

Return-Oriented Programming

CS 4430/7430 Compiler Construction

Announcements

- ▶ Today: Return-oriented Programming
 - ▶ Based on Hovav Shacham, et al.'s The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
 - ▶ <http://cseweb.ucsd.edu/~hovav/papers/s07.html>
 - ▶ Some diagrams are borrowed from his slides



Buffer Overflow: Causes and Cures

- ▶ Typical memory exploit involves **code injection**
 - ▶ Put malicious code at a predictable location in memory, usually masquerading as data
 - ▶ Trick vulnerable program into passing control to it
 - ▶ Overwrite saved EIP, function callback pointer, etc.
- ▶ Idea: **prevent execution of untrusted code**
 - ▶ Make stack and other data areas non-executable
 - ▶ Note: messes up useful functionality (e.g., ActionScript)
 - ▶ Digitally sign all code
 - ▶ Ensure that all control transfers are into a trusted, approved code image



Buffer Overflow Style Attacks

Prime
Target
for SW
Attacks

edx
[foo]
ebx
eax
inst. after call
baz

For caller
stack frame

k

j

i

esi

Caller-saved reg.

Last parameter

Second parameter

First parameter

Return address

Saved EBP

Local var 1

Local var 2

Local var 3

Callee-saved reg.

EBP

EBP - 4

EBP - 8

EBP - 12

ESP

If you can somehow
overwrite the return address,
you can pwn the whole system

W \oplus X / DEP

- ▶ **Mark all writeable memory locations as non-executable**
 - ▶ Example: Microsoft's DEP (Data Execution Prevention)
 - ▶ This mitigates some code injection exploits
- ▶ **Hardware support**
 - ▶ AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
 - ▶ Makes memory page non-executable
- ▶ **Widely deployed**
 - ▶ Windows (since XP SP2), Linux (via PaX patches), OpenBSD, OS X (since 10.5)



What Does $W\oplus X$ Not Prevent?

- ▶ Can still corrupt stack ...
 - ▶ ... or function pointers or critical data on the heap, but that's not important right now
- ▶ As long as “saved EIP” points into existing code, $W\oplus X$ protection will not block control transfer
- ▶ This is the basis of **return-to-libc** exploits
 - ▶ Overwrite saved EIP with address of any library routine, arrange memory to look like arguments
- ▶ Does not look like a huge threat
 - ▶ Attacker cannot execute arbitrary code
 - ▶ ... especially if `system()` is not available



return-into-libc attacks

▶ Idea

- ▶ replace return address of a subroutine with that of another subroutine
- ▶ the replacement subroutine must already be in memory
- ▶ the attack does not inject code
- ▶ Therefore, NX bit feature useless

▶ Why “libc”?

▶ Countermeasure:

- ▶ Address Space Layout Randomization (ASLR)
- ▶ Alter compiler/loader to reorganize code layout (including subroutines) randomly
- ▶ I.e., your copy of the same program will have subroutines at different locations than mine



return-to-libc on Steroids

- ▶ Overwritten saved EIP need not point to the **beginning** of a library routine
- ▶ Any existing instruction in the code image is fine
 - ▶ Will execute the sequence starting from this instruction
- ▶ What if instruction sequence contains RET?
 - ▶ Execution will be transferred... to where?
 - ▶ Read the word pointed to by stack pointer (ESP)
 - ▶ Guess what? Its value is under attacker's control! (why?)
 - ▶ Use it as the new value for EIP
 - ▶ Now control is transferred to an address of attacker's choice!
 - ▶ Increment ESP to point to the next word on the stack

