# CSE543 Computer Security

## Module: Return-Oriented Programming

Asst. Prof. Syed Rafiul Hussain

# Anatomy of Control-Flow Exploits

- Two steps in control-flow exploitation
- First -- attacker gets control of program flow (return address, function pointer)
  - Stack (buffer), heap, format string vulnerability, …
- Second -- attacker uses control of program flow to launch attacks
  - E.g., Code injection
    - Adversary injects malcode into victim
    - E.g., onto stack or into other data region

  - How is code injection done?

# Code Injection

- Advantage
  - Adversary can install any code they want
    - What code do adversaries want?
  - Defenses
    - NX bit - set memory as non-executable (stack)
    - W (xor) X - set memory as either writeable or executable, but not both

- What can adversary do to circumvent these defenses and still execute useful code (for them)?

# Return-Oriented Programming

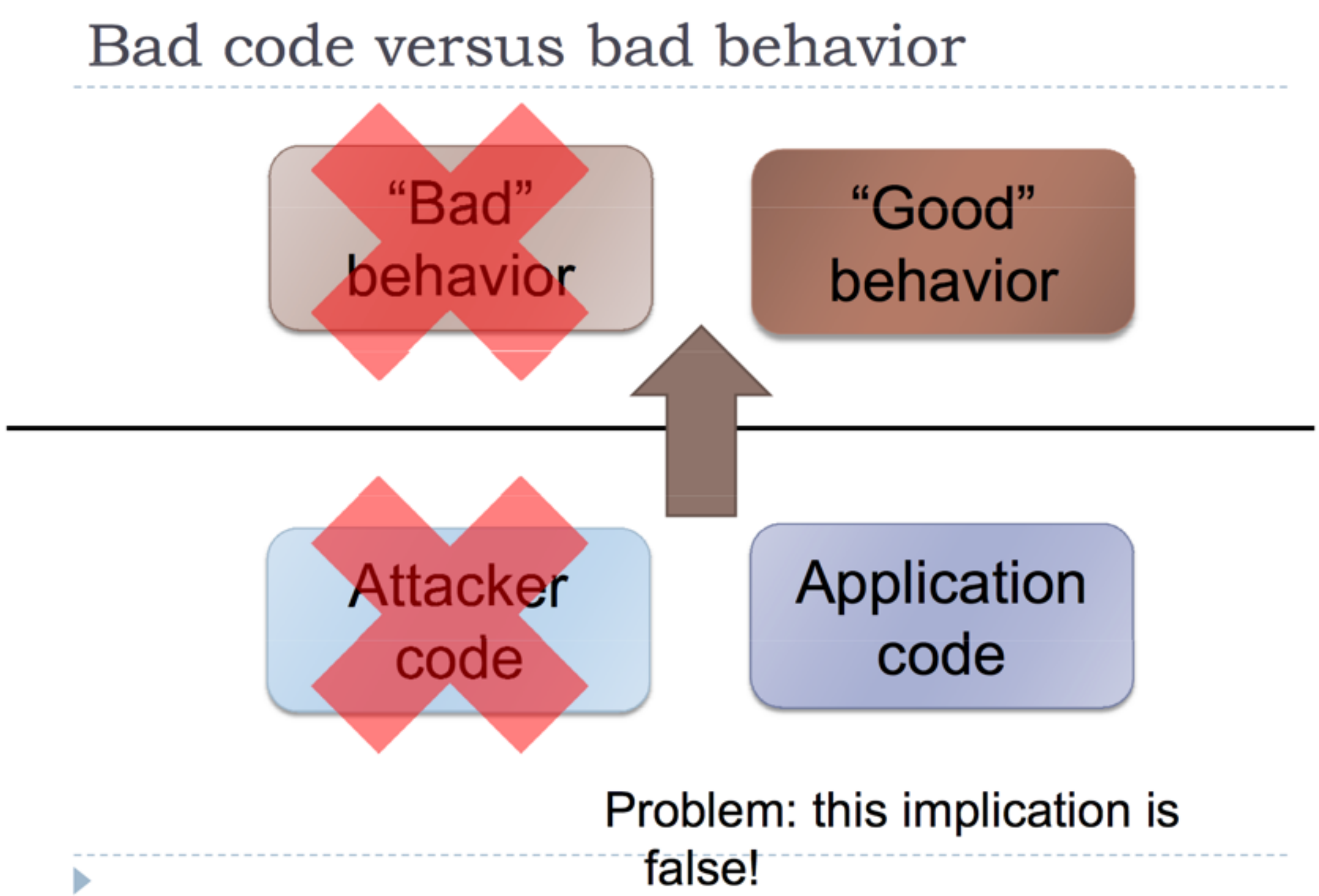
- Arbitrary exploitation without code injection

## Return-oriented Programming: Exploitation without Code Injection

Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham
University of California, San Diego

# Return-Oriented Programming

PennState



# ROP Thesis

any sufficiently large program codebase

↓ ↓

arbitrary attacker computation and behavior,
*without* code injection

(in the absence of control-flow integrity)

# Return-to-libc

- Divert control flow of exploited program into libc code
  - system(), printf(),
- No code injection required

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc — remove system()?

- We show: this perception is *false.*

# ROP vs return-to-libc

attacker control of stack

arbitrary attacker computation and behavior
via return-into-libc techniques

(given any sufficiently large codebase to draw on)

