ROP

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: Most of slides are from CMPSC 447 of PSU(Gang Tan)

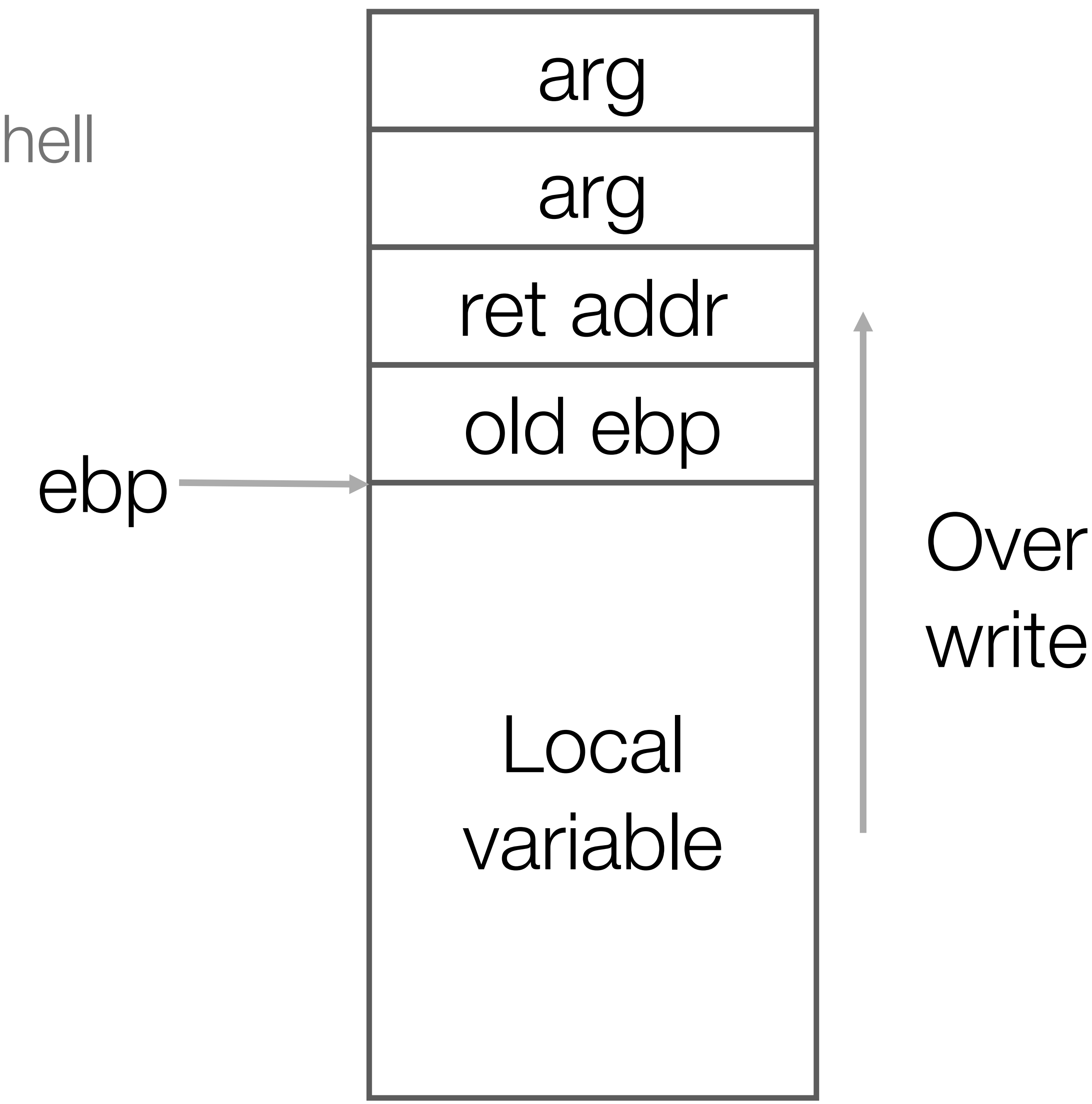
Review

Review

- Ret2libc with/without ASLR

What We Have Learnt So Far

- Stack layout
- Overwrite the return address to shell code on the stack
- Defense
  - Stack canary
  - DEP



Runtime Mitigation: DEP (NX)

- Computer architectures follow a Von-Neumann architecture
  - Storing code as data
  - This allows an attacker to inject code into stack or heap, which is supposed to store only data
- A Harvard architecture is better for security
  - Divide the virtual address space into a **data region** and a **code region**
  - The code region is readable (R) and executable (X)
  - The data region is readable (R) and writable (W)
  - No region is both writable and executable
    - An attacker can inject code into the stack, but cannot execute it

Runtime Mitigation: DEP (NX)

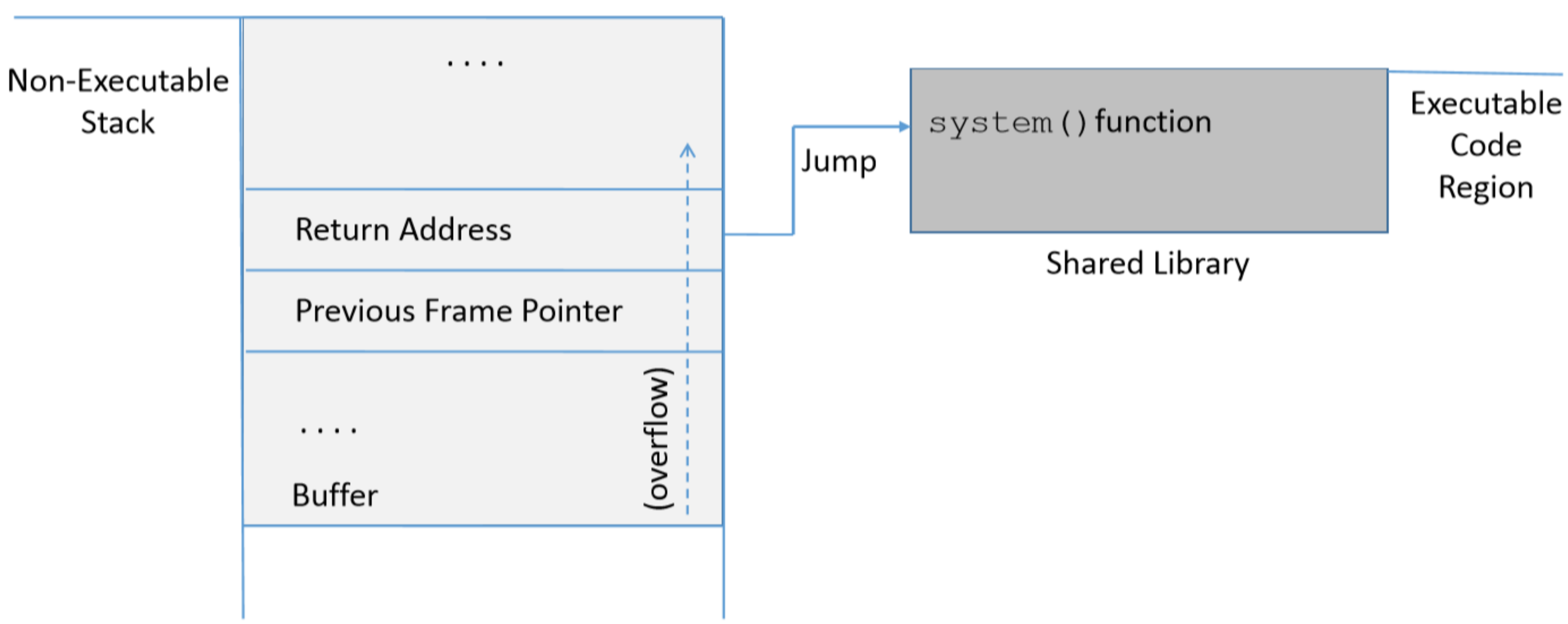
- DEP prevents code-injection attacks
    - AKA Nx-bit (non executable bit), W © X
- DEP is now supported by most OSes and ISAs