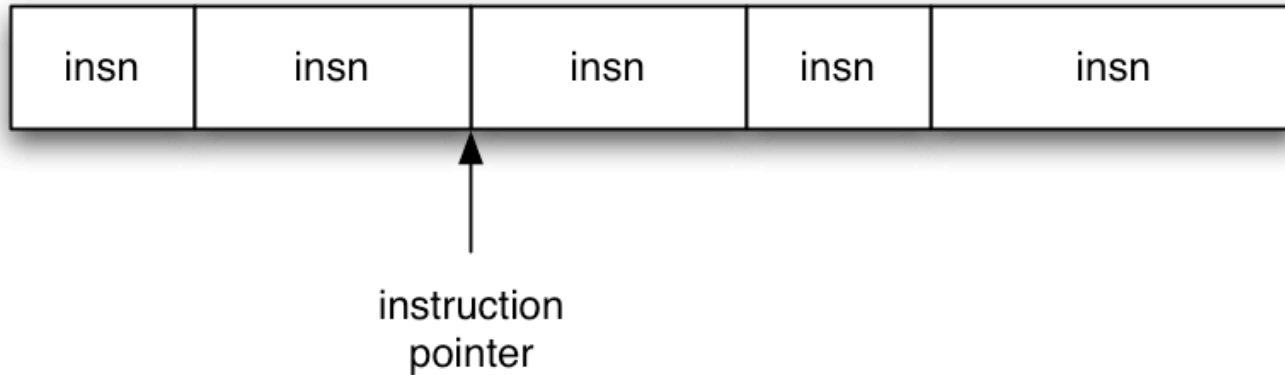


Chaining RETs for Fun and Profit

- ▶ Can chain together sequences ending in RET
 - ▶ Krahmer, “x86-64 buffer overflow exploits and the **borrowed code chunks exploitation** technique” (2005)
- ▶ What is this good for?
- ▶ Answer [Shacham et al.]: **everything**
 - ▶ Turing-complete language
 - ▶ Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
 - ▶ Attack can perform arbitrary computation using no injected code at all!

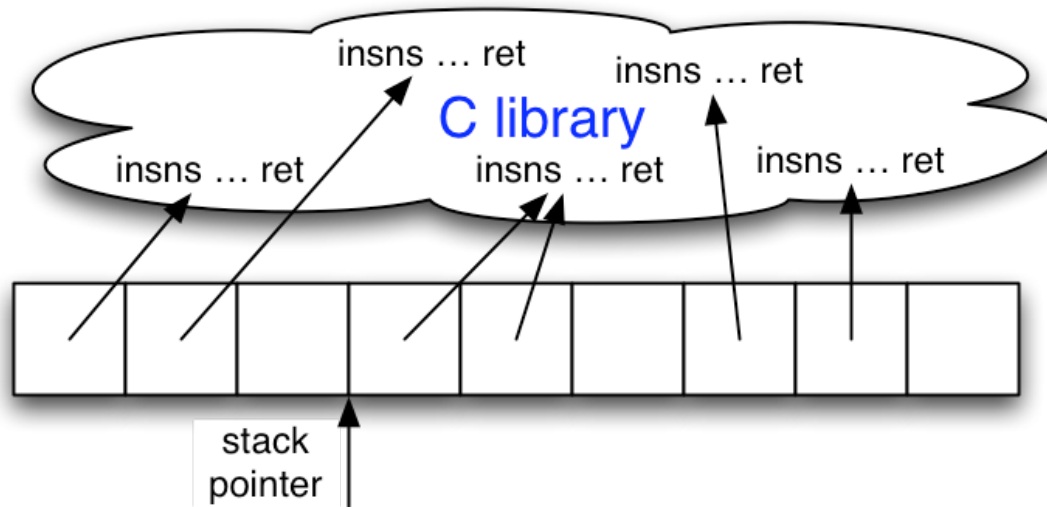


Ordinary Programming



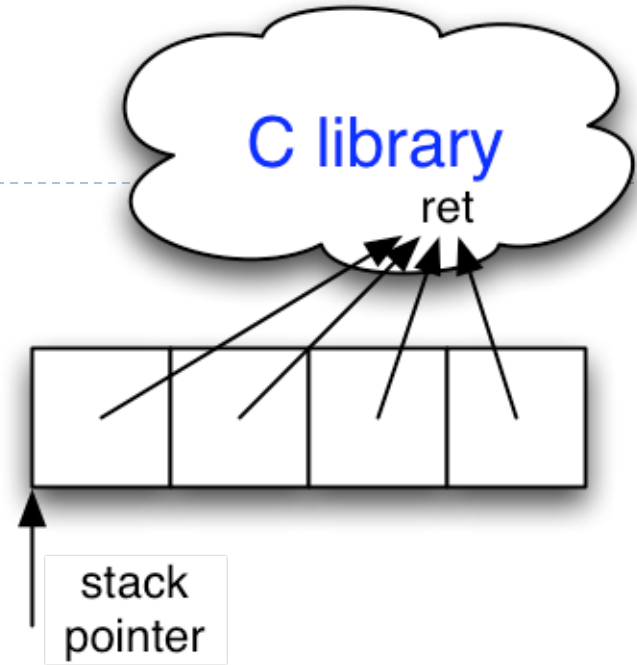
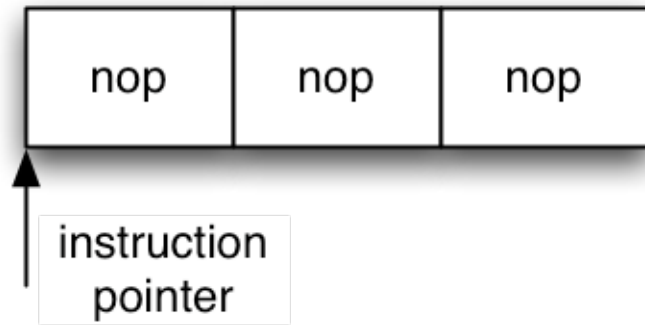
- ▶ Instruction pointer (EIP) determines which instruction to fetch and execute
- ▶ Once processor has executed the instruction, it automatically increments EIP to next instruction
- ▶ Control flow by changing value of EIP