# ROP Example

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

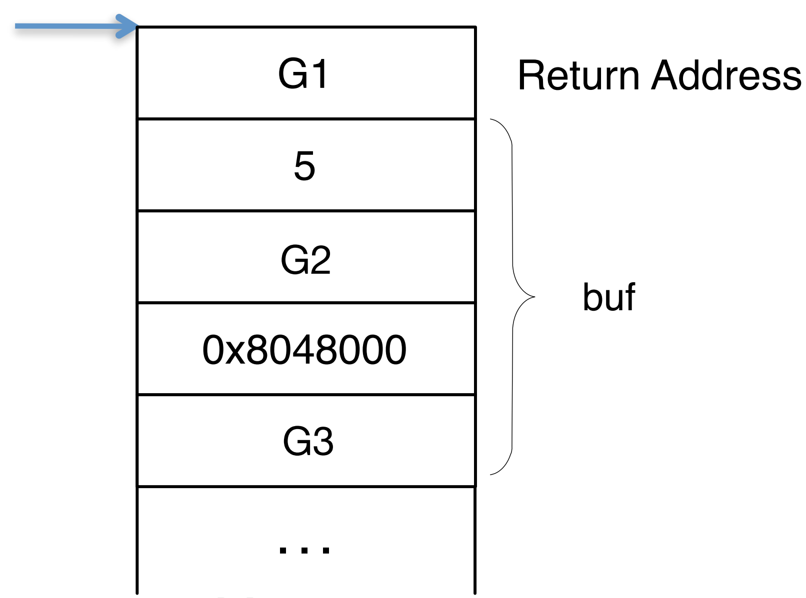
**Code**

```
G1:     pop %eax
        ret

G2:     pop %ebx
        ret

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

**Stack**




**Registers**

%eax =

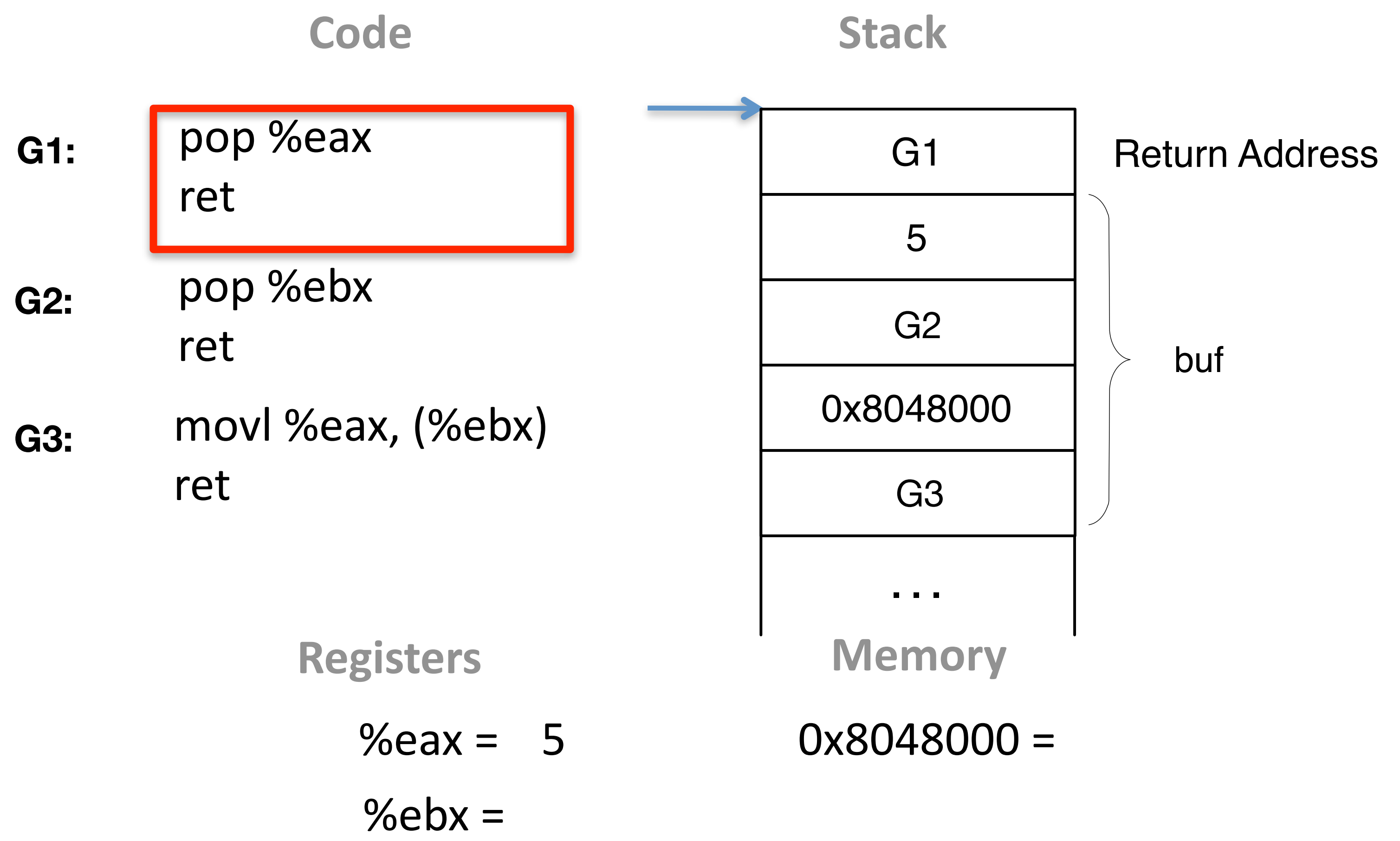
%ebx =

**Memory**

0x8048000 =

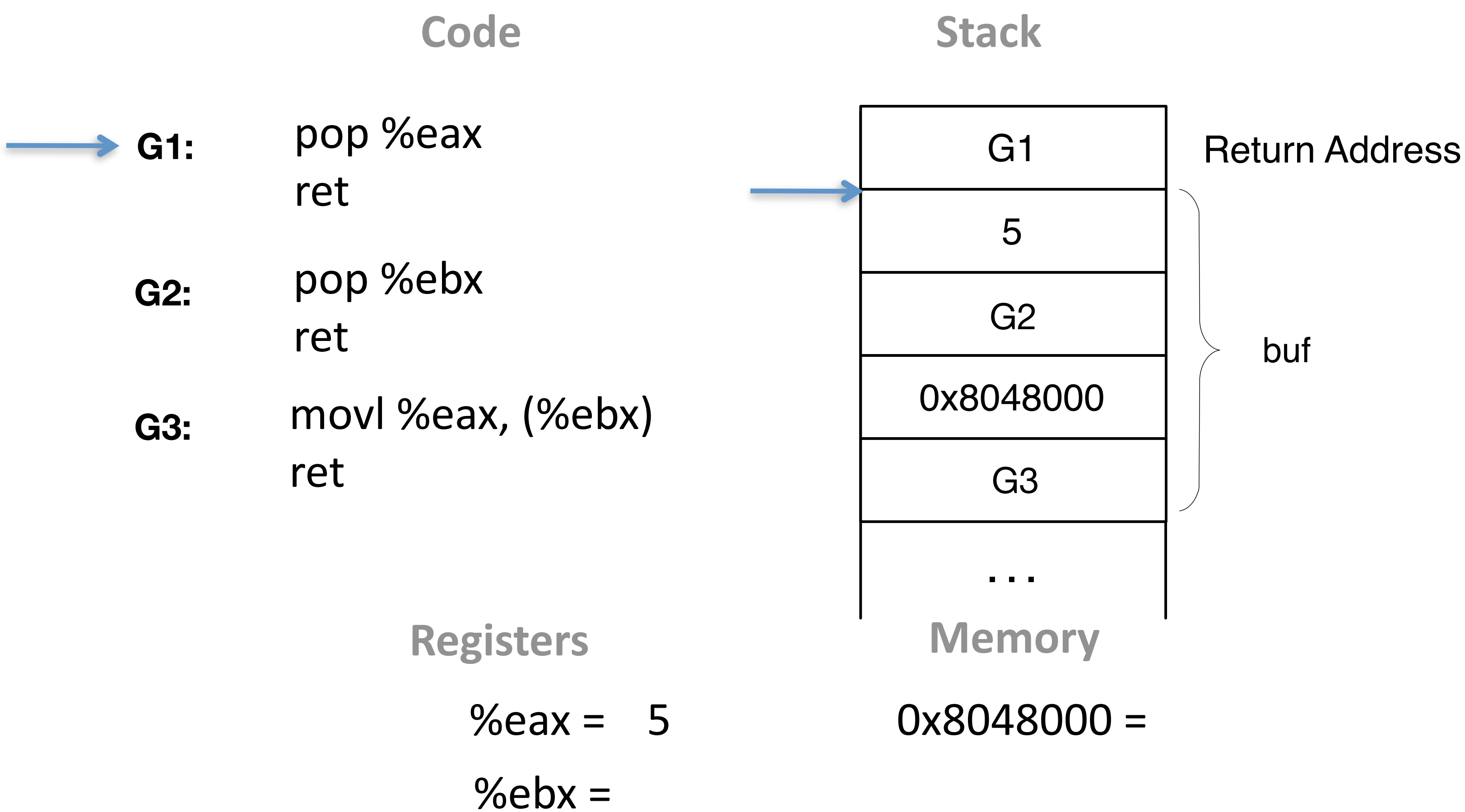
# ROP Example

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



# ROP Example

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



# ROP Example

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

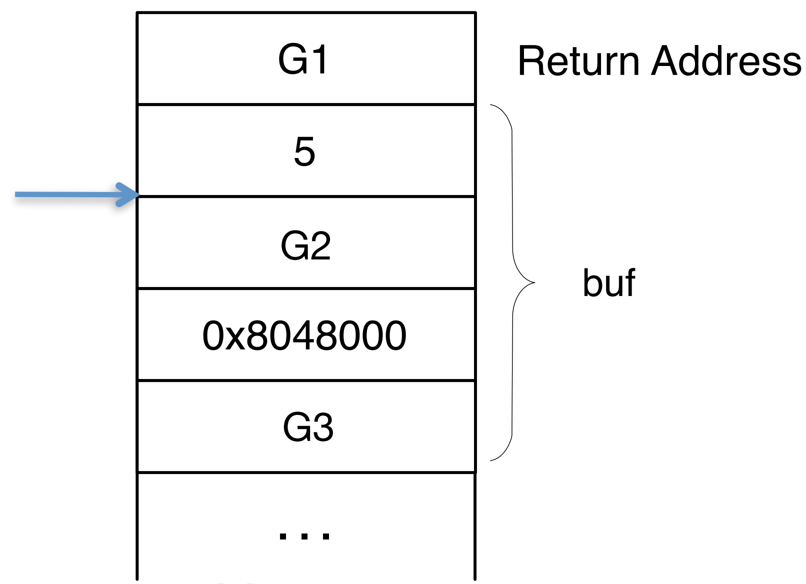
**Code**

