50        80 c1 bf 80 f9 07 77 12 83 c3 bf 41 89 1f 48 8b
0000000100003bd0        41 5d 41 5e 41 5f 5d c3 55 48 89 e5 41 56 53 41
0000000100003c30        c6 08 00 00 89 c7 44 89 f6 5b 41 5e 5d e9 e2 08
0000000100003c60        31 c0 48 83 c4 10 5d c3 55 48 89 e5 e8 e9 08 00
0000000100003c70        00 31 c0 5d c3 55 48 89 e5 41 56 53 89 f8 48 8d
0000000100003d10        5e 5d e9 b5 08 00 00 5b 41 5e 5d c3 55 48 89 e5
0000000100003e40        ff ff 4c 89 e6 4c 89 f9 e8 f5 06 00 00 48 89 c3
0000000100003e90        98 00 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 e8
```

x86 instructions

- Variable length!
- Can begin on any byte boundary!

One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =   mov $0x1,%eax
                              pop %ebx
                              leave
                              ret
```

One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =   add %al,(%eax)
                              pop %ebx
                              leave
                              ret
```

One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =   add %bl,-0x37(%eax)
                              ret
```

One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =   pop %ebx
                              leave
                              ret
```

One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =    leave
                               ret
```

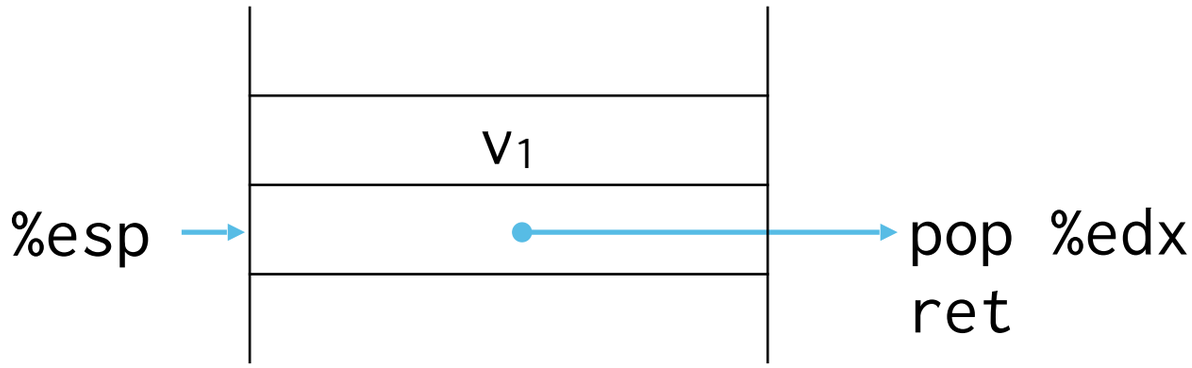
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3   =   ret
```

Why is `ret`?

- Attacker overflows stack allocated buffer
- What happens when function returns?
  - Restore stack frame
    - `leave = movl %ebp, %esp; pop %ebp`
  - Return
    - `ret = pop %eip`
- If instruction sequence at `%eip` ends in `ret` what do we do?

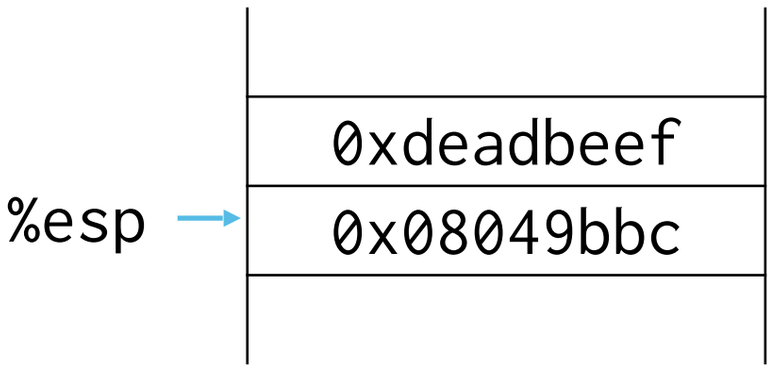
What happens if this is what we overflow the stack with?



relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



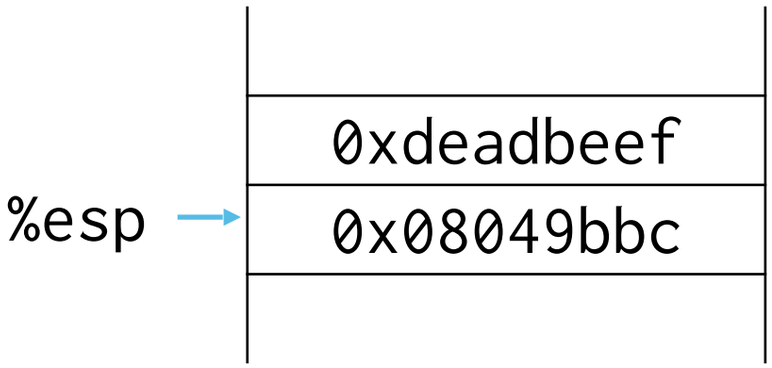
relevant code:

```
%eip → 0x08049b62: nop
       0x08049b63: ret
                   ...

       0x08049bbc: pop %edx
       0x08049bbd: ret
```

relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



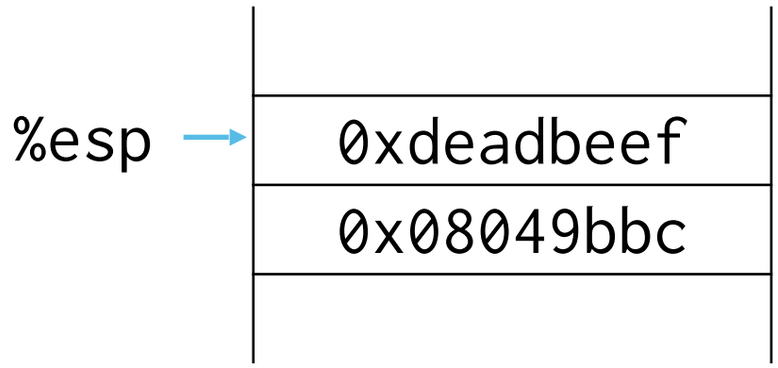
relevant code:

```
          0x08049b62: nop
%eip →    0x08049b63: ret
                  ...

          0x08049bbc: pop %edx
          0x08049bbd: ret
```

relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



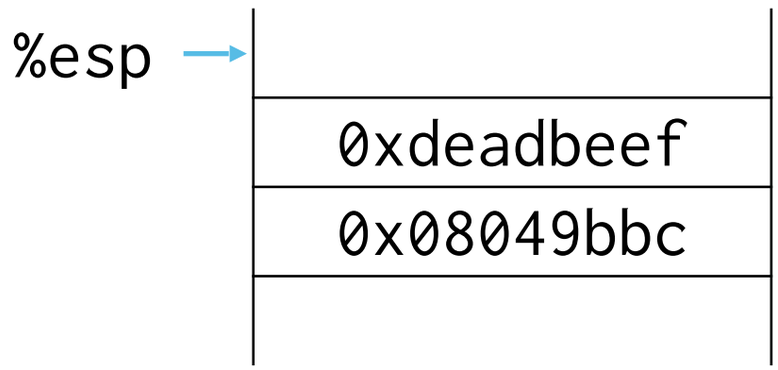
relevant code:

```
          0x08049b62: nop
          0x08049b63: ret
                      ...
%eip →    0x08049bbc: pop %edx
          0x08049bbd: ret
```

relevant register(s):

```
%edx = 0xdeadbeef
```

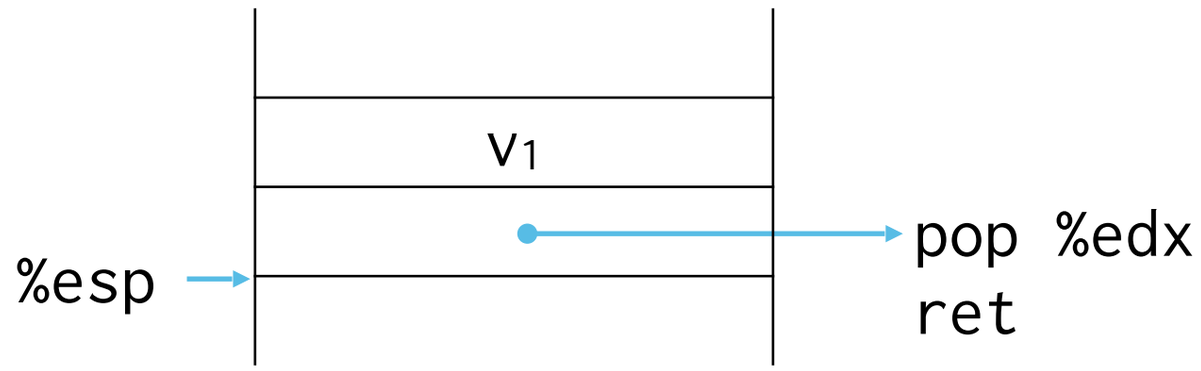
relevant stack:



relevant code:

```
            0x08049b62: nop
            0x08049b63: ret
                   ...