Return-Oriented Programming



- ▶ **Stack pointer** (ESP) determines which instruction sequence to fetch and execute
- ▶ Processor doesn't automatically increment ESP
 - ▶ But the RET at end of each instruction sequence does

No-ops



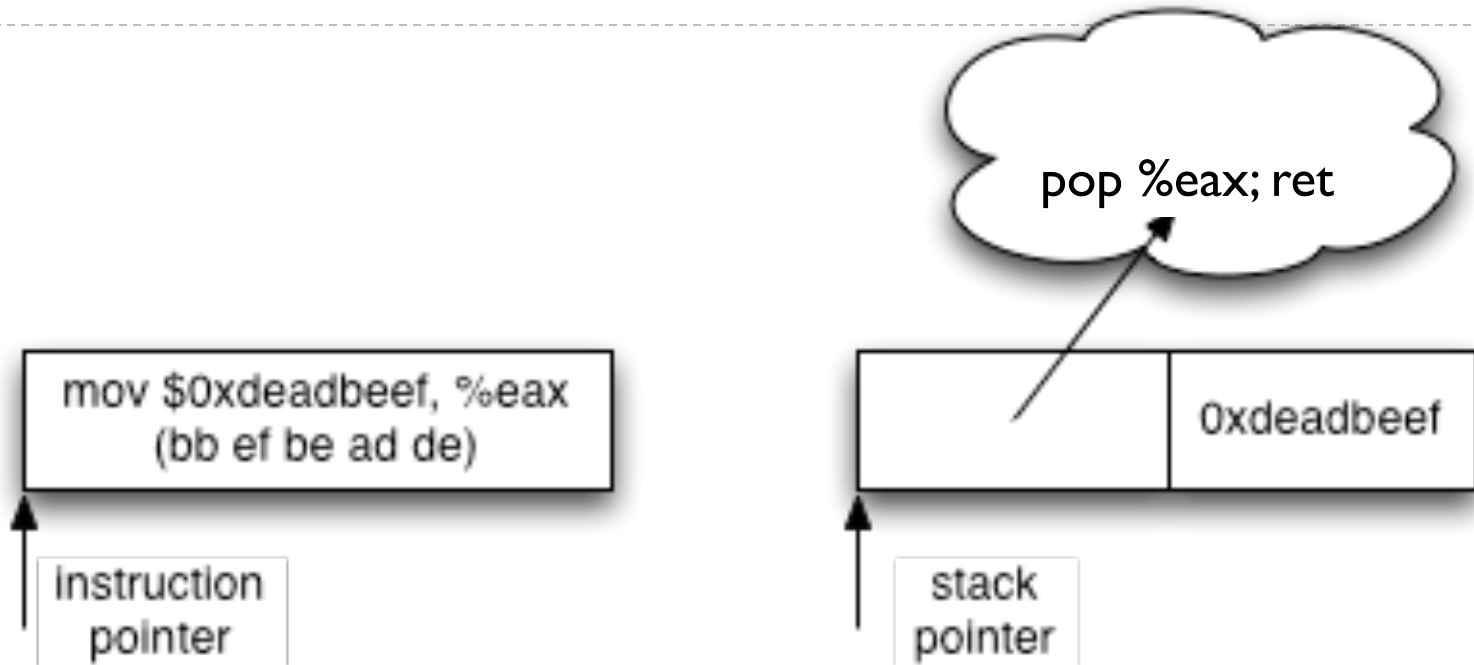
- ▶ No-op instruction does nothing but advance EIP
- ▶ Return-oriented equivalent
 - ▶ Point to return instruction
 - ▶ Advances ESP
- ▶ Useful in a NOP sled (what's that?)

What's a NOP Sled?