```
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)

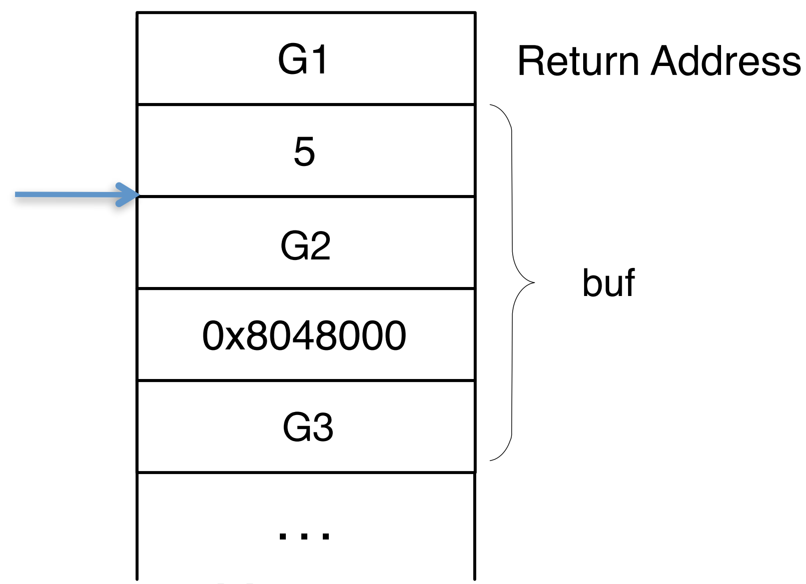
**Code**

**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)

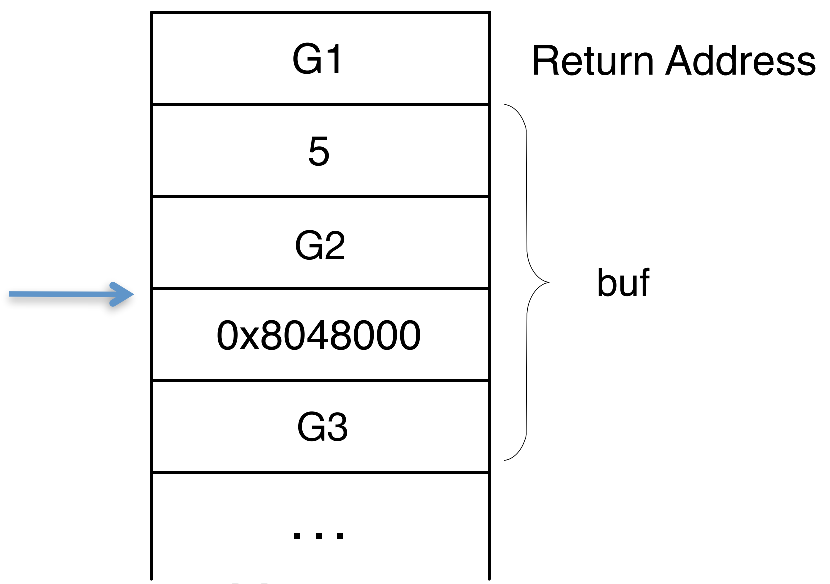
**Code**

```
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)

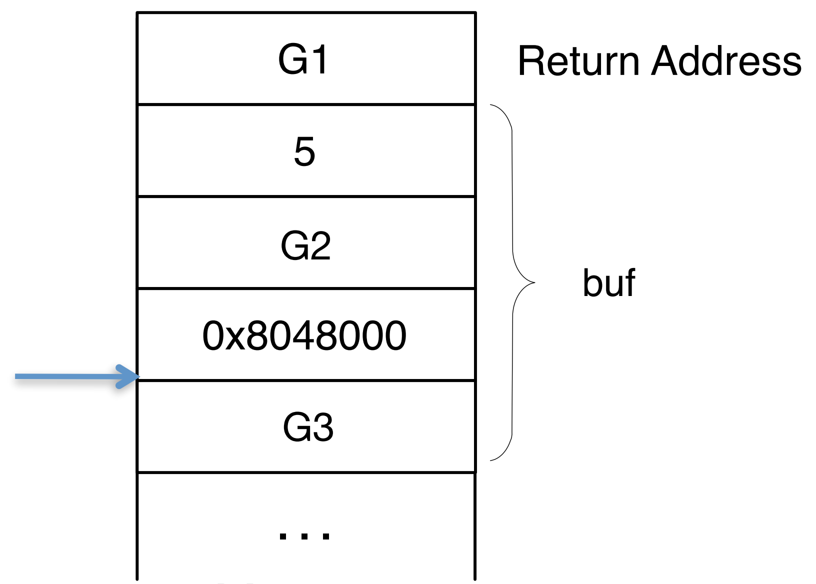
**Code**

```
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)