            0x08049bbc: pop %edx
%eip →      0x08049bbd: ret
```

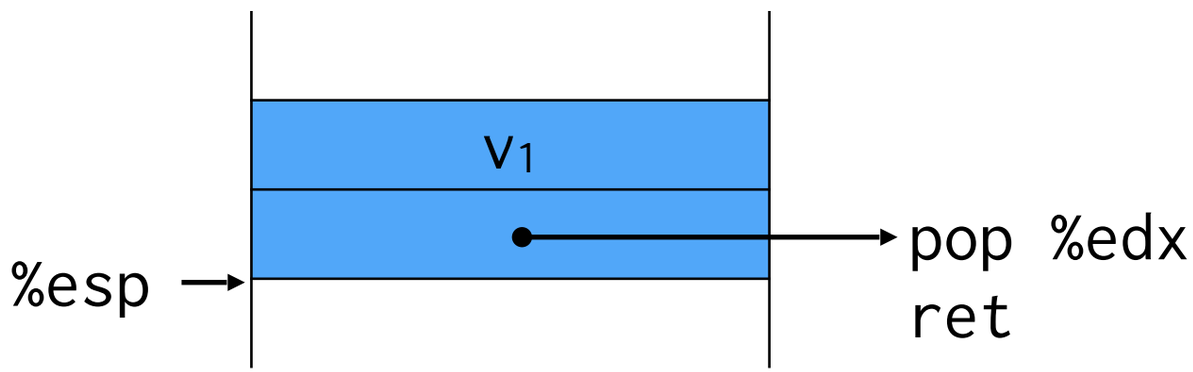
This is a ROP gadget!



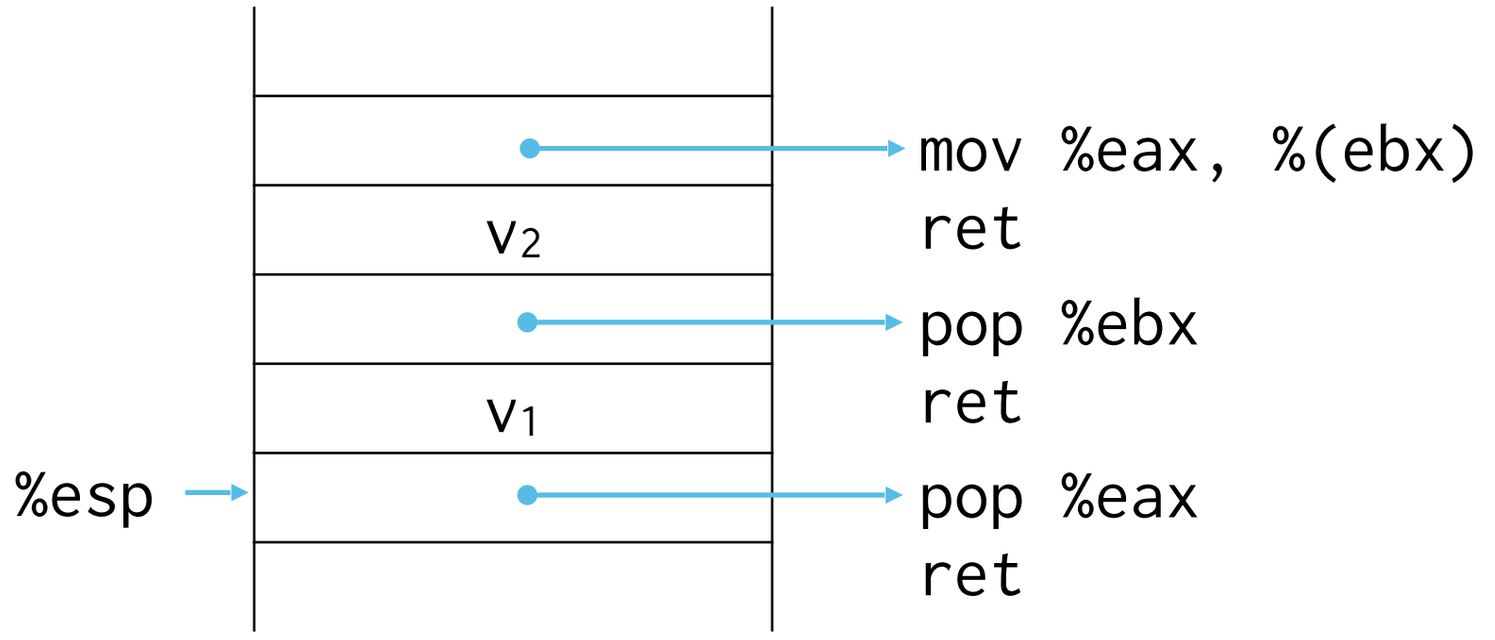
movl v_1, %edx

How dow you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program
- E.g., if shellcode needs to write a value to %edx, use the previous gadget



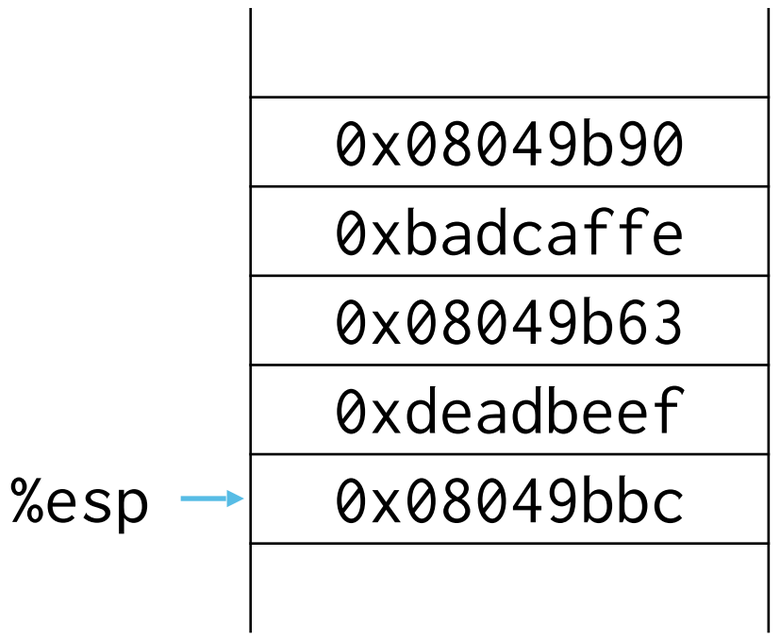
Let’s look at another gadget



relevant register(s):

```
%eax = 0x00000000
%ebx = 0x00000000
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

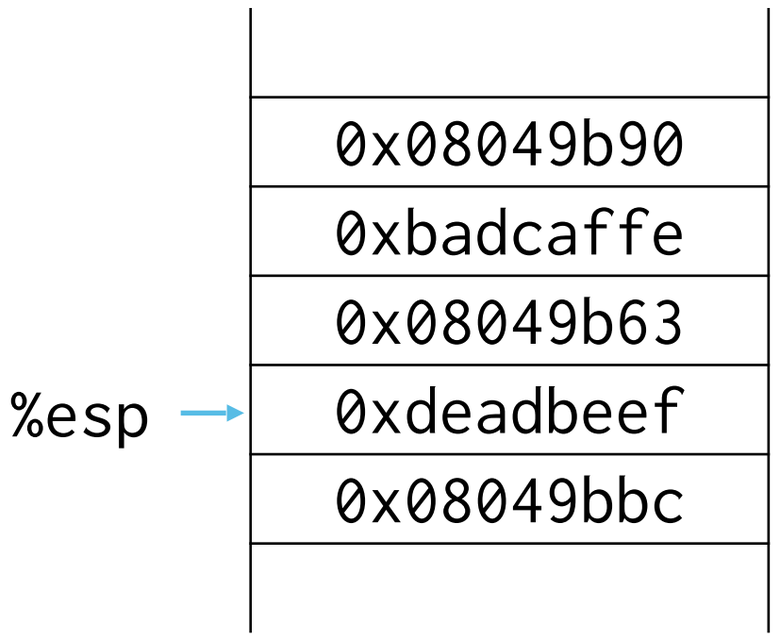
relevant code:

```
%eip → 0x08049b00: ret
       ...
       0x08049b63: pop %ebx
       0x08049b64: ret
       ...
       0x08049b90: mov %eax, %(ebx)
       0x08049b91: ret
       ...
       0x08049bbc: pop %eax
       0x08049bbd: ret
```

relevant register(s):

```
%eax = 0x00000000
%ebx = 0x00000000
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

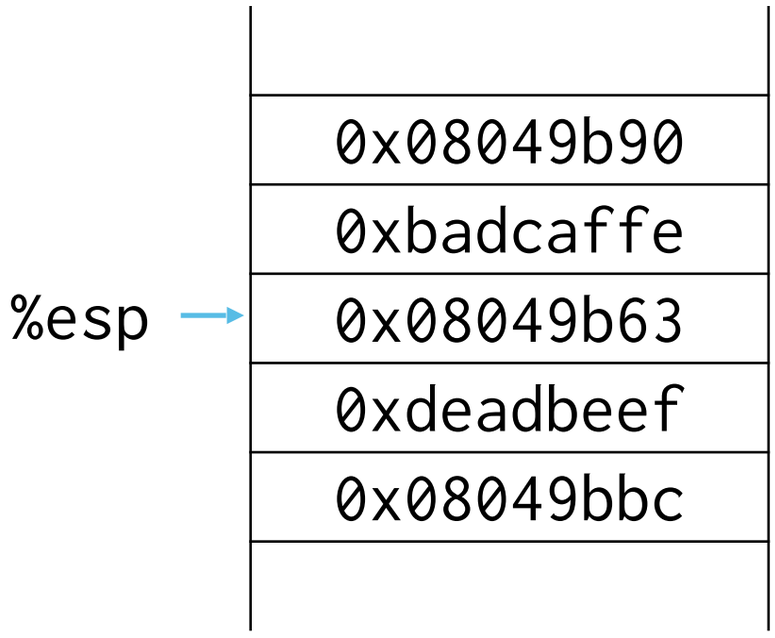
relevant code:

```
        0x08049b00: ret
        ...
        0x08049b63: pop %ebx
        0x08049b64: ret
        ...
        0x08049b90: mov %eax, %(ebx)
        0x08049b91: ret
        ...
%eip →  0x08049bbc: pop %eax
        0x08049bbd: ret
```

relevant register(s):

```
%eax = 0xdeadbeef
%ebx = 0x00000000
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

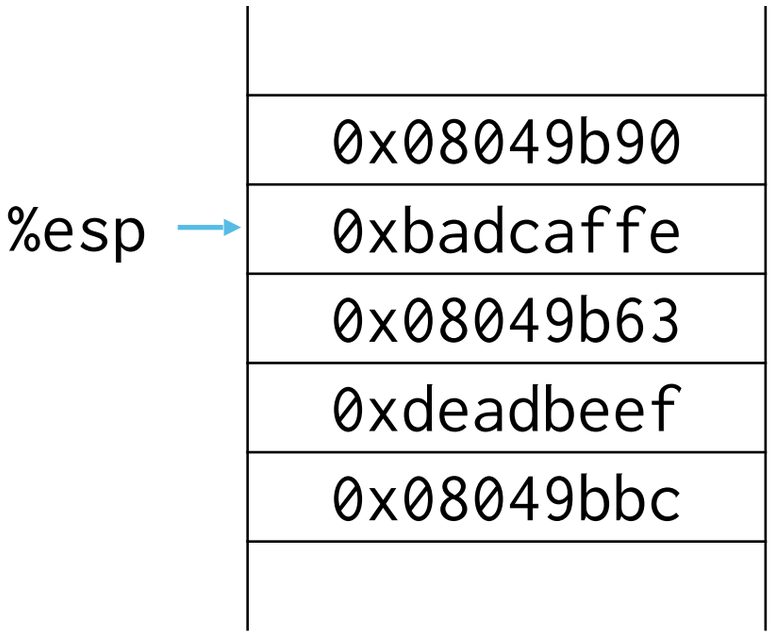
relevant code:

```
    0x08049b00: ret
    ...
    0x08049b63: pop %ebx
    0x08049b64: ret
    ...
    0x08049b90: mov %eax, %(ebx)
    0x08049b91: ret
    ...
    0x08049bbc: pop %eax
%eip → 0x08049bbd: ret
```

relevant register(s):

```
%eax = 0xdeadbeef
%ebx = 0x00000000
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

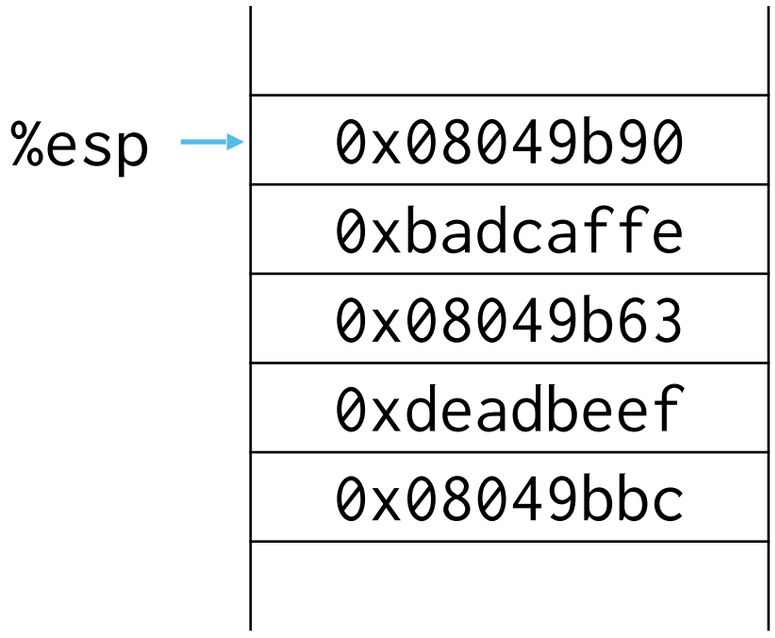
relevant code:

```
        0x08049b00: ret
        ...
%eip →  0x08049b63: pop %ebx
        0x08049b64: ret
        ...
        0x08049b90: mov %eax, %(ebx)
        0x08049b91: ret
        ...
        0x08049bbc: pop %eax
        0x08049bbd: ret
```

relevant register(s):

```
%eax = 0xdeadbeef
%ebx = 0xbadcaffe
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

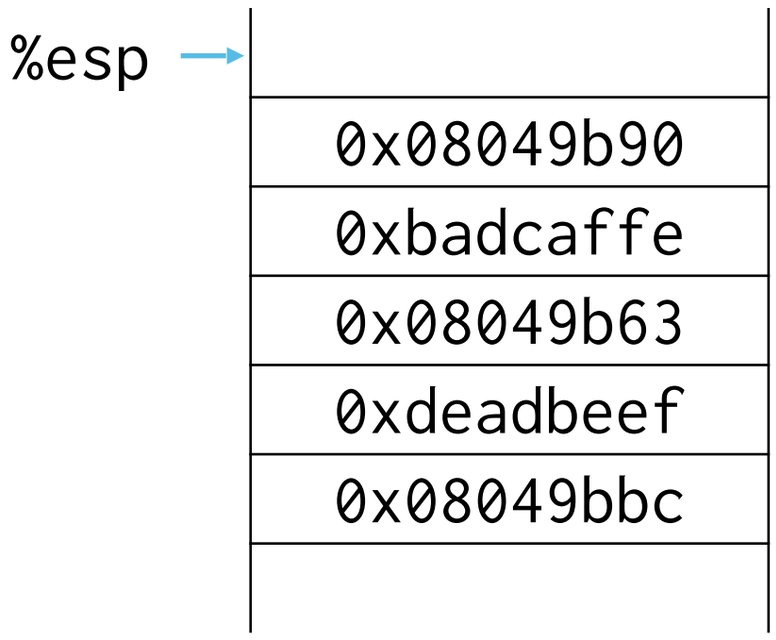
relevant code:

```
        0x08049b00: ret
        ...
        0x08049b63: pop %ebx
%eip →  0x08049b64: ret
        ...
        0x08049b90: mov %eax, %(ebx)
        0x08049b91: ret
        ...
        0x08049bbc: pop %eax
        0x08049bbd: ret
```

relevant register(s):

```
%eax = 0xdeadbeef
%ebx = 0xbadcaffe
```

relevant stack:



relevant memory:

```
0xbadcaffe: 0x00000000
```

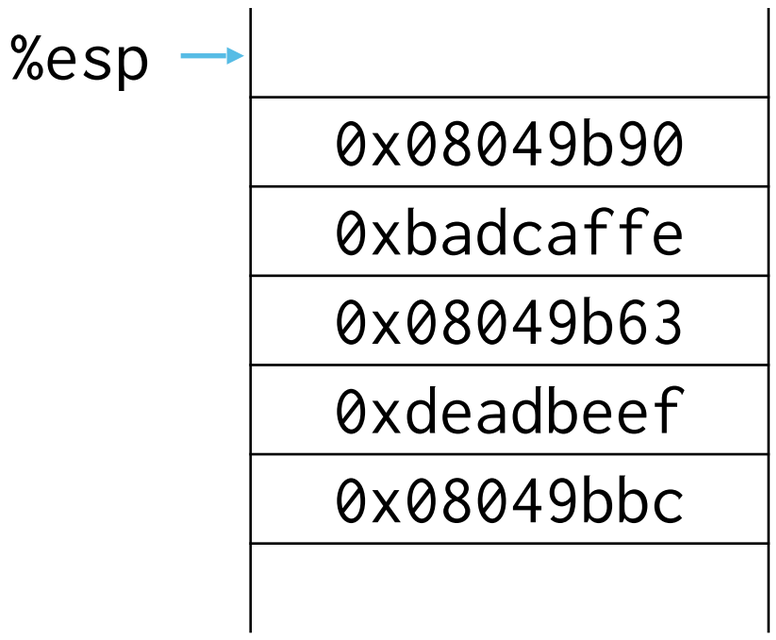
relevant code:

```
        0x08049b00: ret
        ...
        0x08049b63: pop %ebx
        0x08049b64: ret
        ...
%eip →  0x08049b90: mov %eax, %(ebx)
        0x08049b91: ret
        ...
        0x08049bbc: pop %eax
        0x08049bbd: ret
```

relevant register(s):

```
%eax = 0xdeadbeef
%ebx = 0xbadcaffe
```

relevant stack:



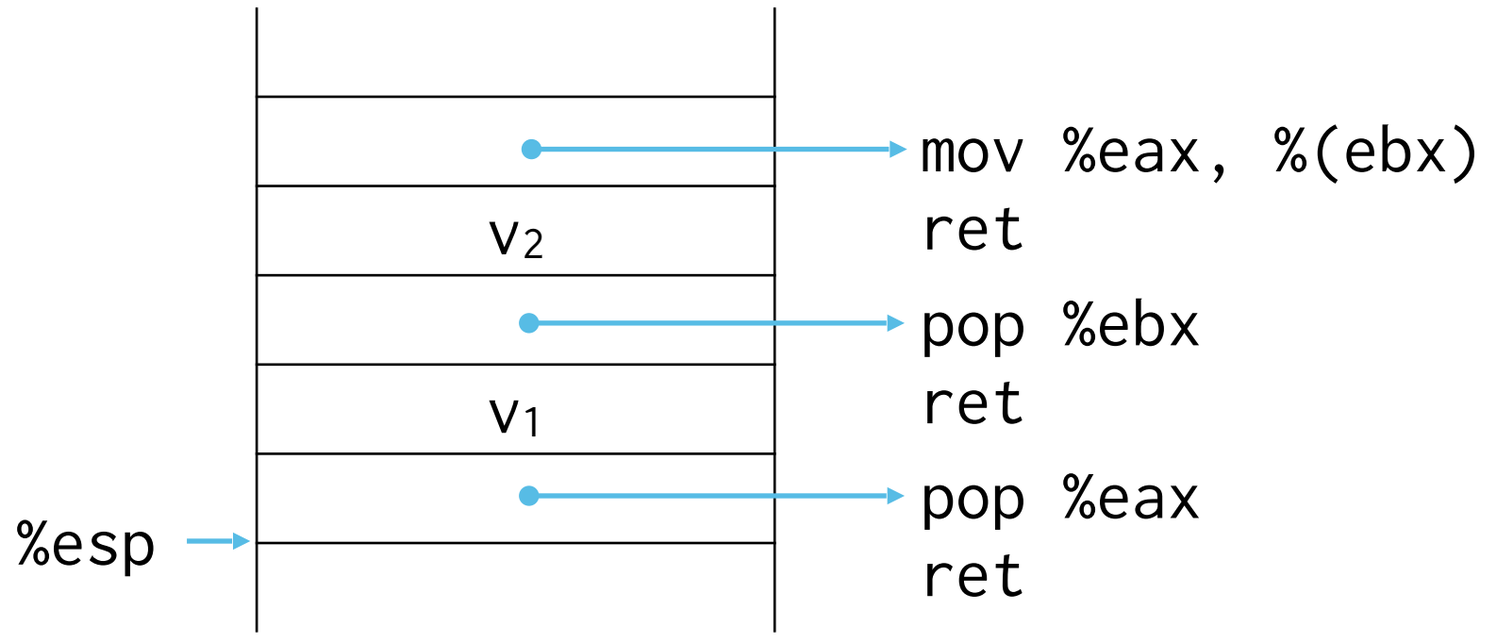
relevant memory:

```
0xbadcaffe: 0xdeadbeef
```

relevant code:

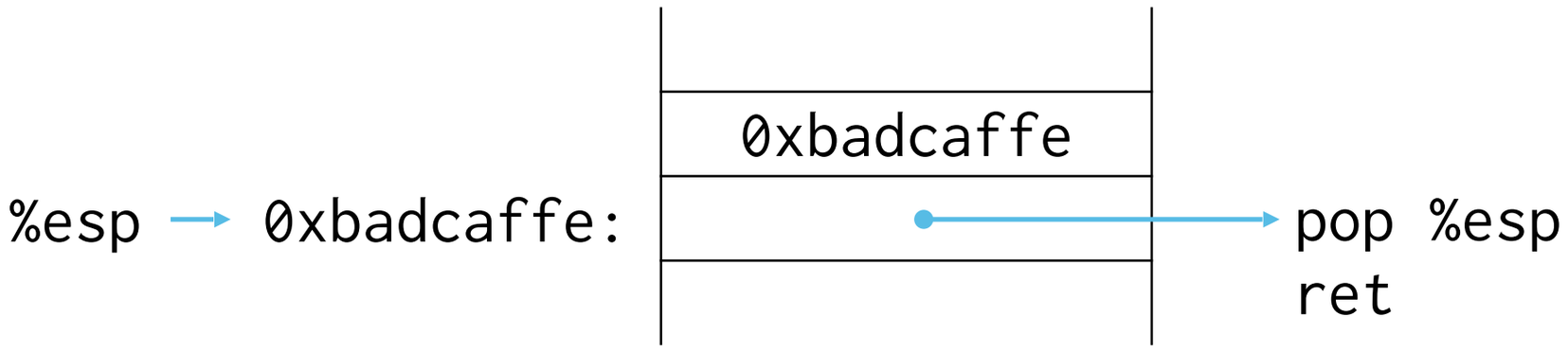
```
        0x08049b00: ret
        ...
        0x08049b63: pop %ebx
        0x08049b64: ret
        ...
        0x08049b90: mov %eax, %(ebx)
%eip →  0x08049b91: ret
        ...
        0x08049bbc: pop %eax
        0x08049bbd: ret
```

What does this gadget do?




```
movl v2, %ebx
movl v1, %(%ebx)
```

What does this gadget do?