- ▶ Determining the correct offset for injecting code is not easy;
- ▶ NOP (non operation) sled can be used to increase the number of potential offsets;
- ▶ Generally, we can fill in the beginning of shellcode with NOPs.
- ▶ The opcode for NOP is 0x90
- ▶ EX: `shellcode[]="\x90\x90\x90\x31\xdb\xb0\x01\xcd\x80"`

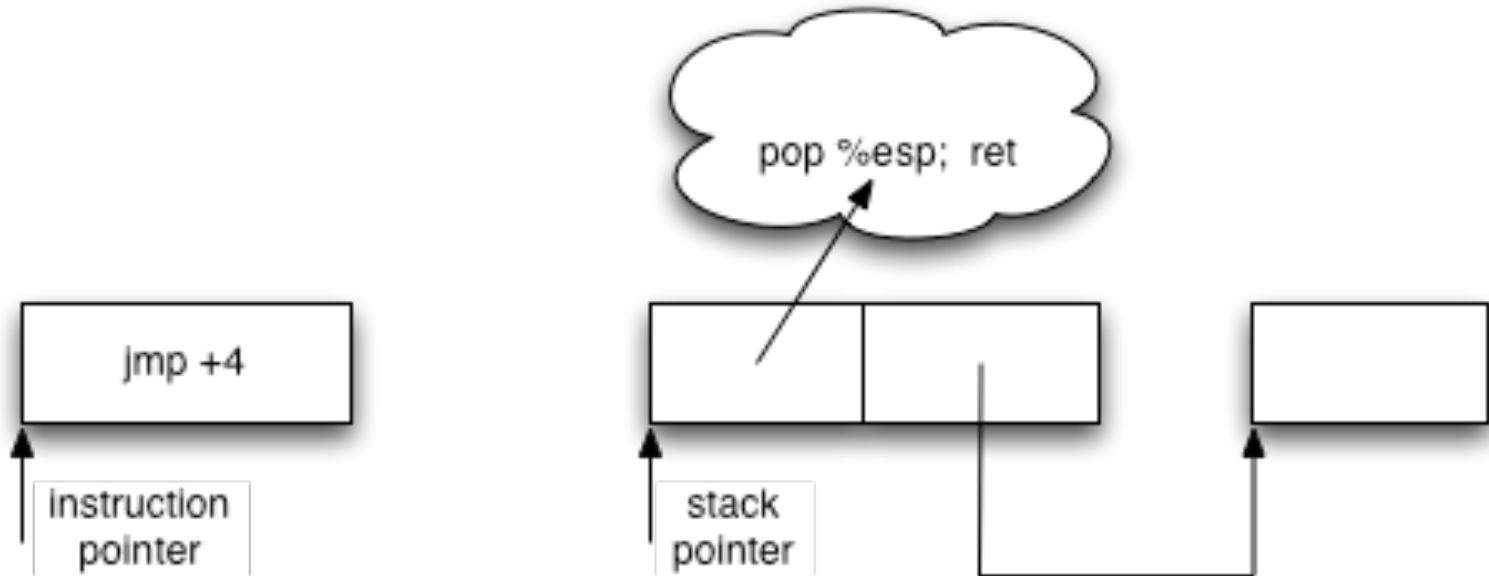


Immediate Constants



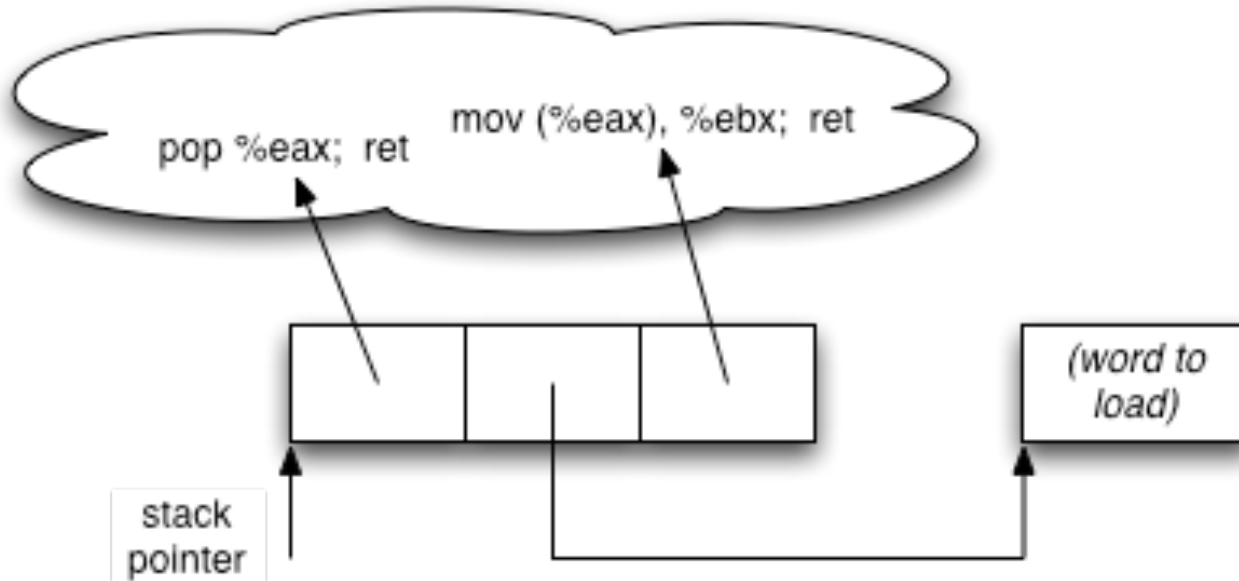
- ▶ Instructions can encode constants
- ▶ Return-oriented equivalent
 - ▶ Store on the stack
 - ▶ Pop into register to use