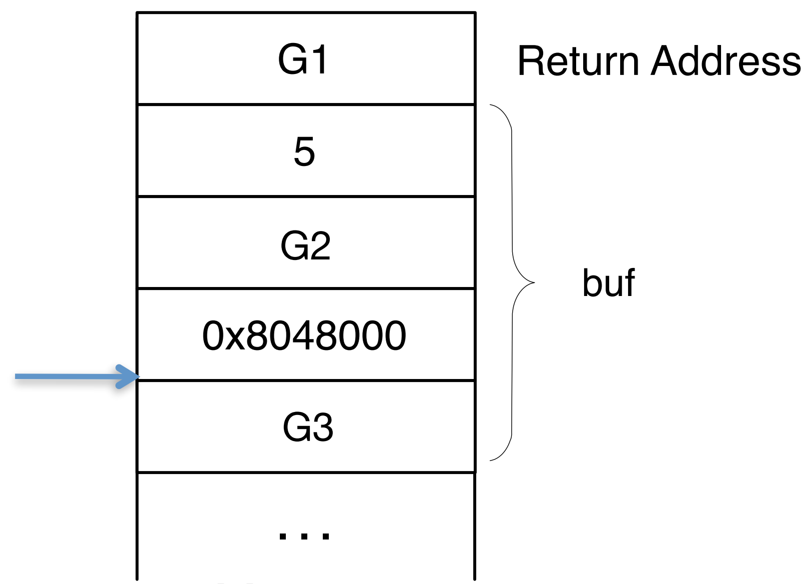
**Code**

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

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

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

**Stack**




**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)

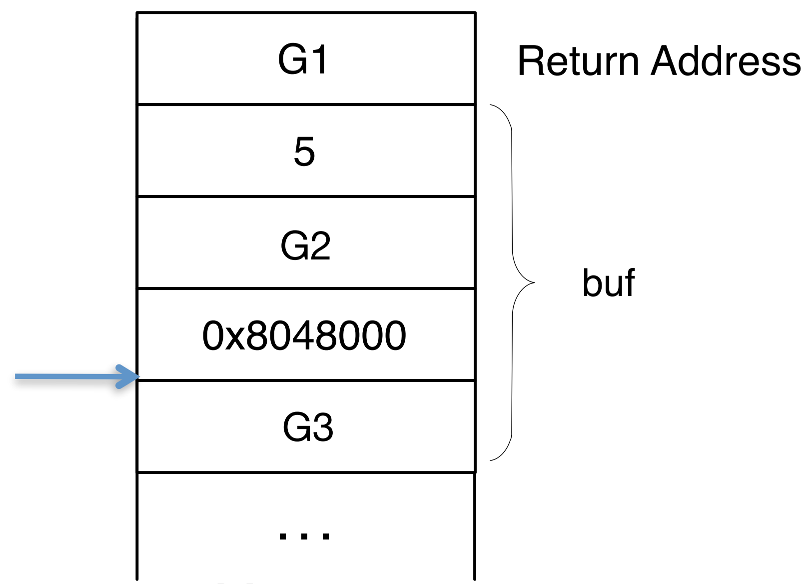
**Code**

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

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

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

**Stack**




**Registers**

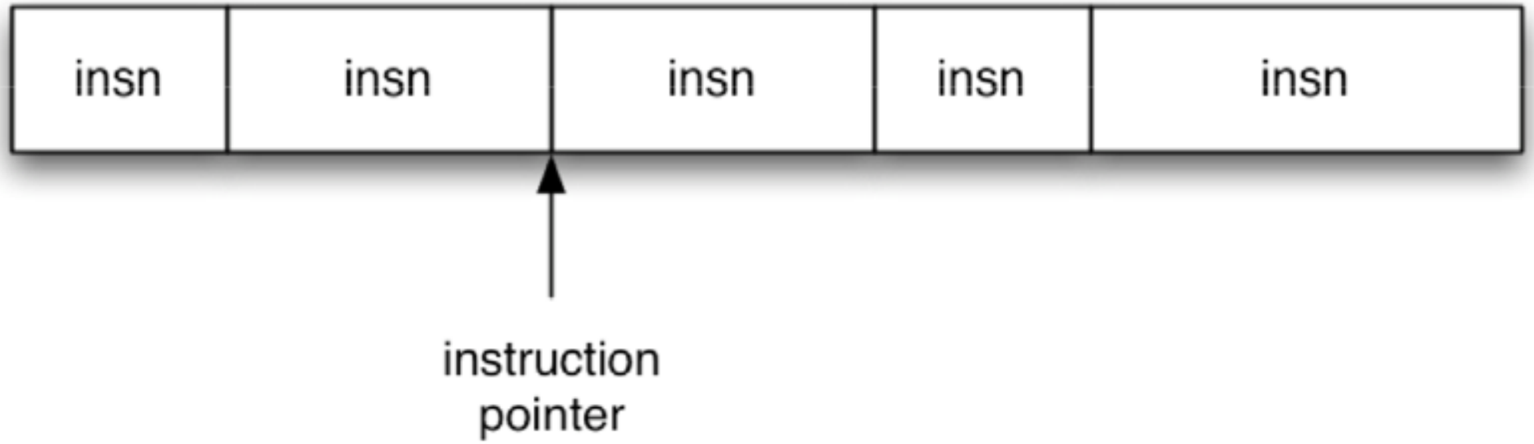
%eax = 5

%ebx = 0x8048000

**Memory**

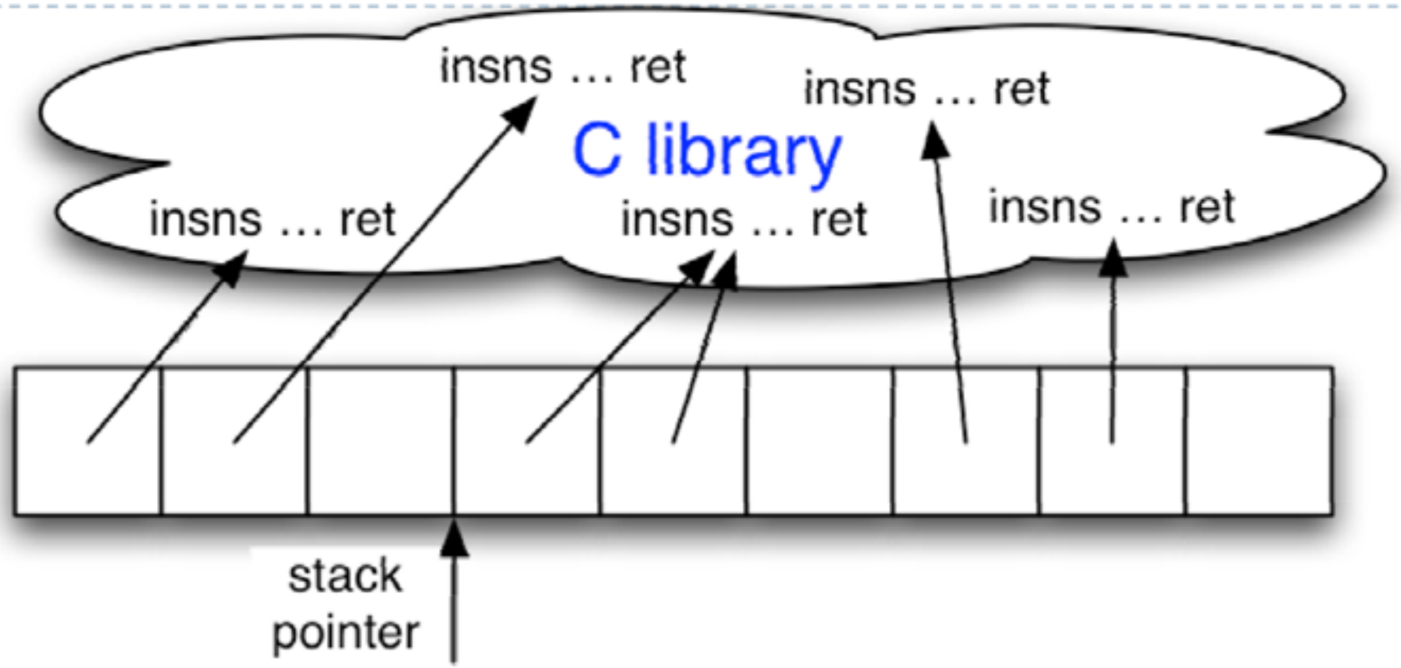
0x8048000 = 5

# Machine Instructions



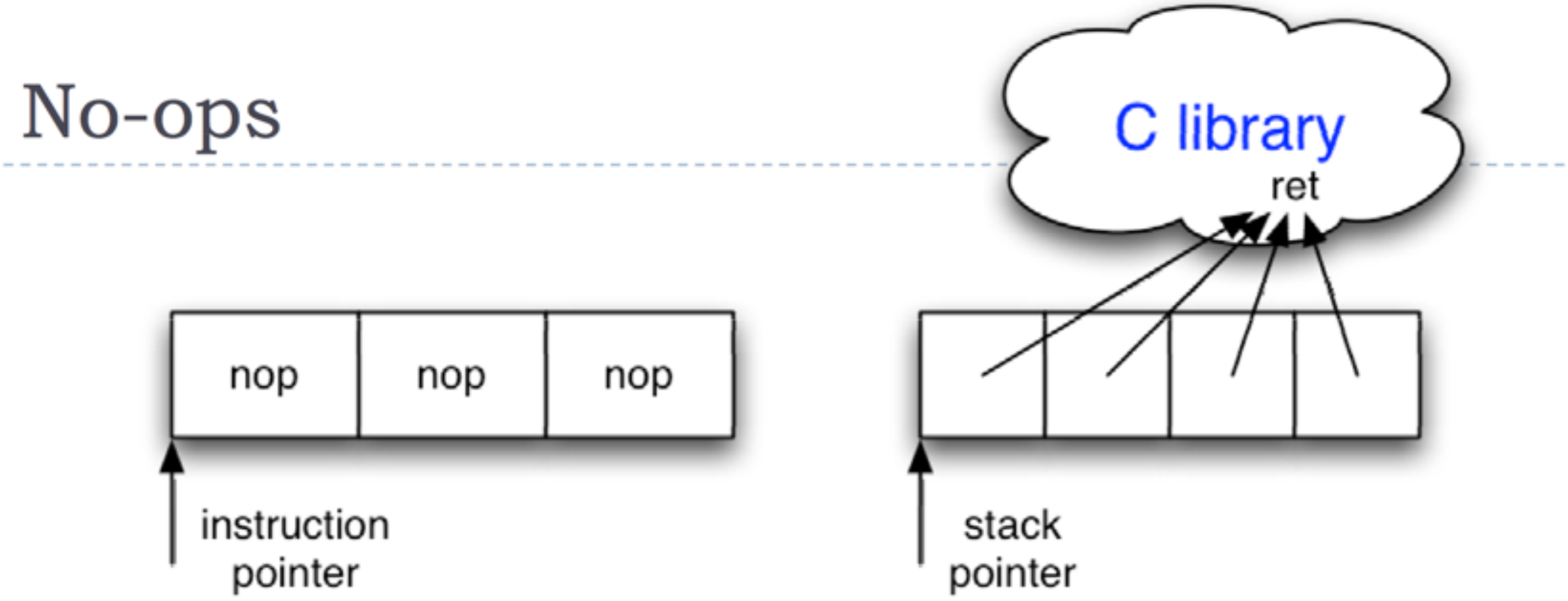

- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip

# 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

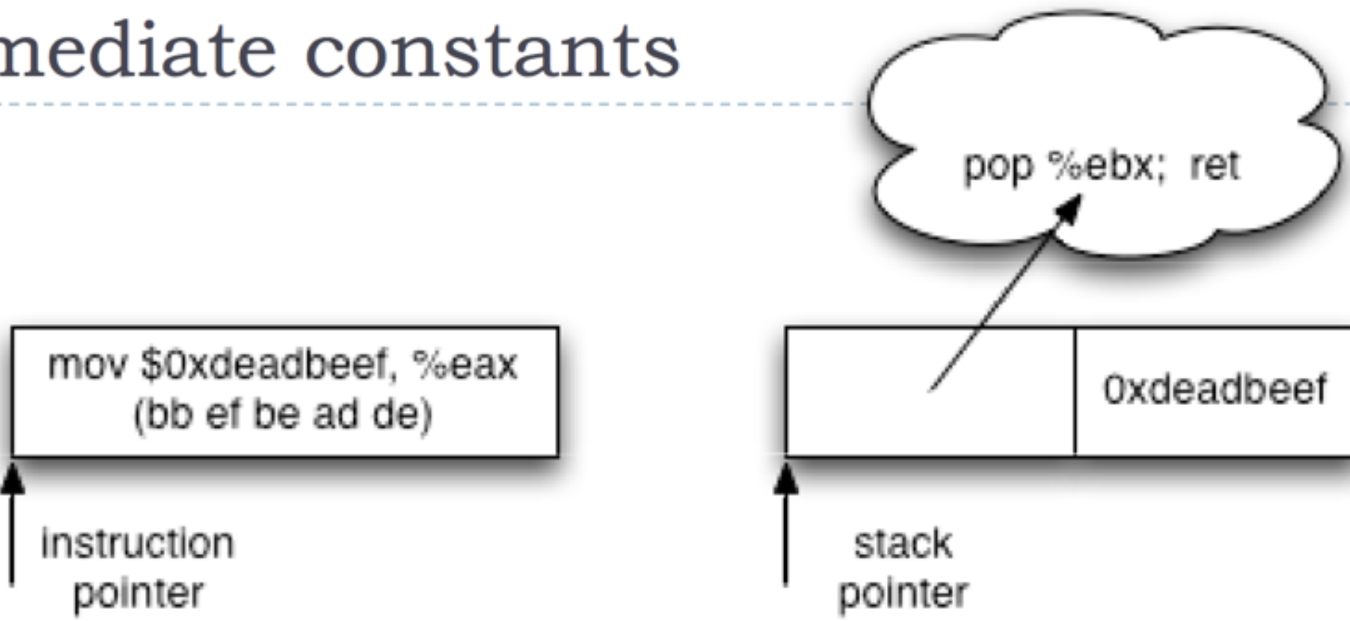
# Building ROP Functionality




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

# Building ROP Functionality

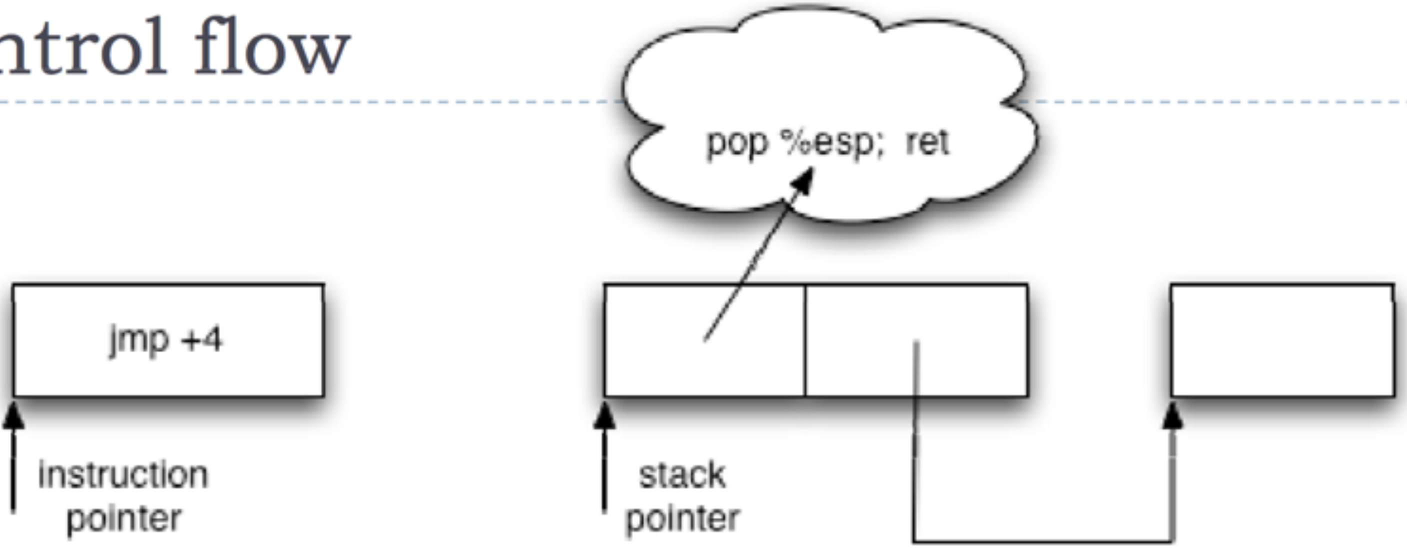
## Immediate constants




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

# Building ROP Functionality

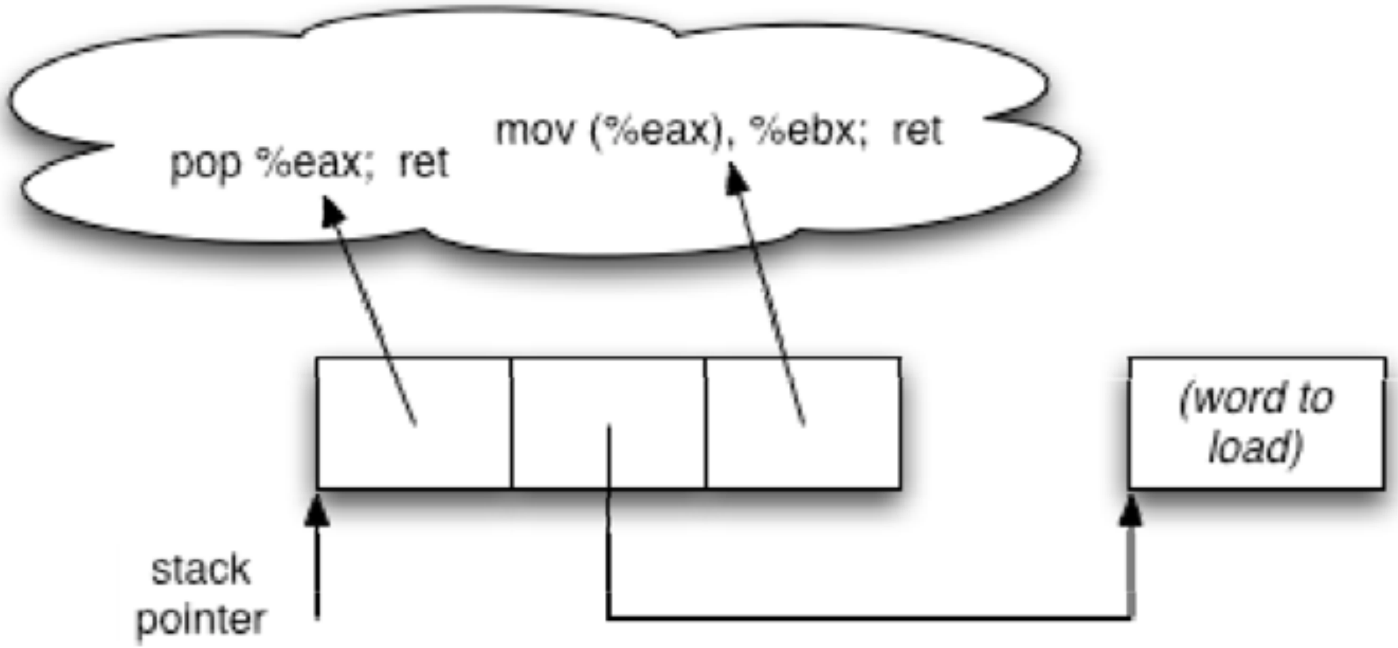
## Control flow




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

# Creating Programs

## **Gadgets**: multiple instruction sequences




- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register:
  - Load address of source word into %eax
  - Load memory at (%eax) into %ebx