Defeating DEP: Code Reuse Attacks

- Idea: reuse code in the program (and libraries)
  - No need to inject code
- Return-to-libc: replace the return address with the address of a dangerous library function
  - attacker constructs suitable parameters on stack above return address
    - On x64, need more work of setting up parameter-passing registers
  - function returns and library function executes
    - e.g. `execve(“/bin/sh”)`
  - can even chain two library calls

How to Attack: Rethink the Stack Layout

- Step I: find the address of system function
- Step II: find the string “/bin/sh”
- Step III: pass “bin/sh” to system function



A Normal Function Call

- A normal function call
  - Caller
    - push parameters on the stack, use call instruction jump to callee, which pushes the return address on the stack
  - Callee
    - push old ebp, move esp to ebp

Function Prologue and Epilogue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

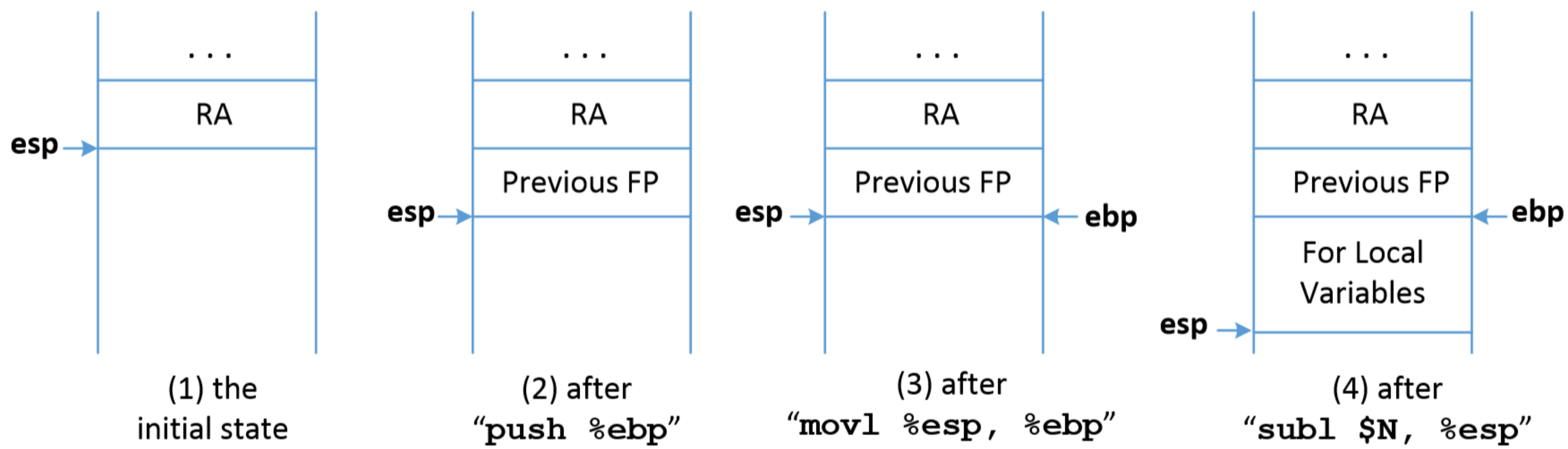
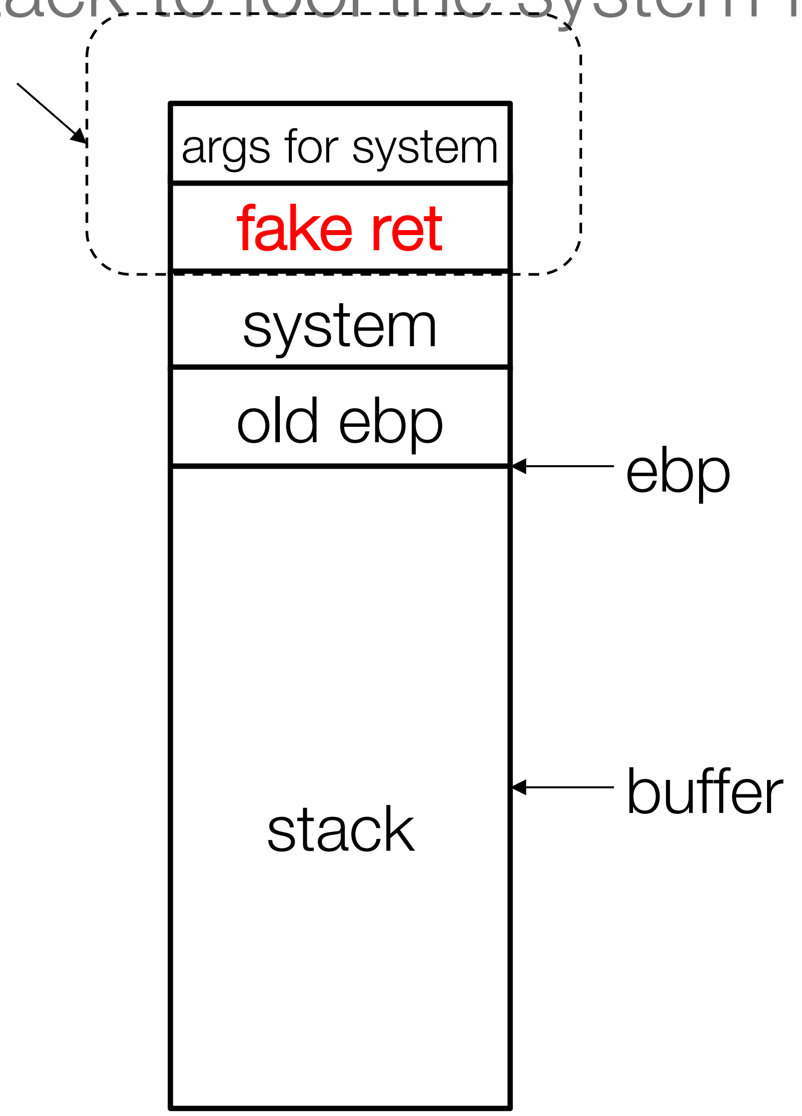


图 5.3: How the stack changes when executing the function prologue

Invoke libc function: system()

- We invoke system by redirecting the return address on the stack, we need to make up the stack to fool the system function.