Can express arbitrary programs

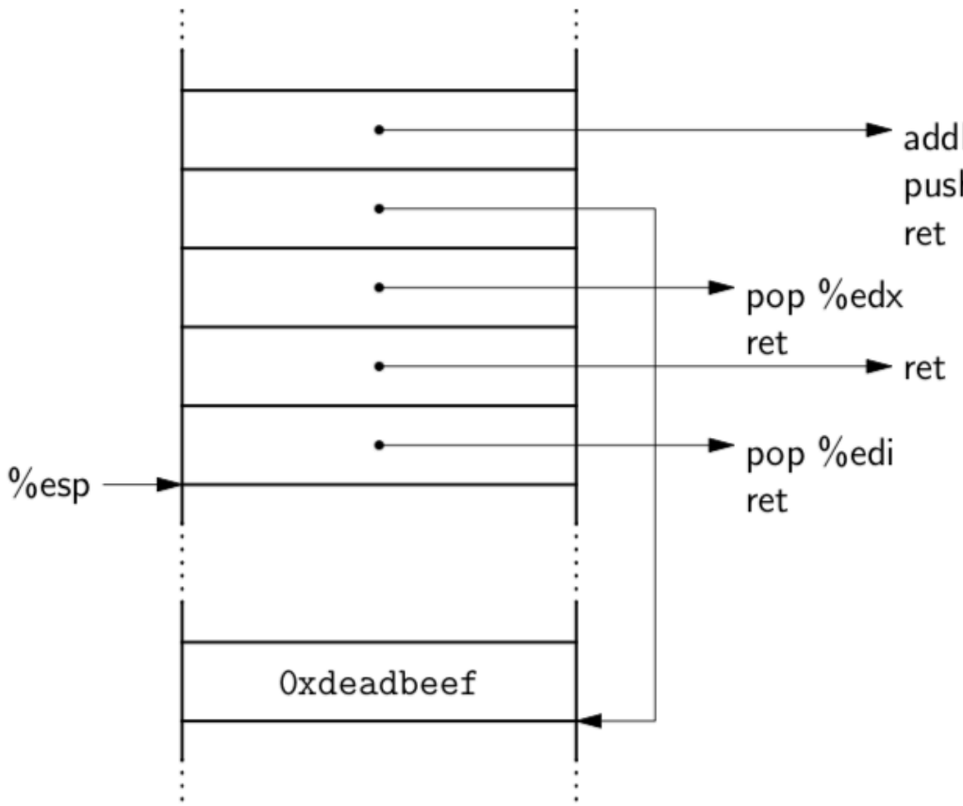


Figure 5: Simple add into %eax.

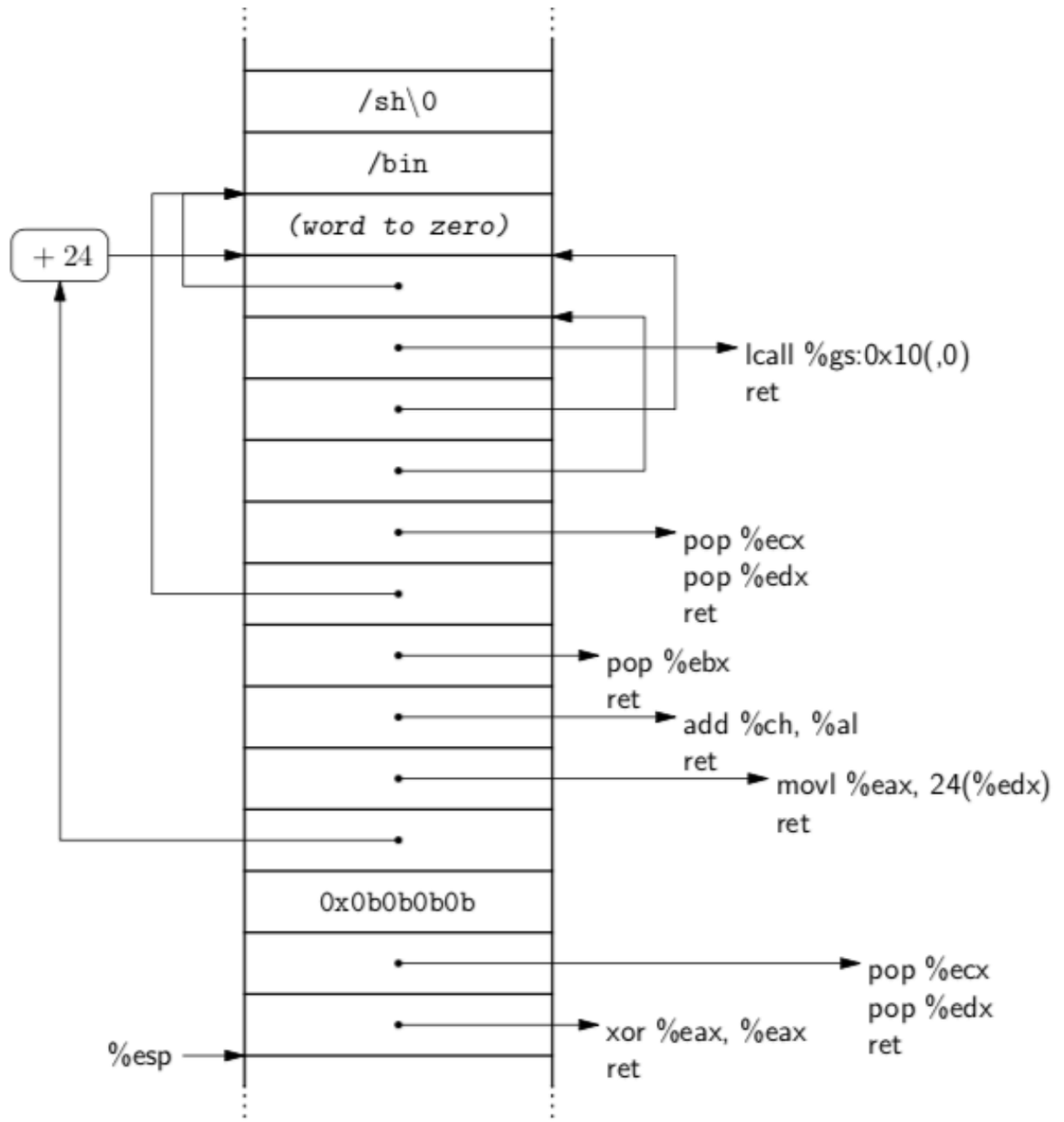


Figure 16: Shellcode.

Can find gadgets automatically

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode)/ ARM64
- PowerPC / PowerPC64

Return-oriented programming

not even really about “returns”…

Rethinking control flow hijacking

Observation: In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses and function pointers)

Control Flow Integrity

Don’t try to stop the memory writes.

Instead: restrict control flow to legitimate paths

- Ensure that jumps, calls, and returns can only go to allowed target destinations

Restrict indirect transfers of control

Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?

Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?
  - Address is hard-coded in instruction. Not under attacker control

Restrict indirect transfers of control

Restrict indirect transfers of control

- What are the ways to transfer control indirectly?

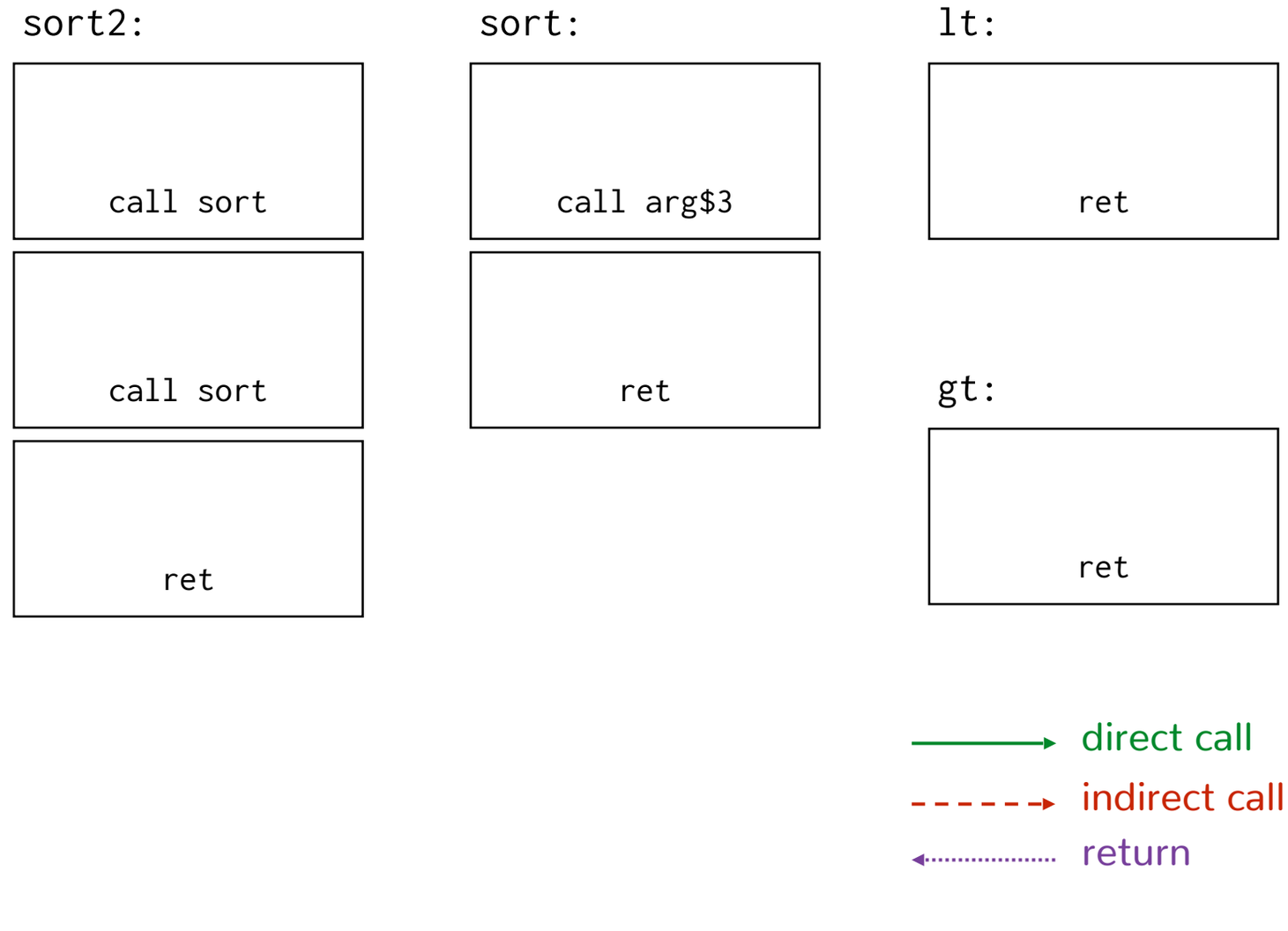
Restrict indirect transfers of control

- What are the ways to transfer control indirectly?
- Forward path: jumping to (or calling function at) an address in register or memory
  - E.g., qsort, interrupt handlers, virtual calls, etc.
- Reverse path: returning from function (uses address on stack)

What's a legitimate target?

Look at the program control-flow graph (CFG)!