# Finding Gadgets

## Finding instruction sequences

- Any instruction sequence ending in "ret" is useful — could be part of a gadget

- **Algorithmic problem**: recover all sequences of valid instructions from libc that end in a "ret" insn
- Idea: at each ret (c3 byte) look back:
  - are preceding *i* bytes a valid length-*i* insn?
  - recursefrom found instructions
- Collect instruction sequences in a trie

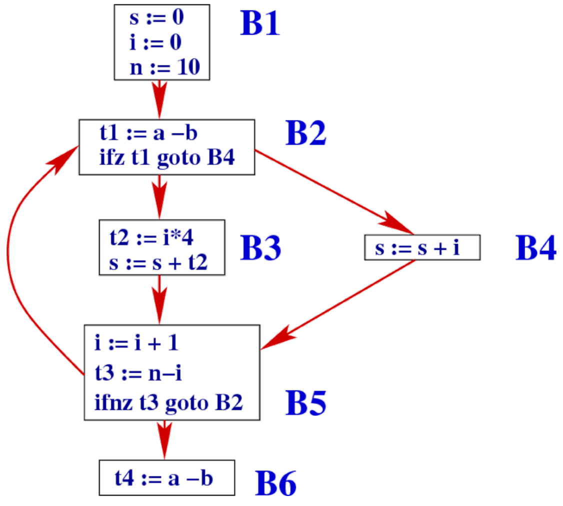
# ROP Conclusions

## Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish "good code" from "bad code" are useless
- Return-oriented programming likely possible on *every* architecture, not just x86
- Compilers make sophisticated return-oriented exploits easy to write

# Control-Flow Integrity

- Goal: Ensure that process control follows source code
  - Adversary can only choose authorized control-flow sequences
- *Build* a model from source code that describes legal control flows
  - E.g., control-flow graph
- *Enforce* the model on program execution
  - Instrument indirect control transfers
    - Jumps, calls, returns, ...
- Challenges
  - Building accurate model
  - Efficient enforcement

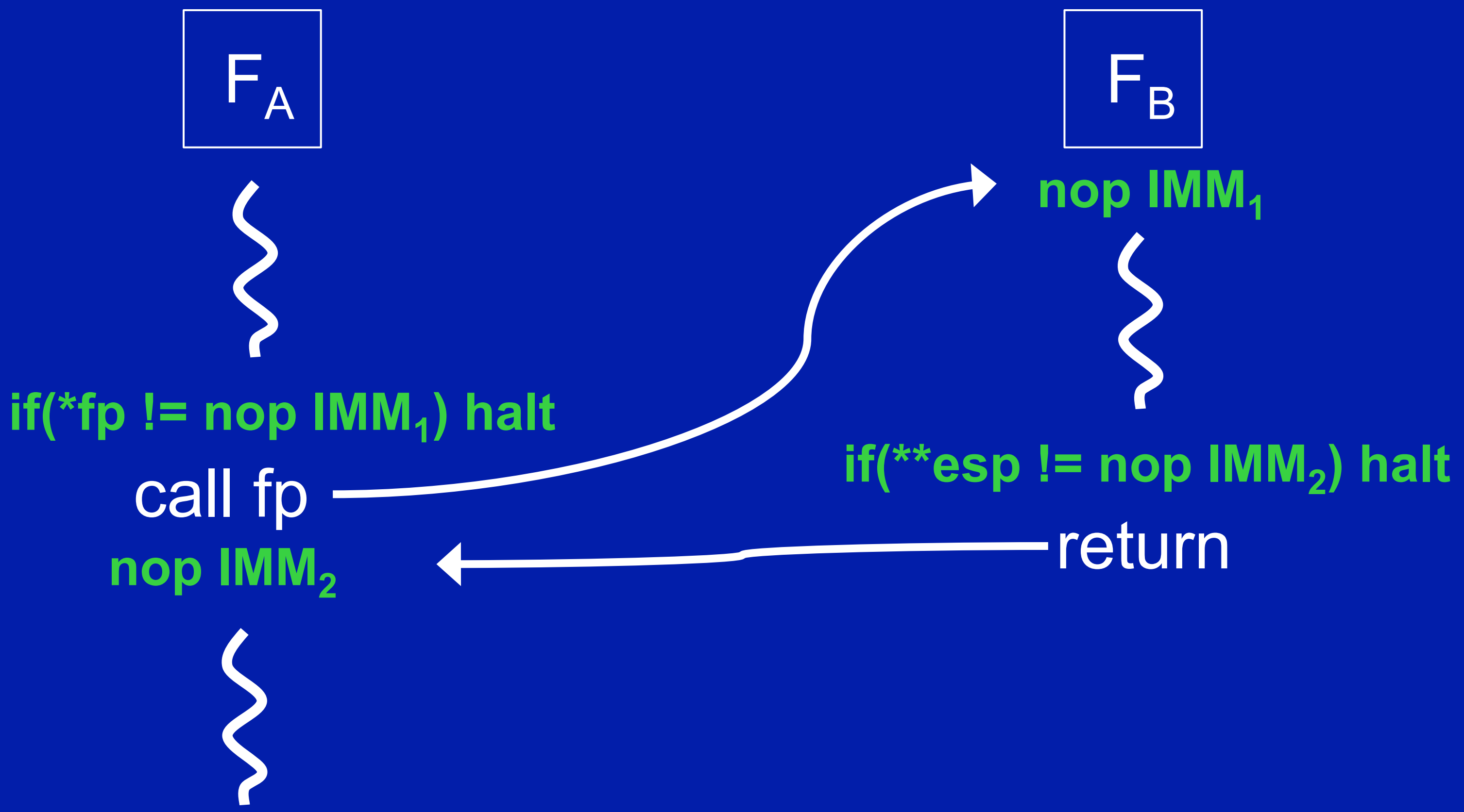


# Software Control Flow Integrity

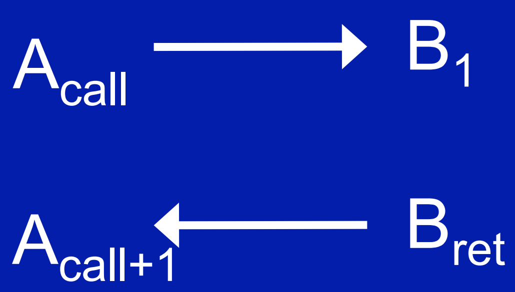
## Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with:
Úlfar Erlingsson and Martín Abadi