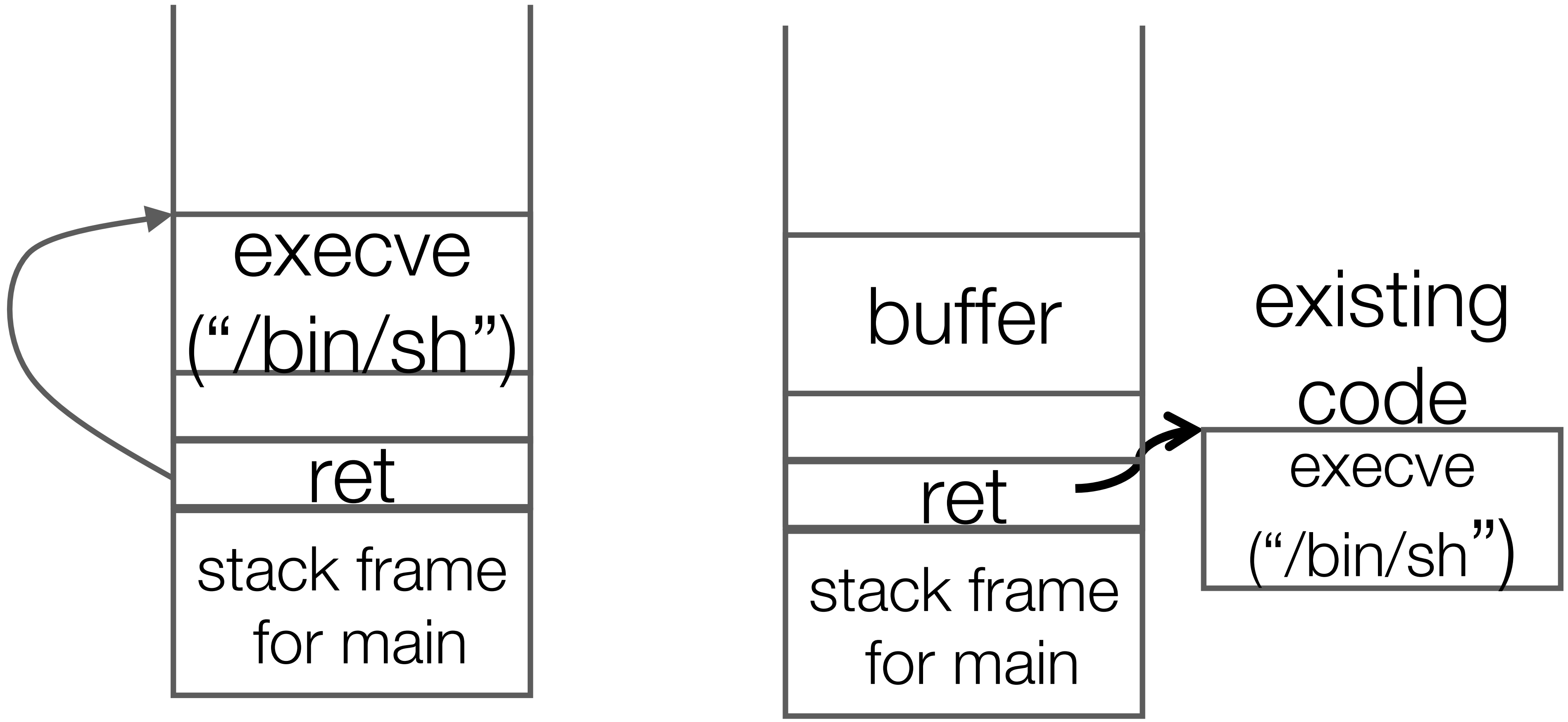
Stack **layout**
to invoke system()
function

ROP

Code Injection vs Code Reuse

- Ret2libc is a code reuse attack
- The difference is subtle, but significant
  - In **code injection**, we wrote the address of execve into buffer on the stack and modified return address to start executing at buffer
    - I.e., we are executing in the stack memory region
  - In **code reuse**, we can modify the return address to point to execve directly, so we continue to execute code
    - Reusing available code to do what the adversary wants

Code Injection vs Code Reuse



code injection   code reuse

Code Reuse

- In many attacks, a code reuse attack is used as a first step to disable DEP
  - Goal is to allow execution of stack memory
  - There's a system call for that

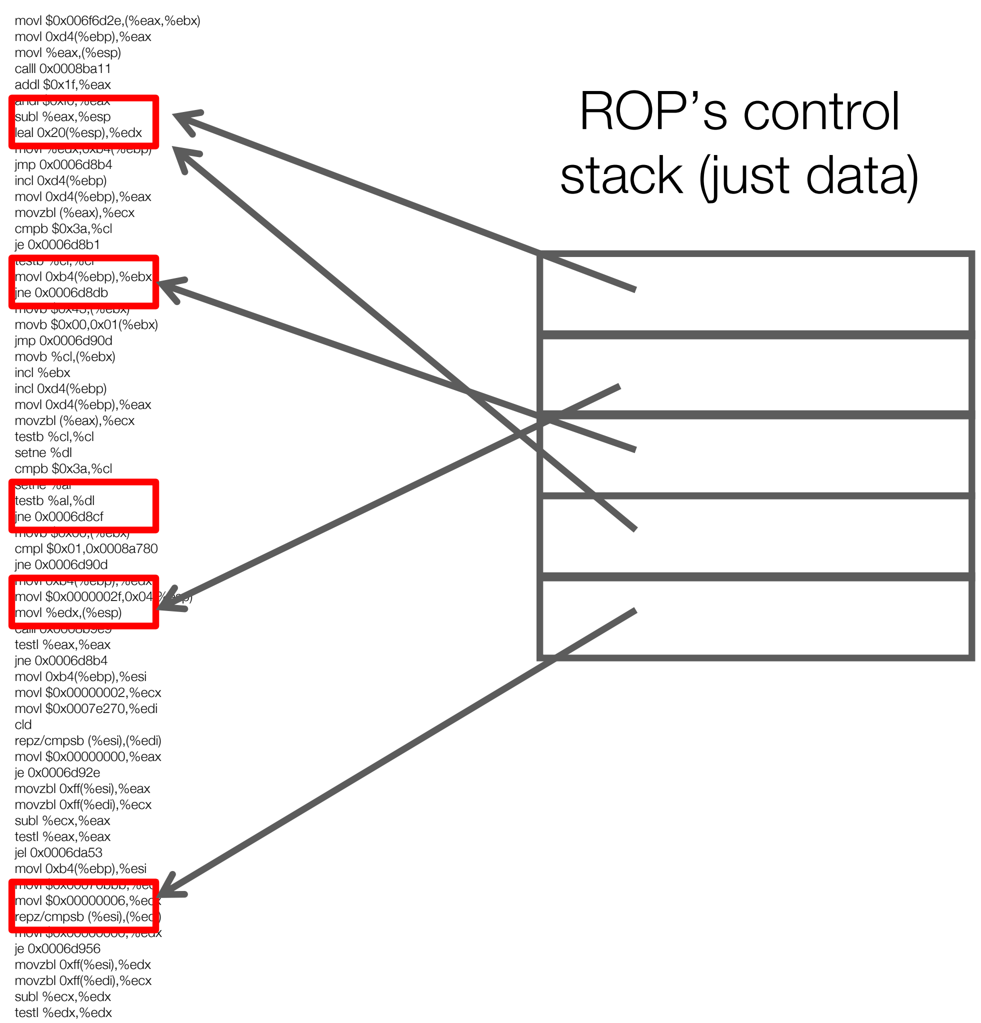
```
int mprotect(void *addr, size_t len, int prot);
```

  - Sets protection for region of memory starting at address
  - Invoke this libray API (system call) to allow execution on stack and then start executing from the injected code