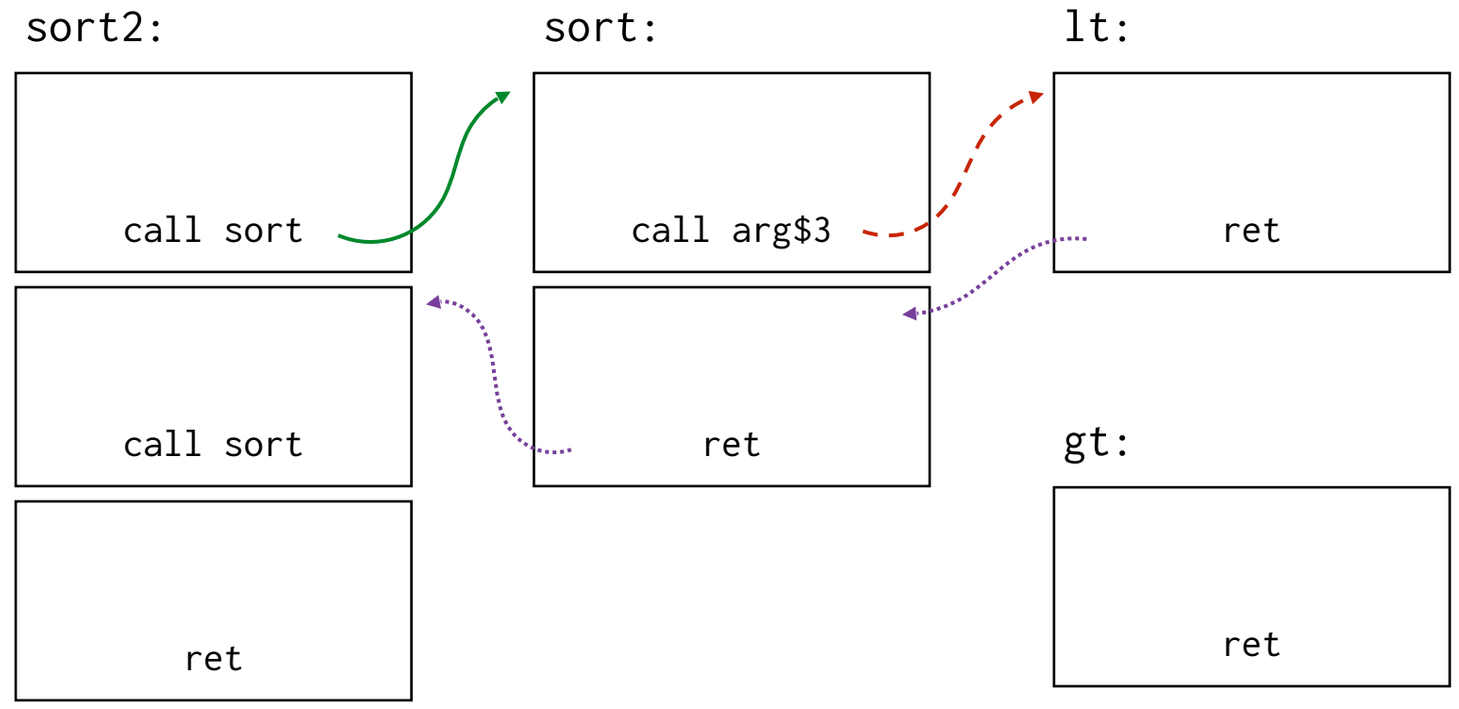
```
void sort2(int* a,int* b, int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

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

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



What's a legitimate target?

Look at the program control-flow graph (CFG)!

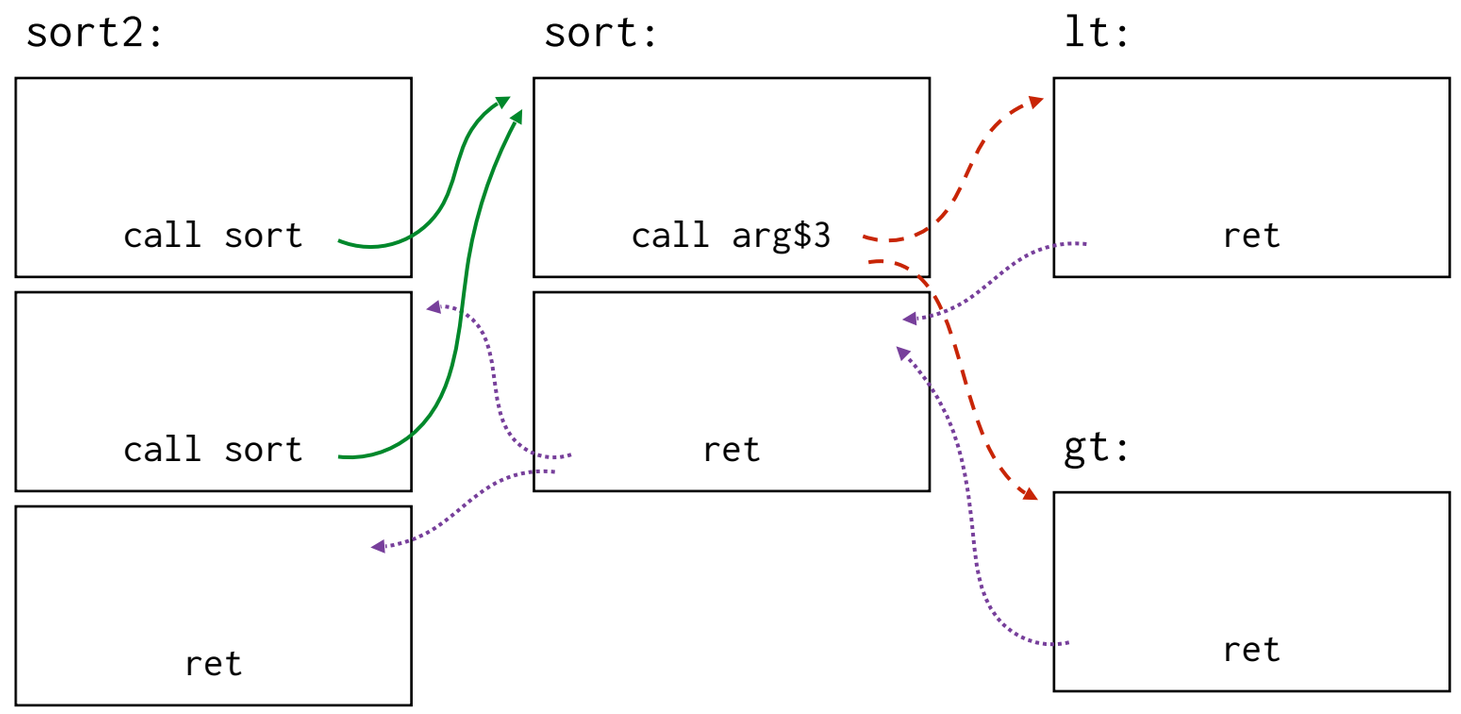
```
void sort2(int* a,int* b, int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

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

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



What's a legitimate target?

Look at the program control-flow graph (CFG)!

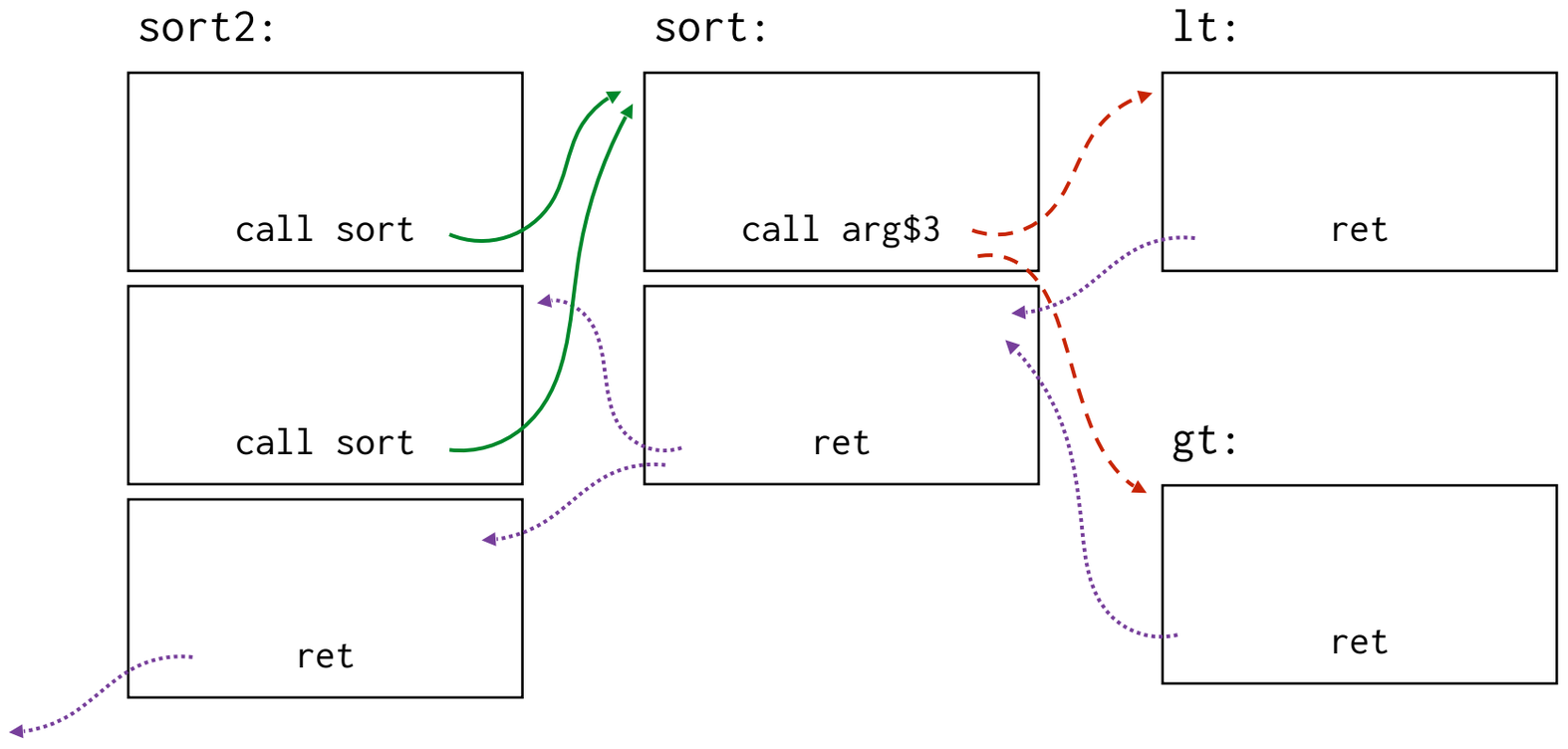
```
void sort2(int* a,int* b, int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

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

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



What's a legitimate target?

Look at the program control-flow graph (CFG)!