# Our Mechanism



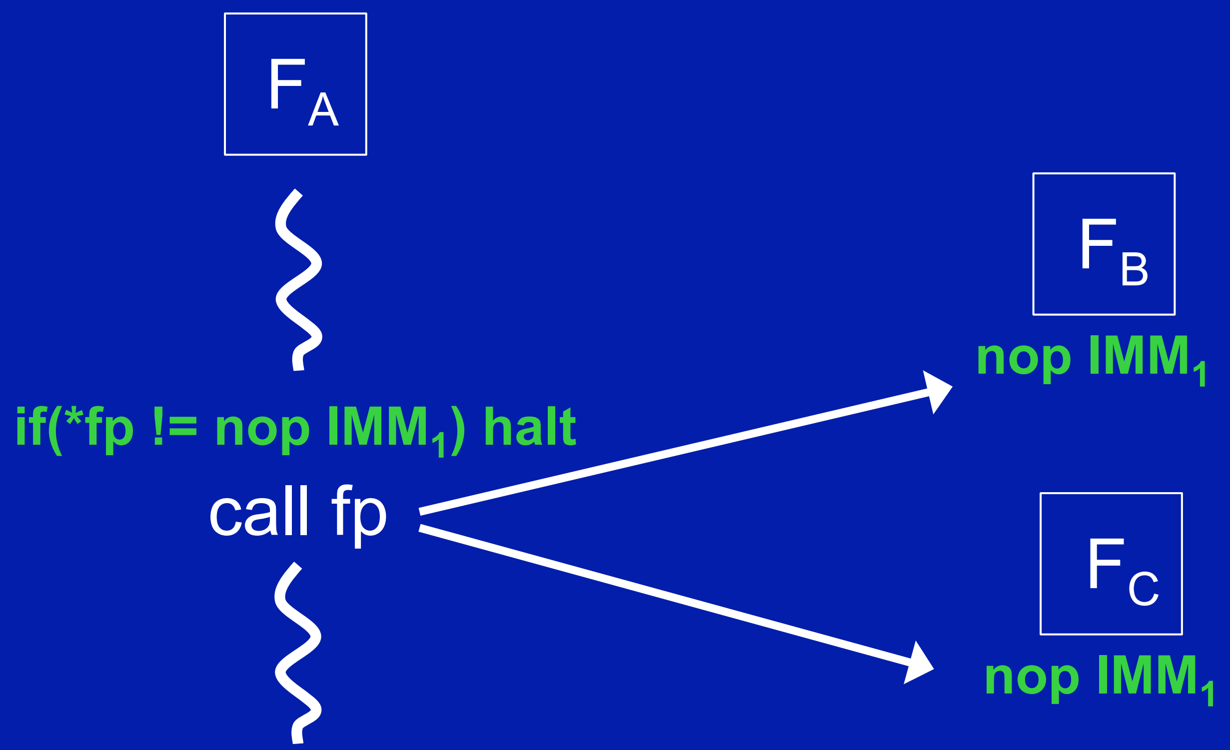

CFG excerpt



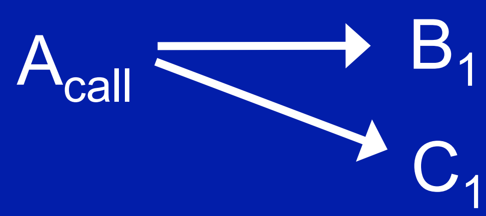

NB: Need to ensure bit patterns for nops appear nowhere else in code memory

# More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int → int function




CFG excerpt



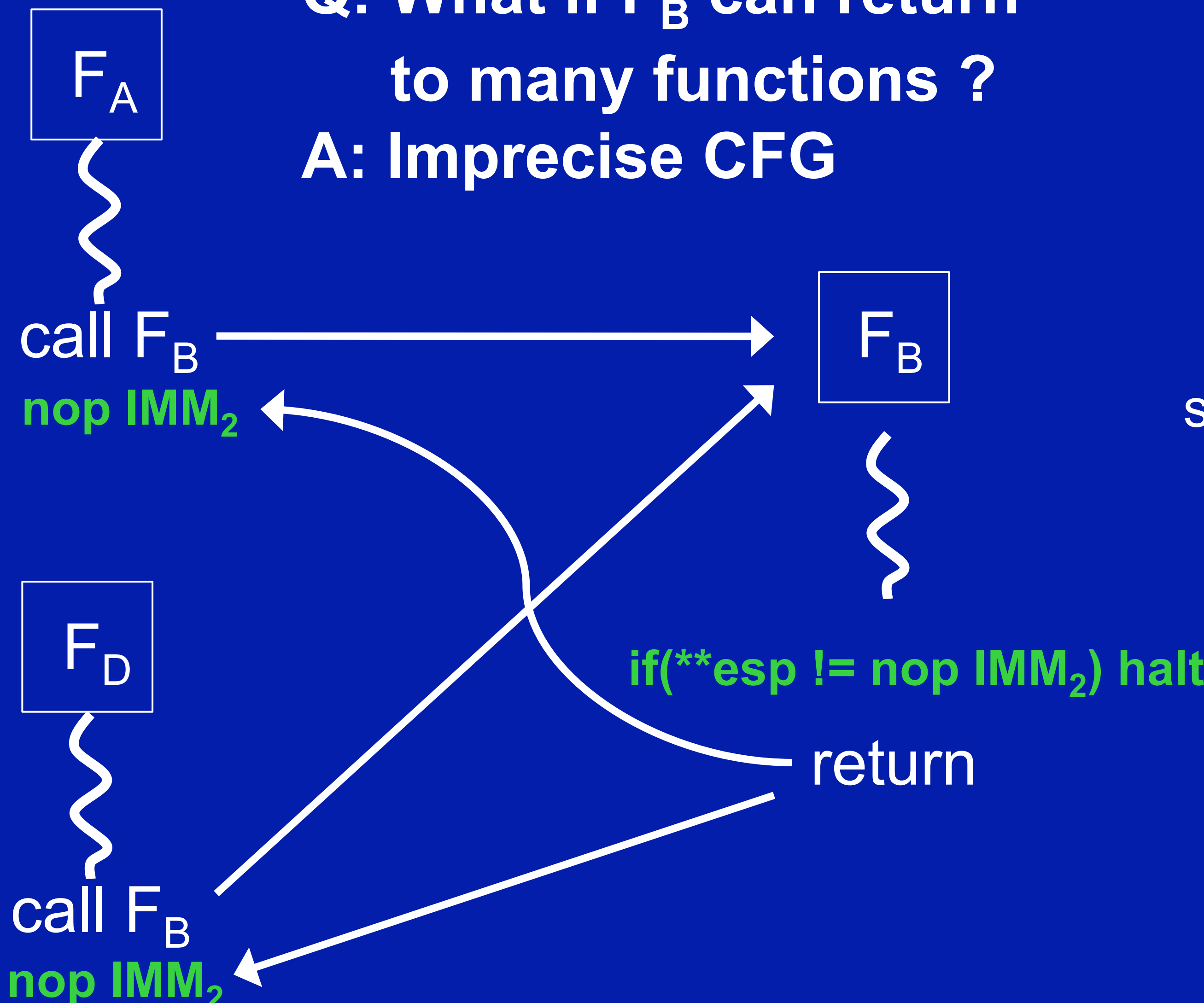

$succ(A_{call}) = \{B_1, C_1\}$

**Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction**

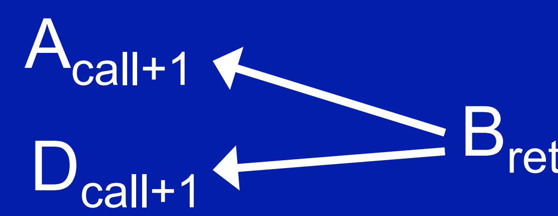
# Imprecise Return Information

**Q: What if $F_B$ can return to many functions ?**

**A: Imprecise CFG**




CFG excerpt



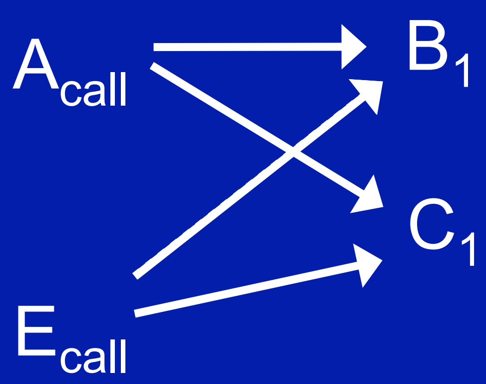

$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$

**CFG Integrity:** Changes to the PC are only to valid successor PCs, per succ().

# No “Zig-Zag” Imprecision

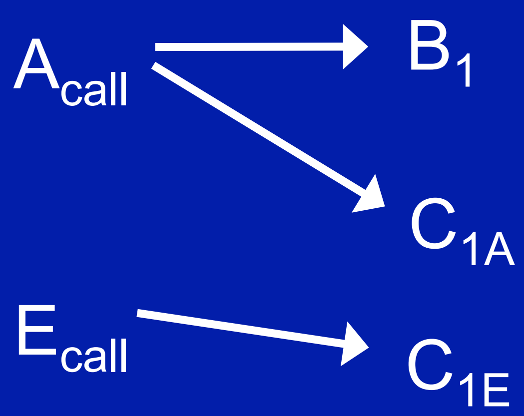
Solution I: Allow the imprecision

CFG excerpt




Solution II: Duplicate code to remove zig-zags

CFG excerpt



# CFG Imprecision

- Best reduced by a technique developed in the "HyperSafe" system
  - "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity" IEEE Symposium on Security and Privacy, 2010
- On indirect call (forward edge)
  - Check the proposed target against the set of legal targets from the CFG
- On return (backward edge)
  - Check the proposed return location against the set of legal return locations from the CFG
- Tricky to make that efficient (see the paper)

# Shadow Stack

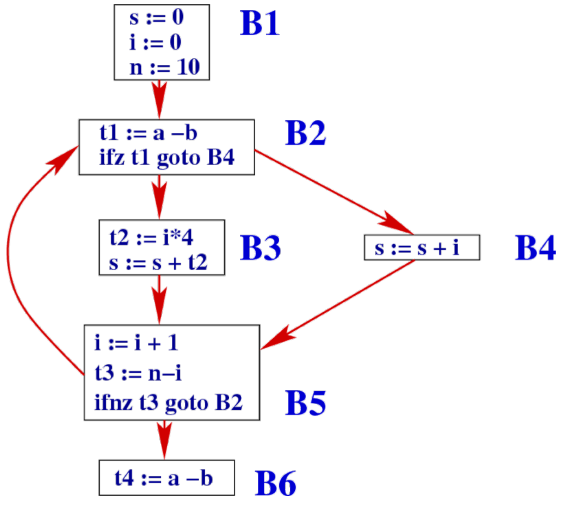
- What should be the target of a return instruction?
  - Return to caller
  - But, need a way to protect return value
- Shadow stack
  - Stack that can only be accessed by trusted code (e.g., software fault isolation)
  - Off limits to overflows

# CFG Computation

- What should be the target of a call instruction?
  - Direct call - hard coded, so no problem
  - Indirect call (function pointer) - would be any legal value for the function pointer
    - That is, anywhere it can point
    - The "points-to" problem in general, which is undecidable
- So, there are various techniques to over-approximate the target set for each indirect call

# More Challenges

- Predicting return targets can be hard
  - Exceptions, signals, and setjmp/longjmp
- Runtime generation of indirect jumps
  - E.g., dynamically linked libraries
- Indirect jumps using arithmetic operators
  - E.g., assembly
- Is enforcing fine-grained CFI sufficient to prevent exploits?



# Recent Result

- Suppose a program is protected by fine-grained CFG on calls and a shadow stack on returns