Code Reuse: ROP

- Return-Oriented Programming (ROP)
  - [Shacham et al], 2008
  - **Arbitrary behavior without code injection**
  - Combine snippets of existing code (gadgets)
  - A set of Turing-complete gadgets and a way of chaining these gadgets
  - People have shown that in small programs (e.g., 16KB), they can find a Turing-complete set of gadgets

ROP: Illustrated



- Use gadgets to perform general programming
  - arithmetics;
  - arbitrary control flow: jumps; loops; …

Return-Oriented Programming

*The following slides are by Dr. Shacham

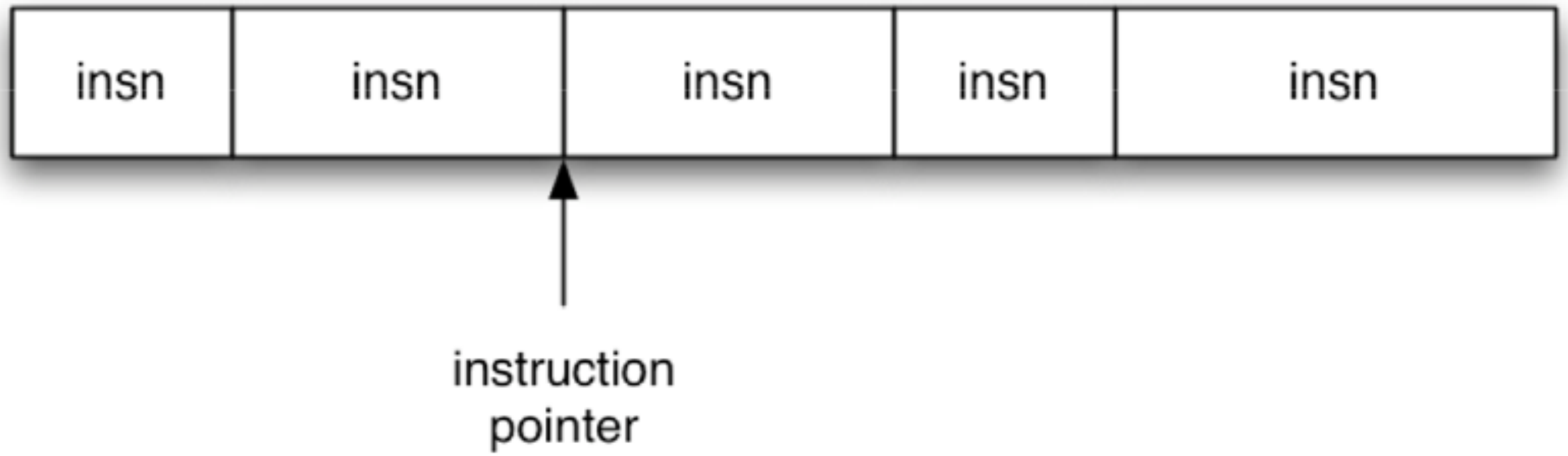
ROP Thesis

any sufficiently large program codebase

↓ ↓

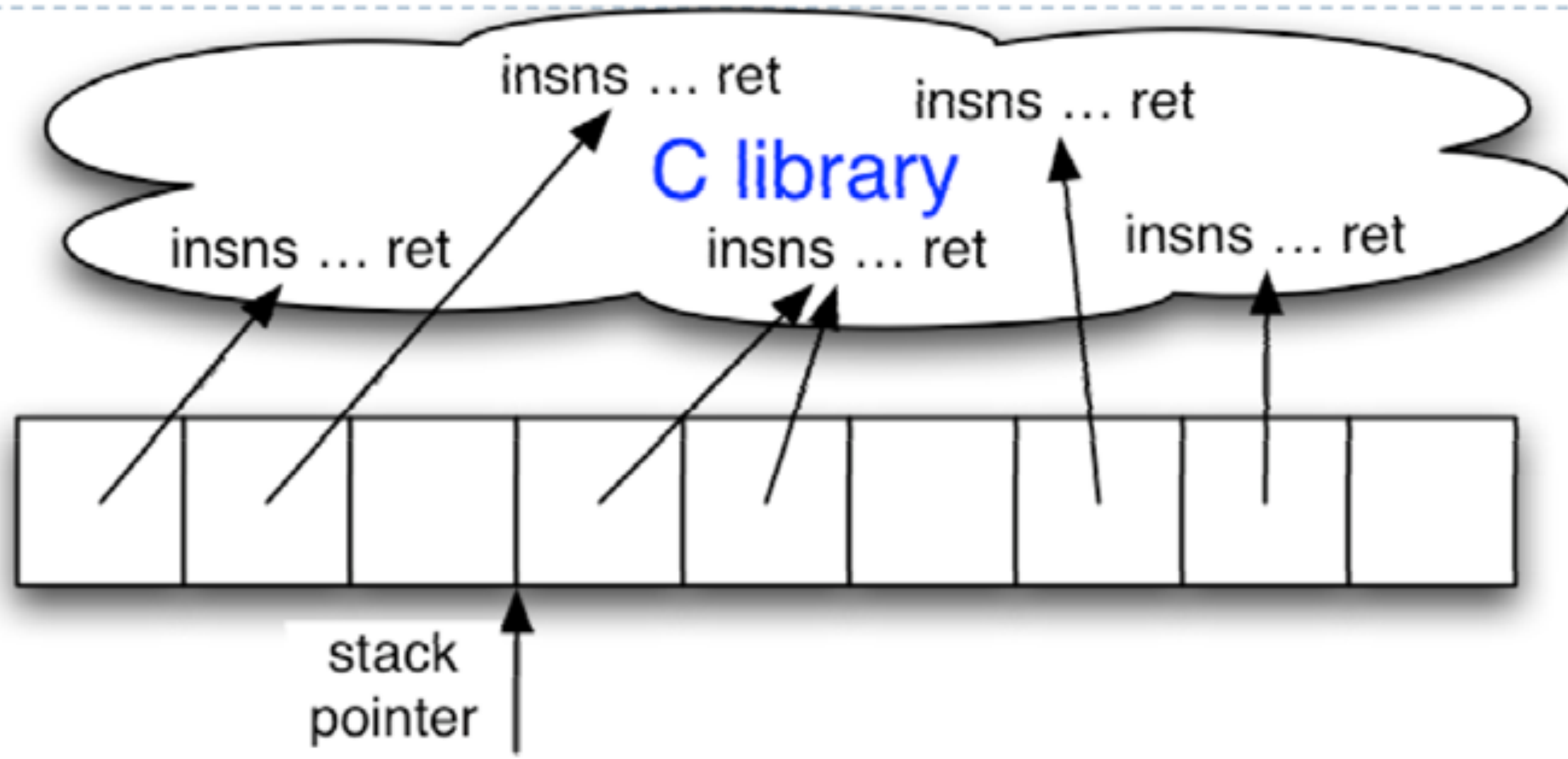
arbitrary attacker computation and behavior,
without code injection

Normal Machine Instructions



- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip

ROP Execution



- *Stack pointer* (%esp) determines which instruction sequence to fetch & execute
- Processor doesn't automatically increment %esp; — but the "ret" at end of each instruction sequence does