```
void 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;
}
```



How do we restrict jumps to CFG?

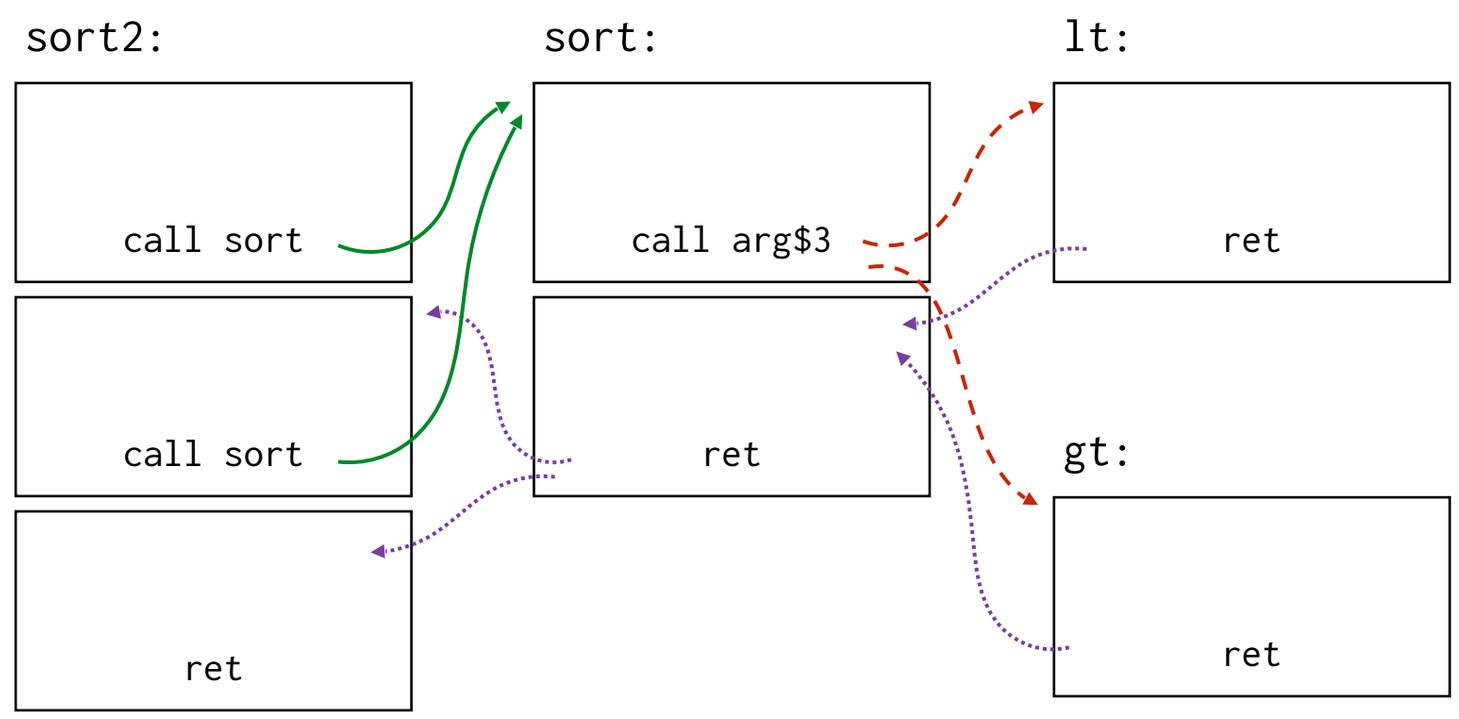
- Assign labels to all indirect jumps and their targets
- Before taking an indirect jump, validate that target label matches jump site
  - Like stack canaries, but for for control flow target

Fine grained CFI (Abadi et al.)

- Statically compute CFG
- Dynamically ensure program never deviates
  - Assign label to each target of indirect transfer
  - Instrument indirect transfers to compare label of destination with the expected label to ensure it's valid

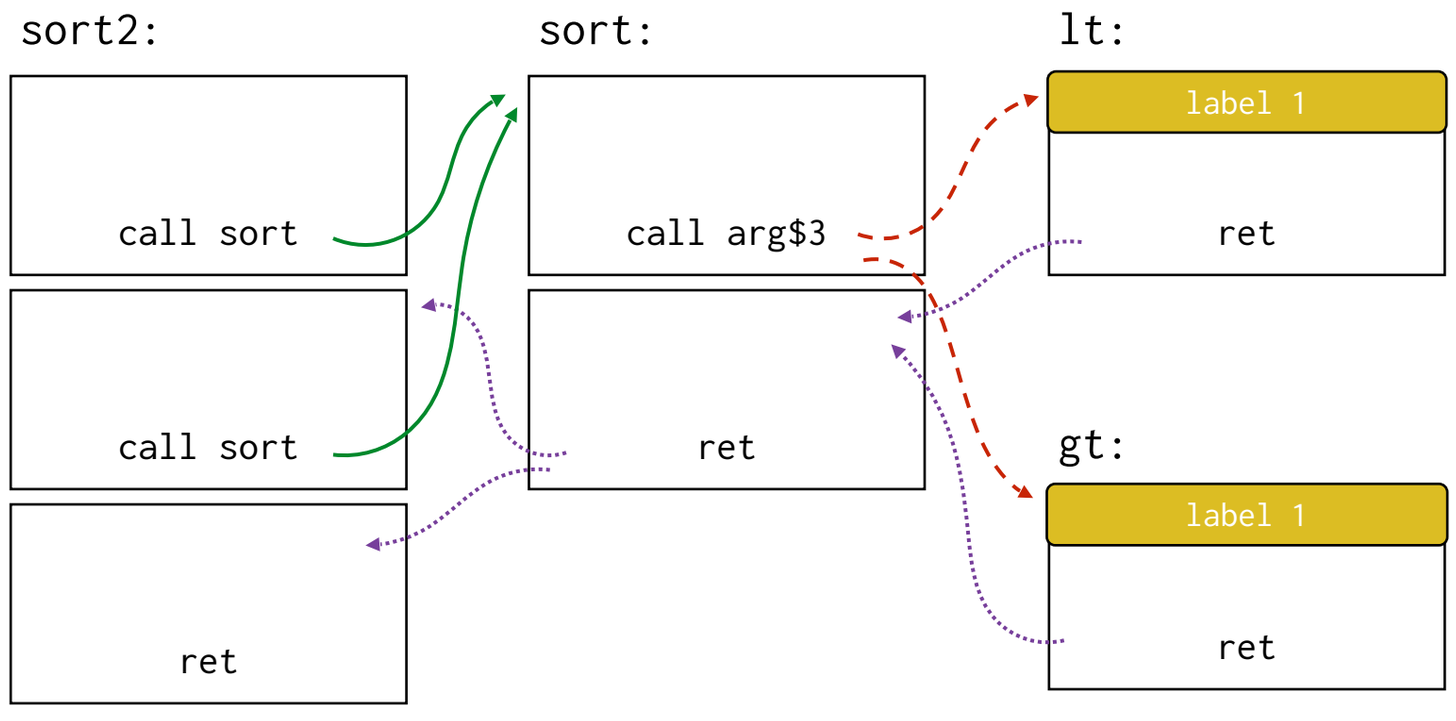