- Further suppose that the program contains an "arbitrary write primitive" (e.g., based on a memory error)
- For these programs, exploits can be generated over 80% of the time, even against CFI defenses
  - "Block Oriented Programming: Automating Data-Only Attacks", ACM CCS 2018
- Exploits follow CFG, but manipulate memory to complete exploit
  - Called "data-oriented programming"

# Alternatives to CFI?

- What are the fundamental enablers of ROP attacks?
  - (1) CFI: violate control flow
  - (2) Adversary can choose gadgets
- Can we prevent adversaries from choosing useful gadgets?
  - In general, adversaries can create/obtain the same binary as is run by the victim
  - But, that need not be the case

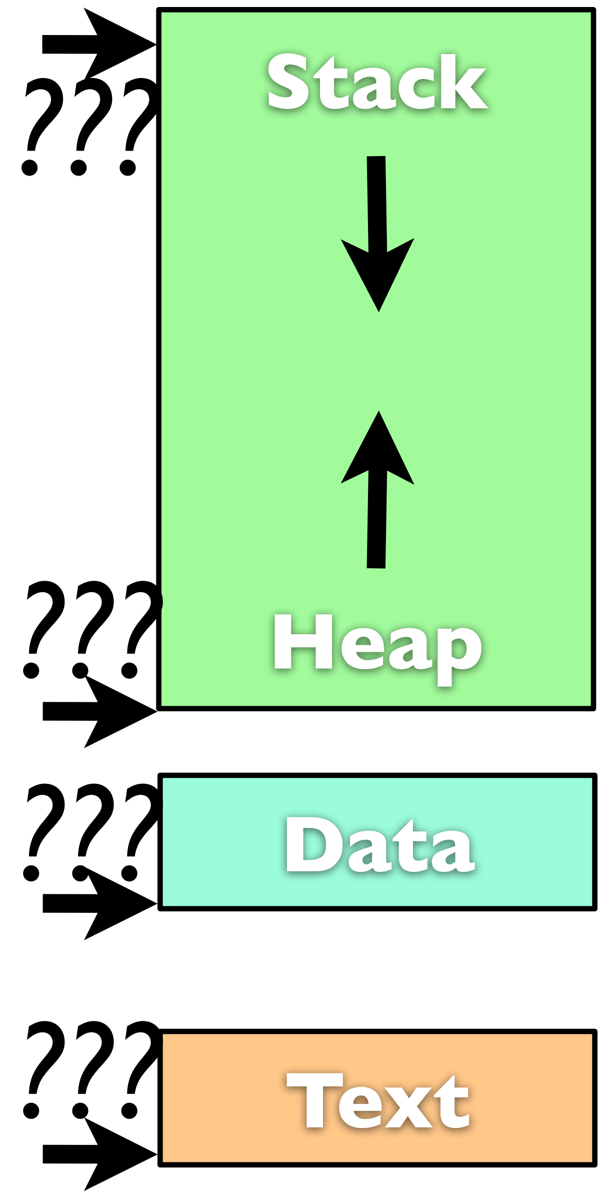
# Apply Crypto to Code?

PennState

- Can we randomize the program's execution in such a way that an adversary cannot select gadgets?
- Given a secret key and a program address space, encrypt the address space such that
  - the probability that an adversary can locate a particular instruction (start of gadget) is sufficiently low
  - and the program still runs correctly and efficiently
- Called address space randomization

# ASLR

- For control-flow attacks, attacker needs absolute addresses
- Address-space Layout Randomization (ASLR) randomizes base addresses of memory segments on each invocation of the program
  - Attacker cannot predict absolute addresses
- Heap, stack, data, text, mmap, ...



# ASLR Implementations

- Linux
  - Introduced in Linux 2.6.12 (June 2005)
  - Shacham et al. [2004]:16 bits of randomization defeated by a (remote) brute force attack in minutes
  - Reality: ASLR for text segment (PIE) is rarely used
    - Only few programs in Linux use PIE
    - Enough gadgets for ROP can be found in unrandomized code [Schwartz 2011]

# ASLR Limitations

- Attacks may leak randomization information
  - Disclosure attacks
  - Use buffer over-read to read unauthorized program memory (extract code or randomizing state)
- ASLR can be bypassed by information leaks about memory layout
  - E.g., format string vulnerabilities
- So, what can we do?
  - How do we avoid leaking the "key"?

# Conclusion

- Control-flow attack defenses operate at two stages
  - Prevent attacker from getting control
    - StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  - Prevent attacker from using control for malice
    - NX, W (xor) X, ASLR, Control Flow Integrity (CFI), ...
- For maximum security, a system may need to use a combination of these defenses
- *Q. Is subverting control-flow the only goal of an attacker?*