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

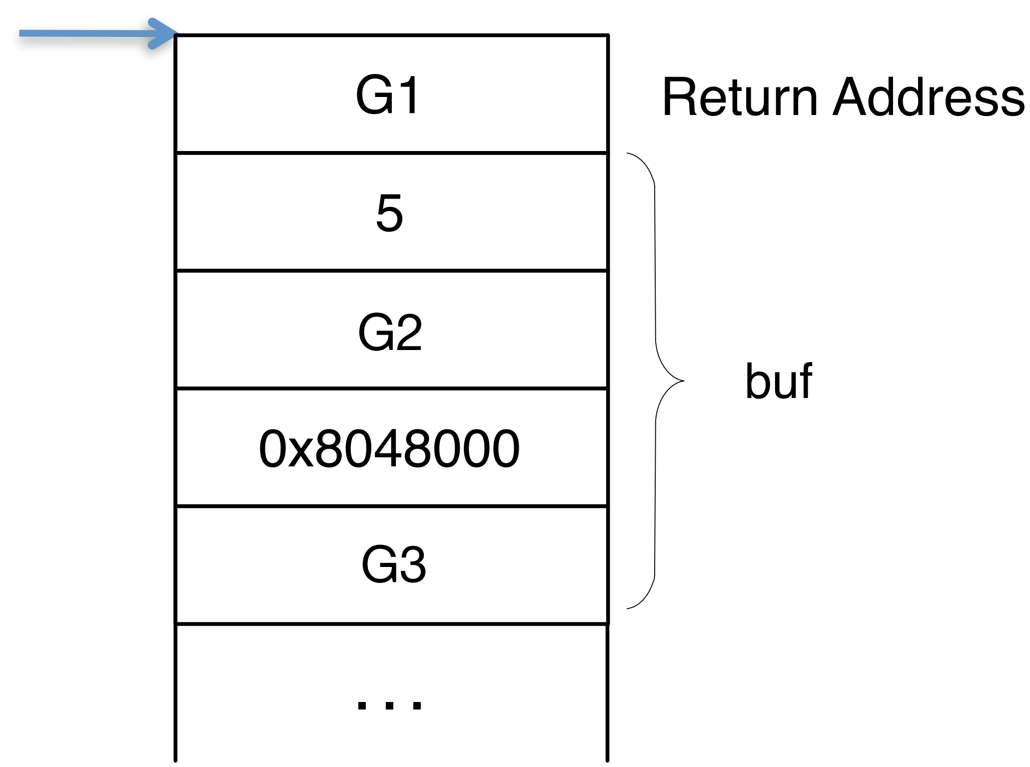
Code

G1: pop %eax
ret

G2: pop %ebx
ret

G3: movl %eax, (%ebx)
ret

Stack



Registers

%eax =

%ebx =

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

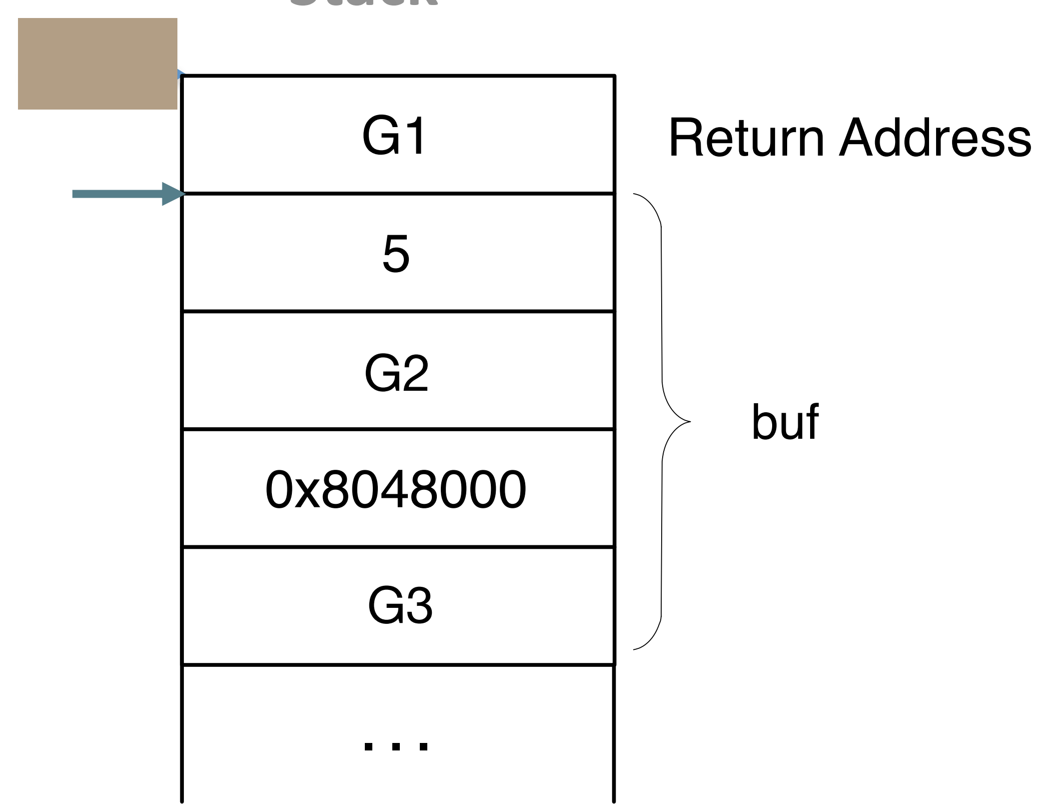
Code

G1:
```
pop %eax
ret
```

G2:
```
pop %ebx
ret
```

G3:
```
movl %eax, (%ebx)
ret
```

Stack



Registers

%eax = 5

%ebx =

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

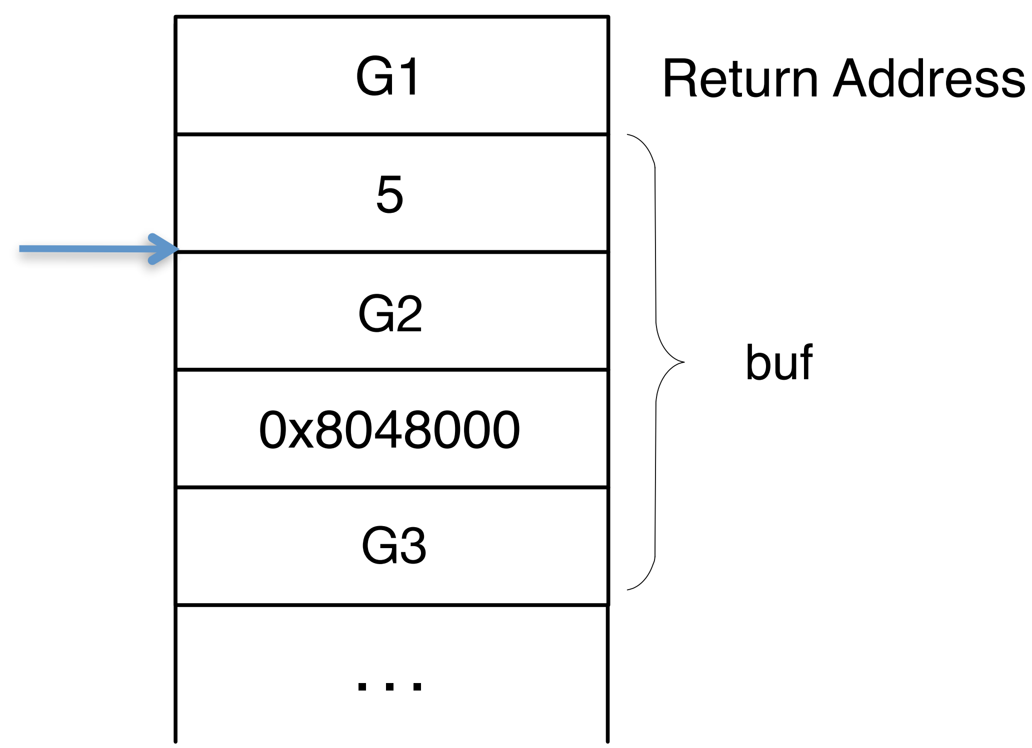
Code

```
G1:   pop %eax
      ret

G2:   pop %ebx
      ret

G3:   movl %eax, (%ebx)
      ret
```