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



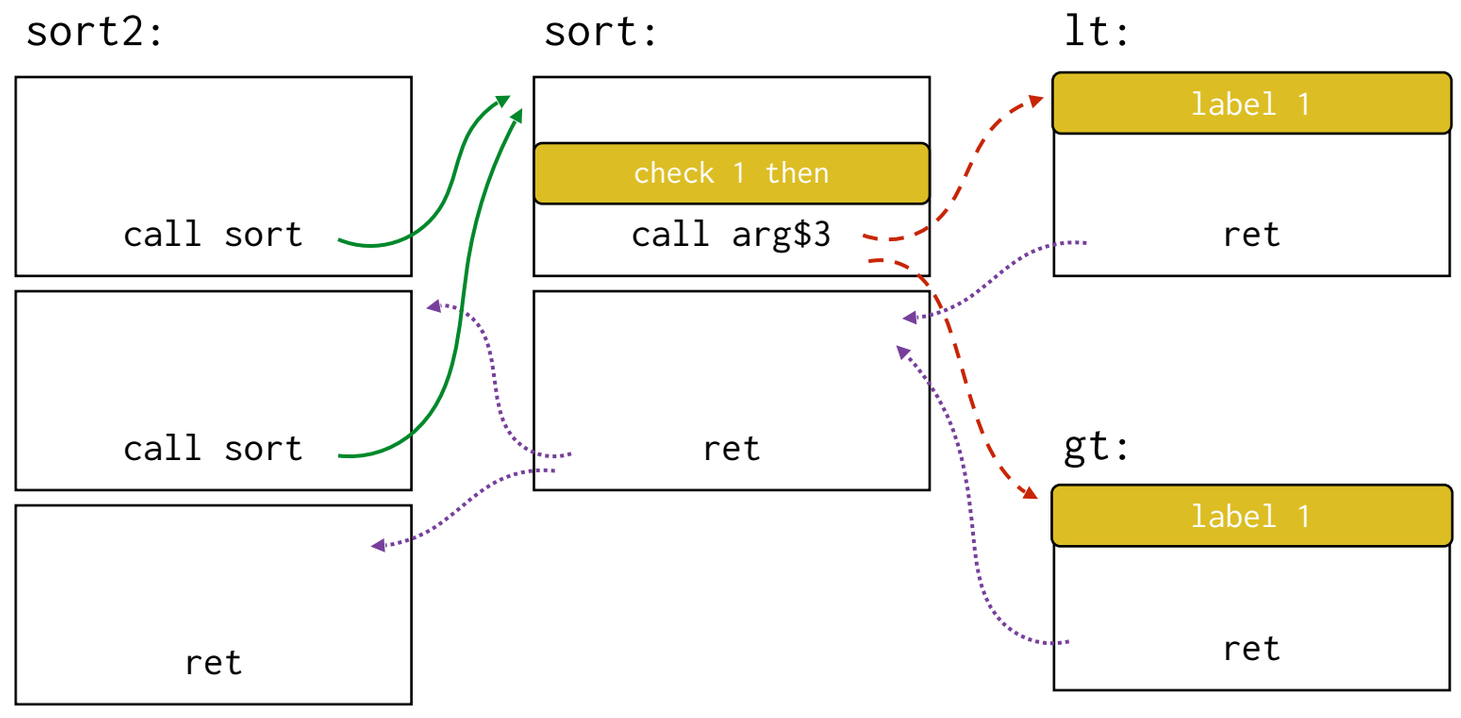
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



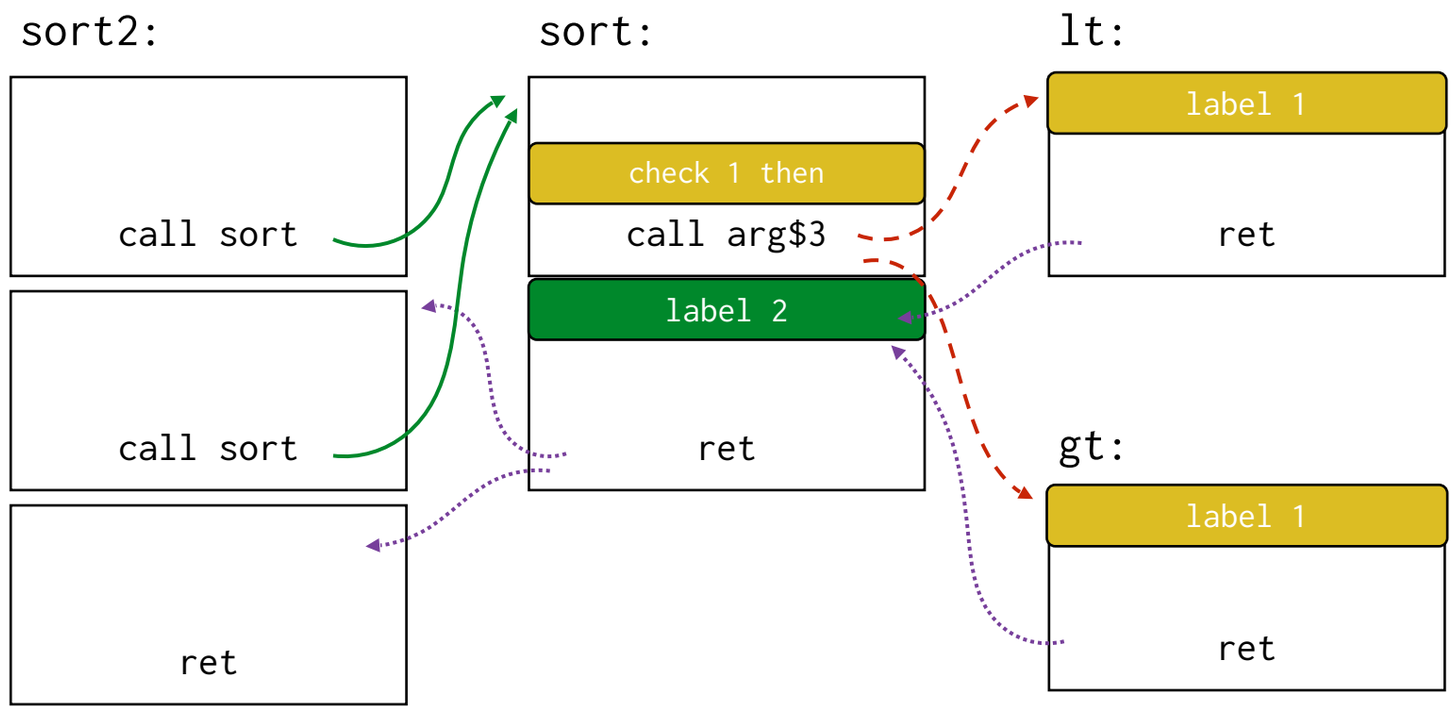
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



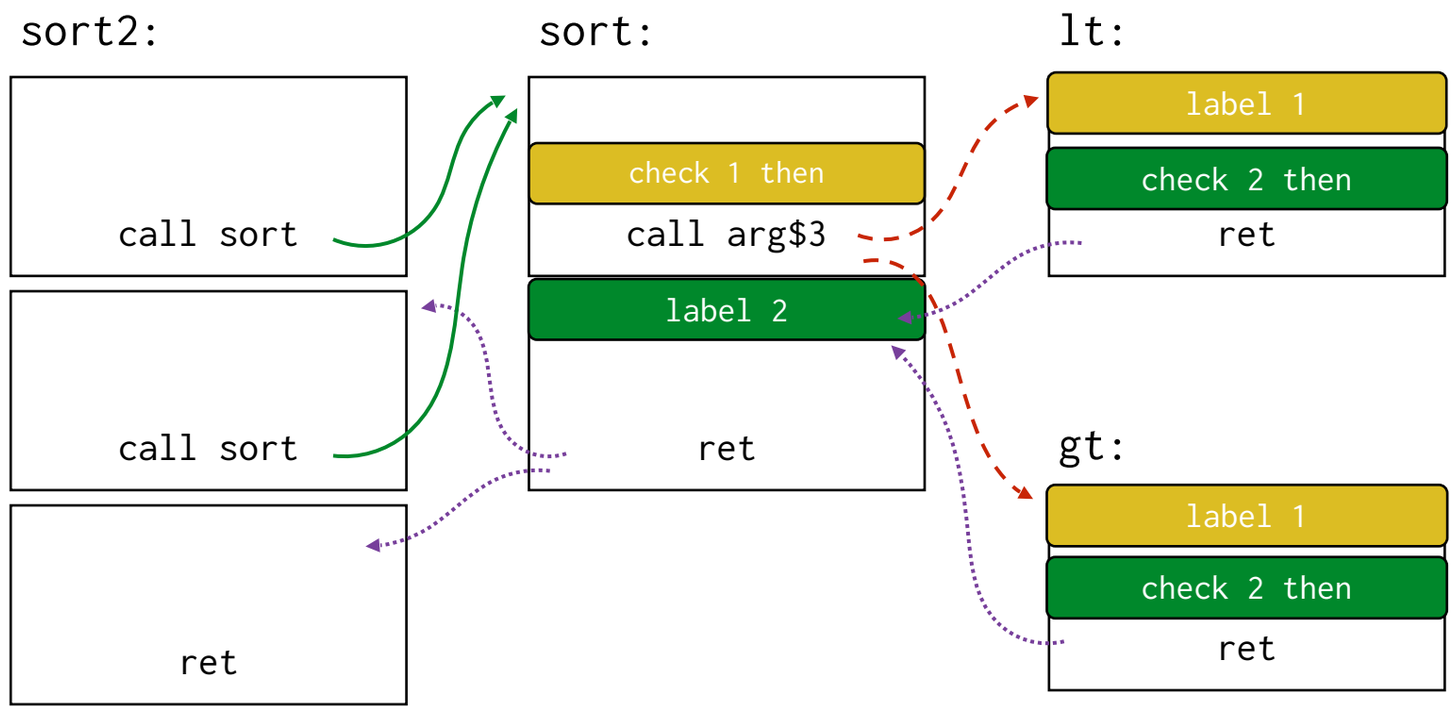
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



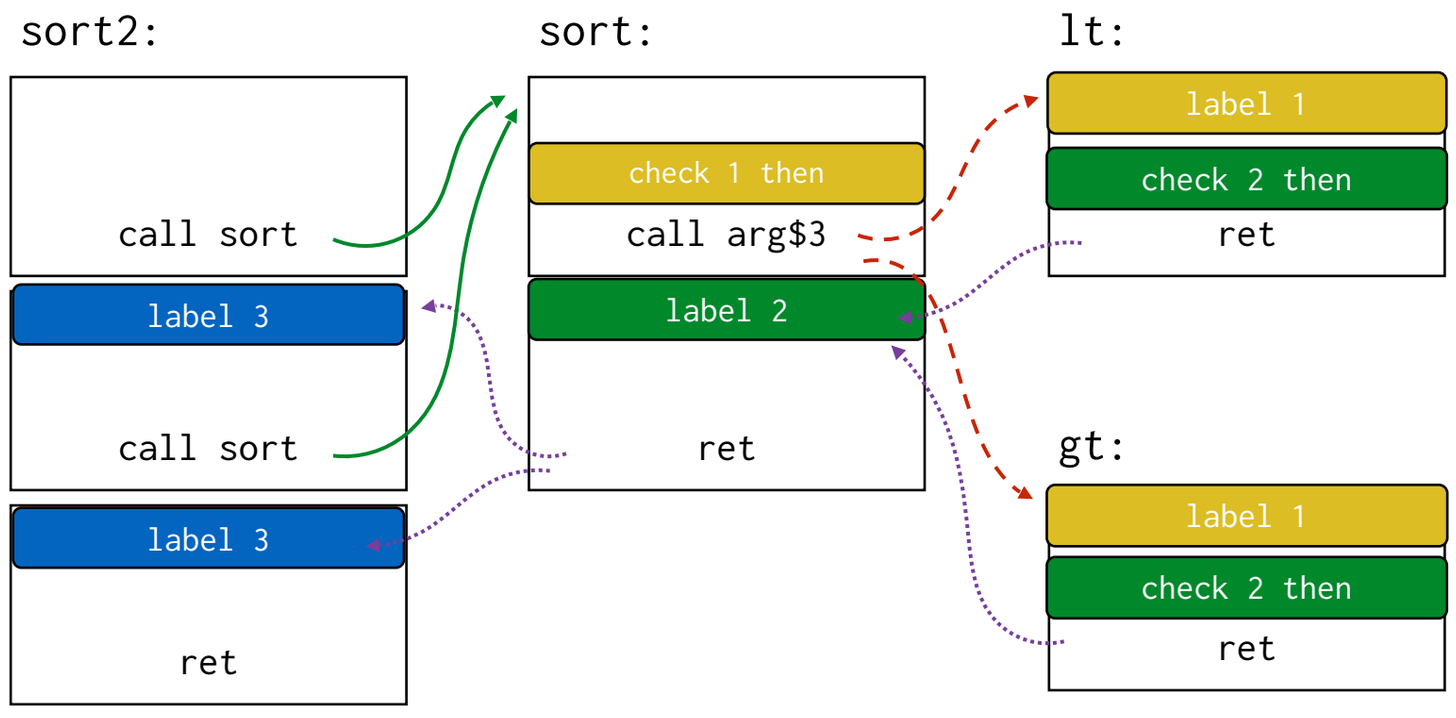
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



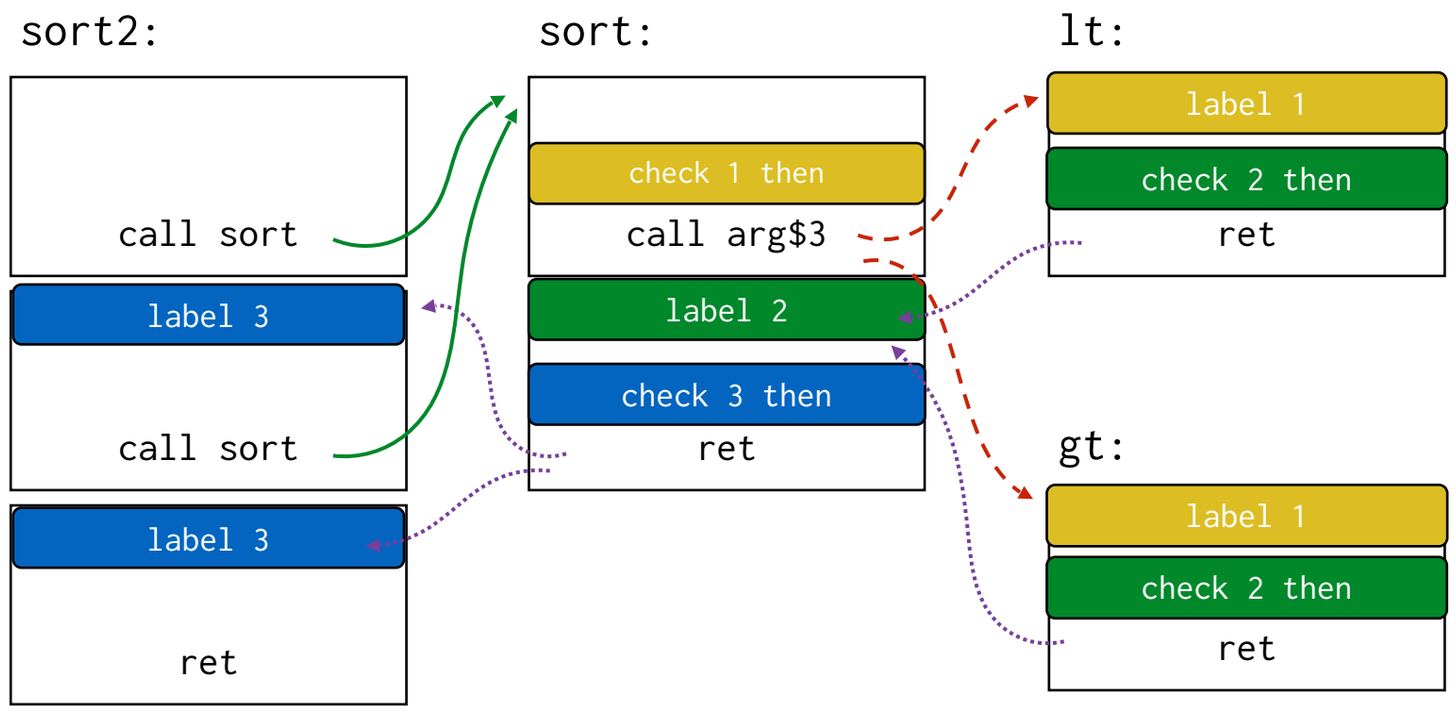
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



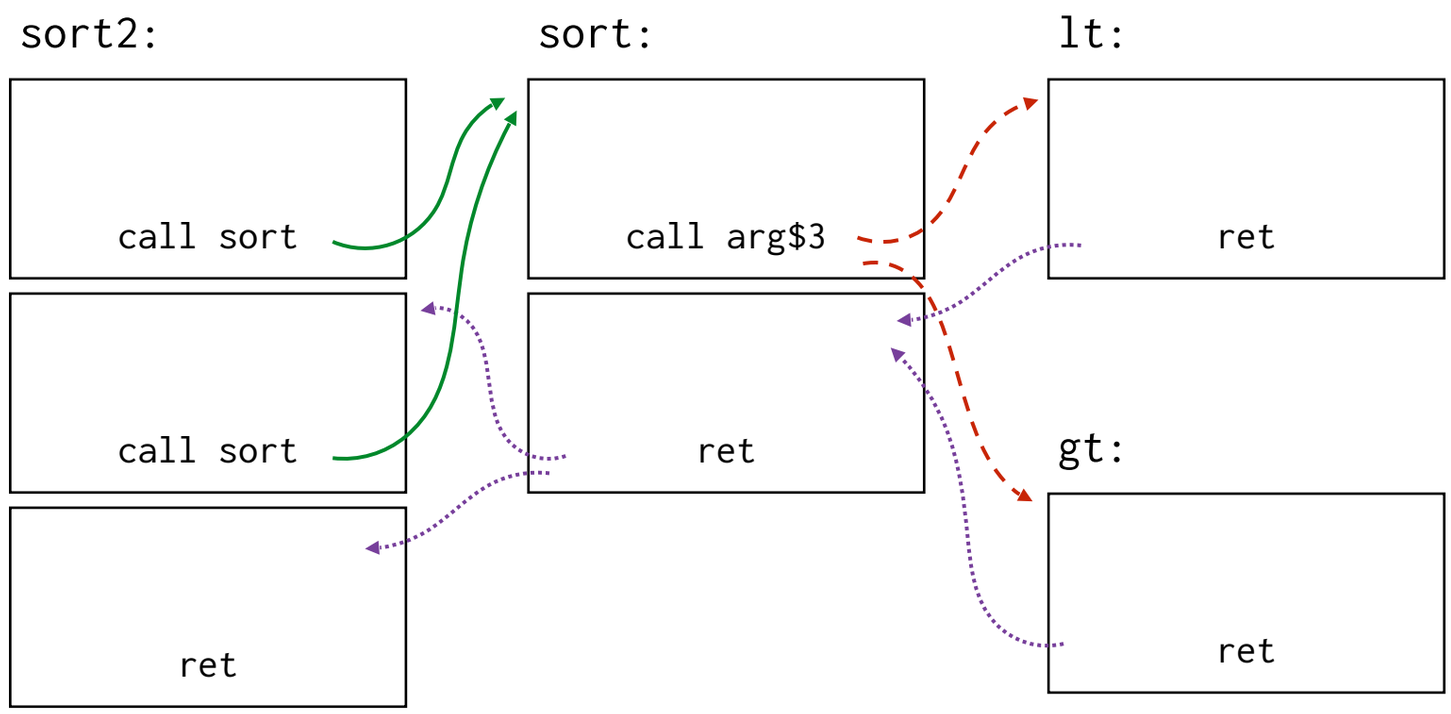
Fine grained CFI (Abadi et al.)