Stack



Registers

%eax = 5

%ebx =

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

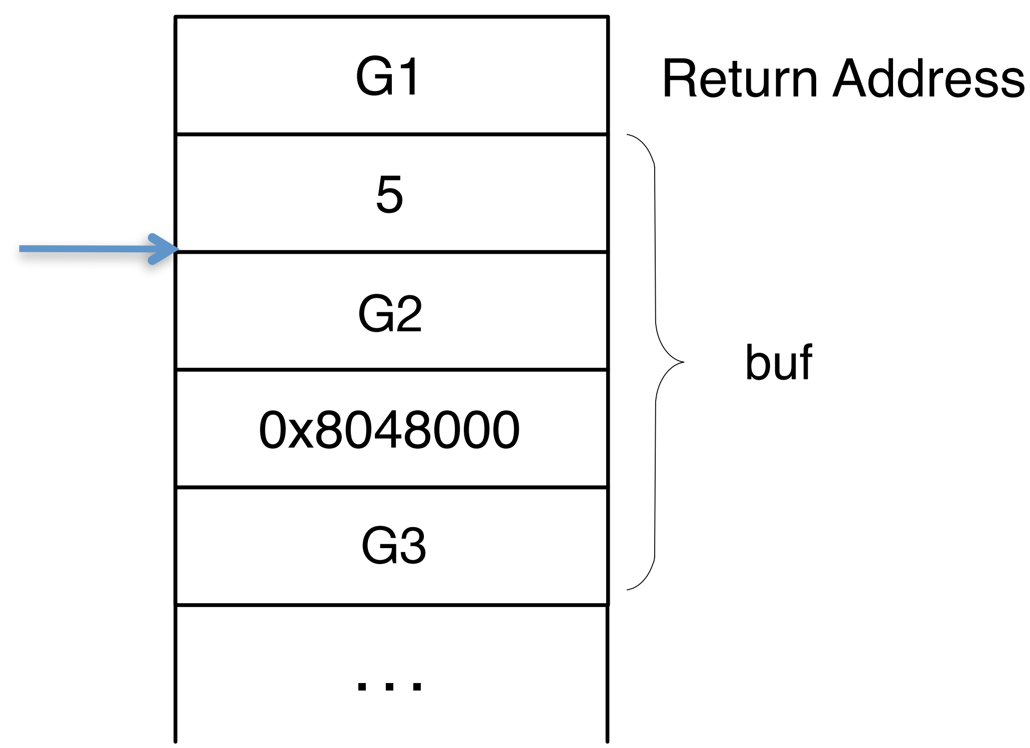
Code

```
G1:   pop %eax
      ret

G2:   pop %ebx
      ret

G3:   movl %eax, (%ebx)
      ret
```

Stack



Registers

%eax = 5

%ebx = 0x8048000

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

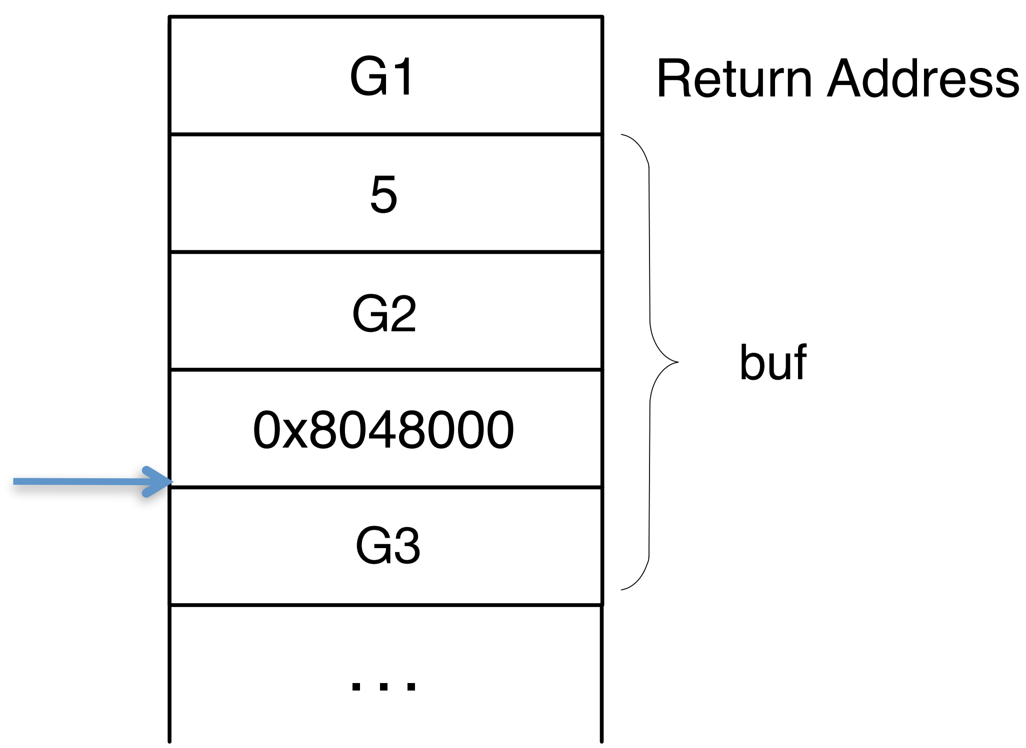
Code

```
G1:   pop %eax
      ret

G2:   pop %ebx
      ret

G3:   movl %eax, (%ebx)
      ret
```

Stack



Registers

%eax = 5

%ebx = 0x8048000

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

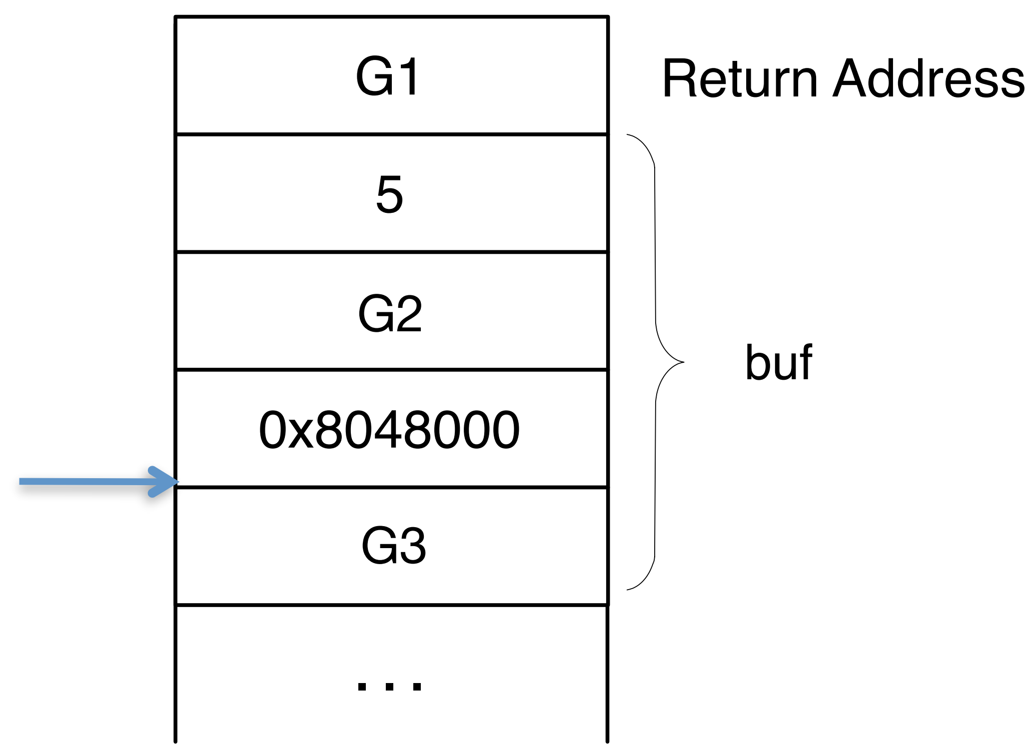
Code

```
G1:   pop %eax
      ret

G2:   pop %ebx
      ret

G3:   movl %eax, (%ebx)
      ret
```

Stack



Registers

%eax = 5

%ebx = 0x8048000

Memory

0x8048000 =

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

Code

```
G1:   pop %eax
      ret

G2:   pop %ebx
      ret

G3:   movl %eax, (%ebx)
      ret
```

Stack

Stack	
G1	Return Address
5	buf
G2	buf
0x8048000	buf
G3	buf
...	