```
void 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;
}
```



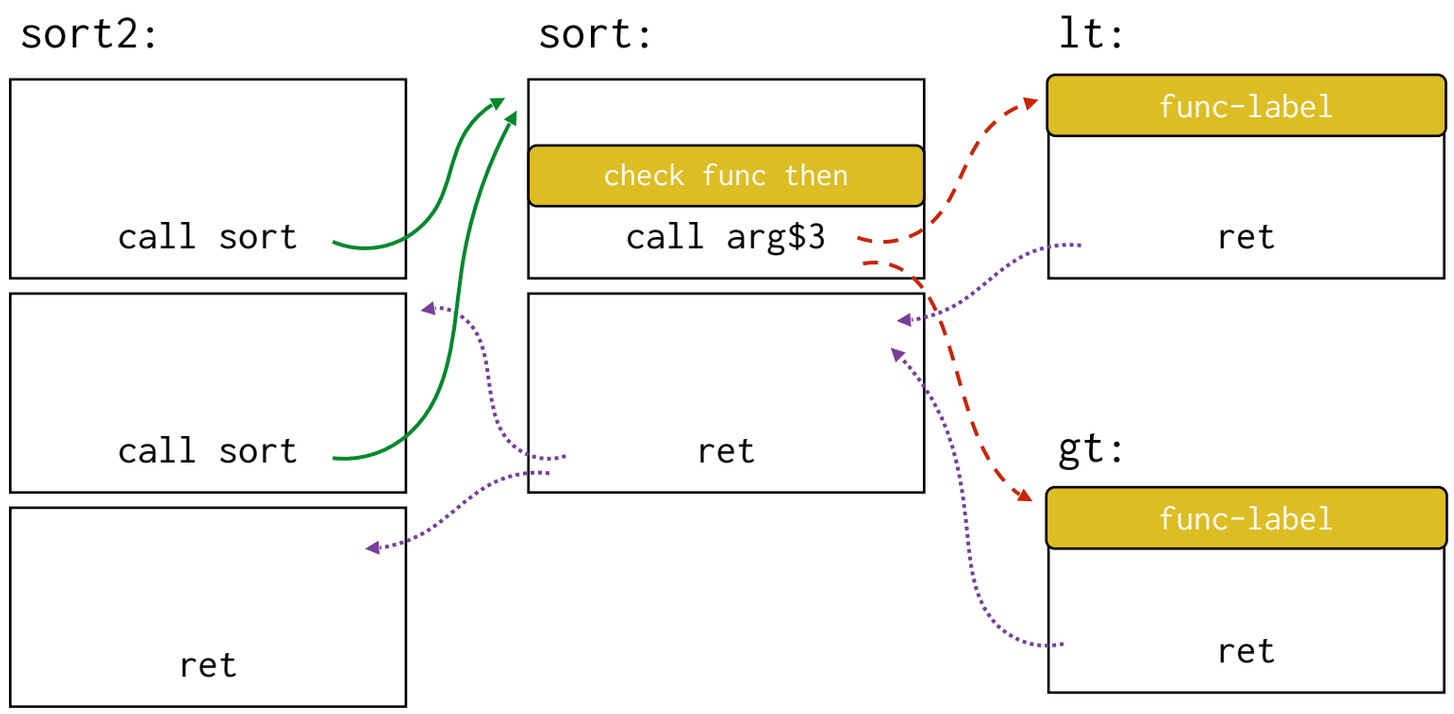
Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Every indirect call lands on function entry
- Label for destination of rets and indirect jumps
  - Every indirect jump lands at start of block



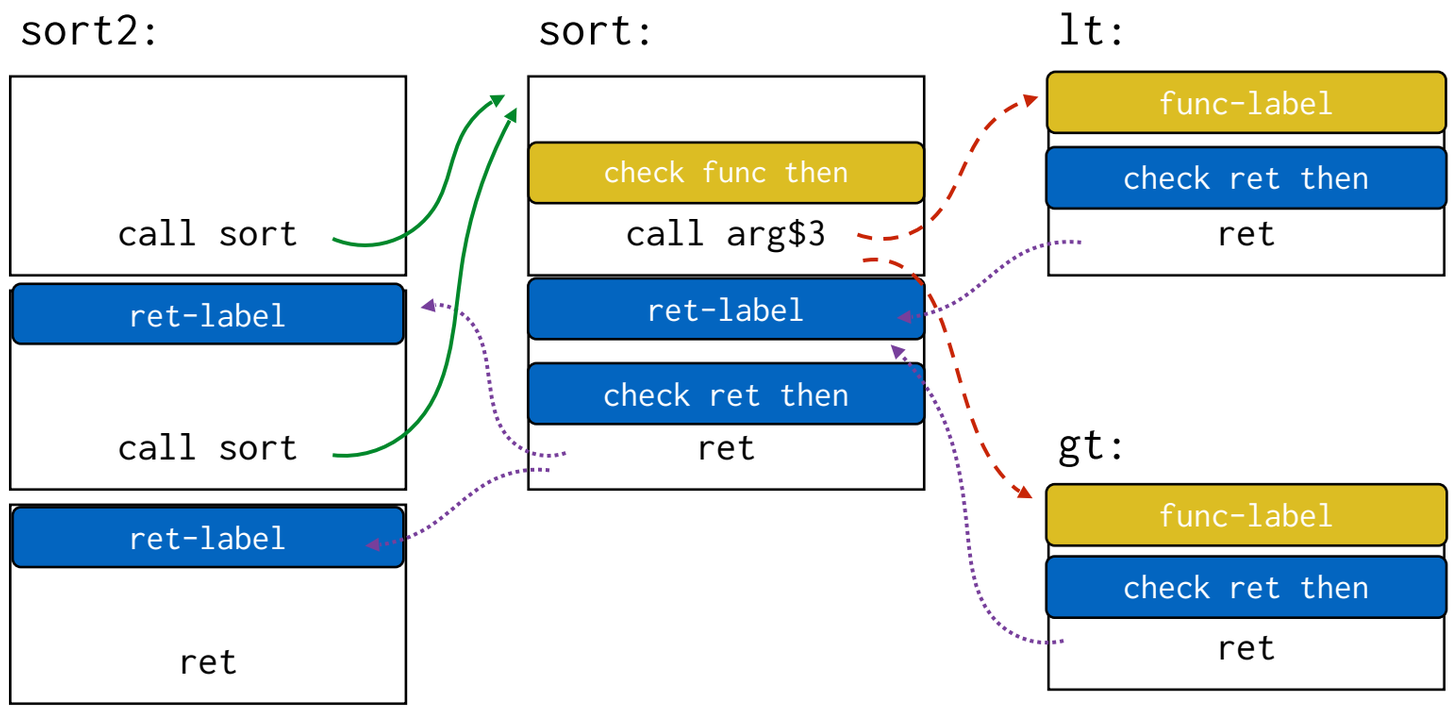
Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Every indirect call lands on function entry
- Label for destination of rets and indirect jumps
  - Every indirect jump lands at start of block



Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Every indirect call lands on function entry
- Label for destination of rets and indirect jumps
  - Every indirect jump lands at start of block



How else can you choose labels?

- Use function signature!
  - This ensures indirect calls at preserve type
  - LLVM-CFI and WebAssembly do this

CFI limitations

- Overhead
  - Runtime: every indirect branch instruction
  - Size: code before indirect branch + encode label at destination
- Scope
  - CFI does not protect against data-only attacks
  - Needs reliable W^X

How can you defeat CFI?

- Imprecision can allow for control-flow hijacking

How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
  - E.g., even if we use Wasm's labels `int system(char*)` and `int myFunc(char*)` share the same label

How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
    - E.g., even if we use Wasm's labels `int system(char*)` and `int myFunc(char*)` share the same label
  - Can return to many more sites
    - But, real way to do backward edge CFI is to use a shadow stack. (This is actually great!)

Today

- Return oriented programming (ROP)
- Control flow integrity