Registers

%eax = 5

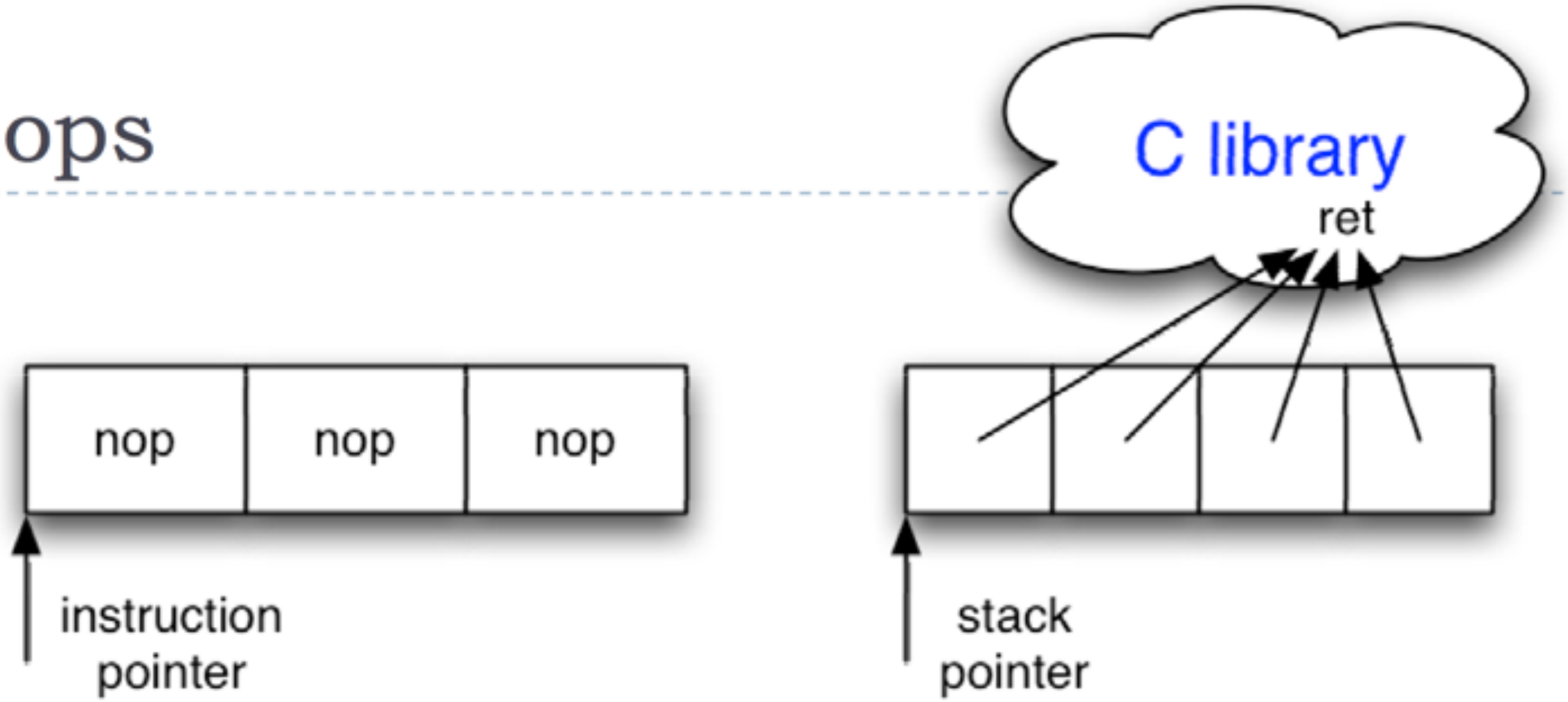
%ebx = 0x8048000

Memory

0x8048000 = 5

Building ROP Functionality

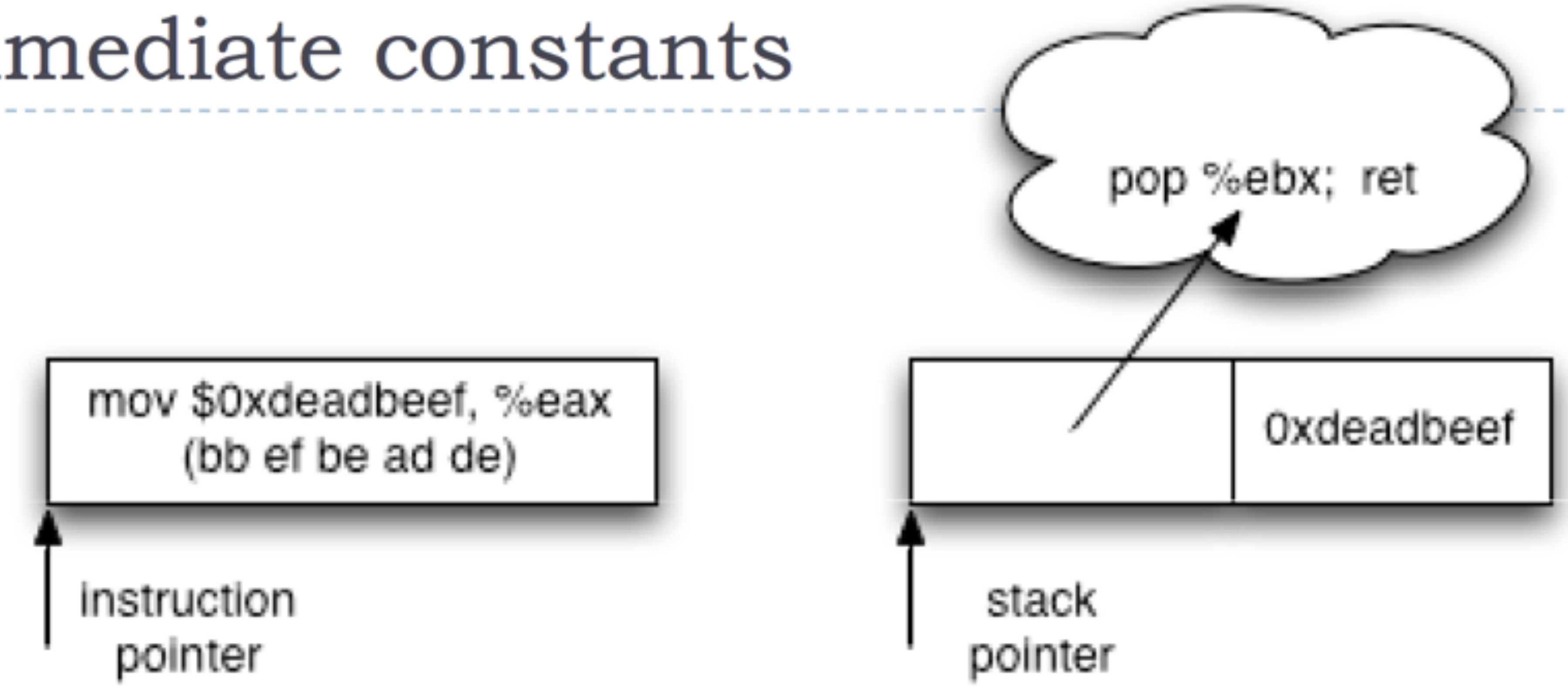
No-ops



- No-op instruction does nothing but advance %eip
- Return-oriented equivalent:
  - point to return instruction
  - advances %esp
- Useful in nop sled

Building ROP Functionality

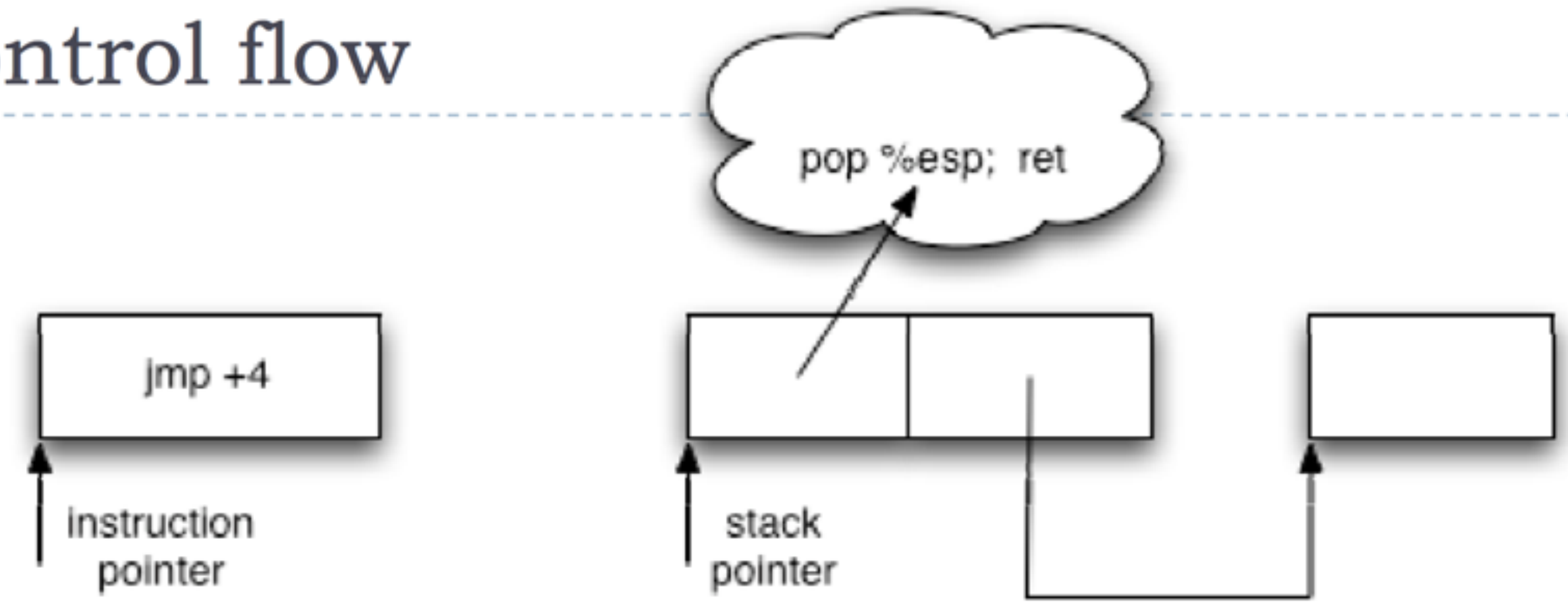
Immediate constants



- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use

Building ROP Functionality

Control flow



- Ordinary programming:
  - (Conditionally) set %eip to new value
- Return-oriented equivalent:
  - (Conditionally) set %esp to new value

Return-oriented Programming

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know?

Return-oriented Programming

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know? Turing completeness
- A language is Turing complete if it has (loosely)
  - Conditional branching
  - Can change memory arbitrarily
- Both are possible in ROP

Protection against ROP

- ROP works by changing the control flow of the program
- Control-flow integrity (CFI)
  - Take a vulnerable program and a pre-determined **a control-flow graph**
  - Insert checks into the program so that it stops working if an illegal control flow transfer happens during runtime
    - **Via compiler changes or binary rewriting**
  - More on this later

Runtime Mitigation: Randomization

- Exploits requires knowing code/data addresses
  - E.g., the start address of a buffer
  - E.g., the address of a library function
- Idea: introduce artificial diversity (randomization)
  - Make addresses unpredictable for attackers
- Many ways of doing randomization
  - Randomize **location of the stack, location of key data structures on the heap, and location of library functions**
  - Randomly pad stack frames
  - At compile time, randomize code generation for defending against ROP

Implementation of Randomization

- Can be performed
  - At compile time
  - At link time
  - Or at runtime (e.g., via dynamic binary rewriting)

Linux Address-Space Layout Randomization (ASLR)

- **For a position-independent executable (PIE), randomize**
  - **The base address of the executable**
- All libraries are PIE
  - So their base addresses are randomized
- **Main executables may not be PIE**
  - **May not be protected by ASLR**
- A form of coarse-grained randomization
  - Only the base address is randomized
  - Relative distances between memory objects are not changed

Ways of Defeating ASLR

- Perform an exhaustive search, if the random space is small
    - E.g., Linux provides 16-bit of randomness
        - It can be defeated by an exhaustive search in about 200s
- ASLR often defeated by **memory disclosure**
    - E.g., if the attacker can read the value of a pointer to the stack
        - Then he can use it to discover where the stack is

Summary

- Code injection vs code reuse
- How ROP works
- Ways to mitigate ROP