Control Flow



- ◆ Ordinary programming
(Conditionally) set EIP to new value
- ◆ Return-oriented equivalent
(Conditionally) set ESP to new value

Gadgets: Multi-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

Gadget Design

- ▶ Testbed: libc-2.3.5.so, Fedora Core 4
- ▶ Gadgets built from found code sequences:
 - ▶ Load-store, arithmetic & logic, control flow, syscalls
- ▶ “Found” code sequences are challenging to use!
 - ▶ Short; perform a small unit of work
 - ▶ No standard function prologue/epilogue
 - ▶ Haphazard interface, not an ABI (Application Binary Interface)
 - ▶ Some convenient instructions not always available



A Warning to the Curious

- ▶ One of the challenges of reading the gadget implementations arises from the fact that gadgets are **found** code sequences
 - ▶ i.e., you have to make do with the code you find
- ▶ As a consequence, there may be instructions in a gadget that are "useful" and some that are "coincidental"

```
addl (%eax), %esp ; want %esp := %esp+(%eax)
addb %cl, 0(%eax) ; don't care
ret
```



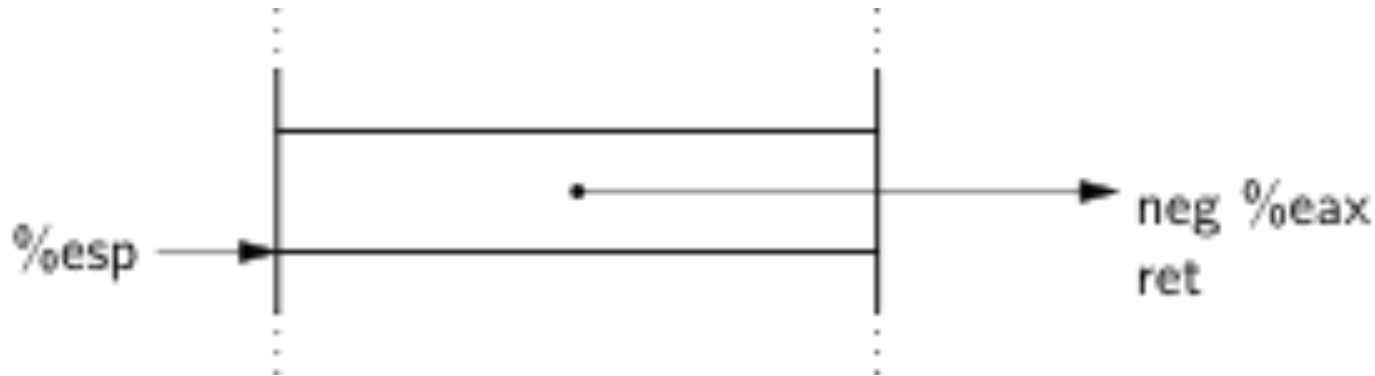
Conditional Jumps*

- ▶ **cmp** compares operands and sets a number of flags in the EFLAGS register
 - ▶ Luckily, many other ops set EFLAGS as a side effect
- ▶ **jcc** jumps when flags satisfy certain conditions
 - ▶ But this causes a change in EIP... not useful (why?)
- ▶ Need conditional change in stack pointer (ESP)
- ▶ Strategy:
 - ▶ Move flags to general-purpose register
 - ▶ Compute either delta (if flag is 1) or 0 (if flag is 0)
 - ▶ Perturb ESP by the computed delta

* Intricate – talk more about it Wednesday.

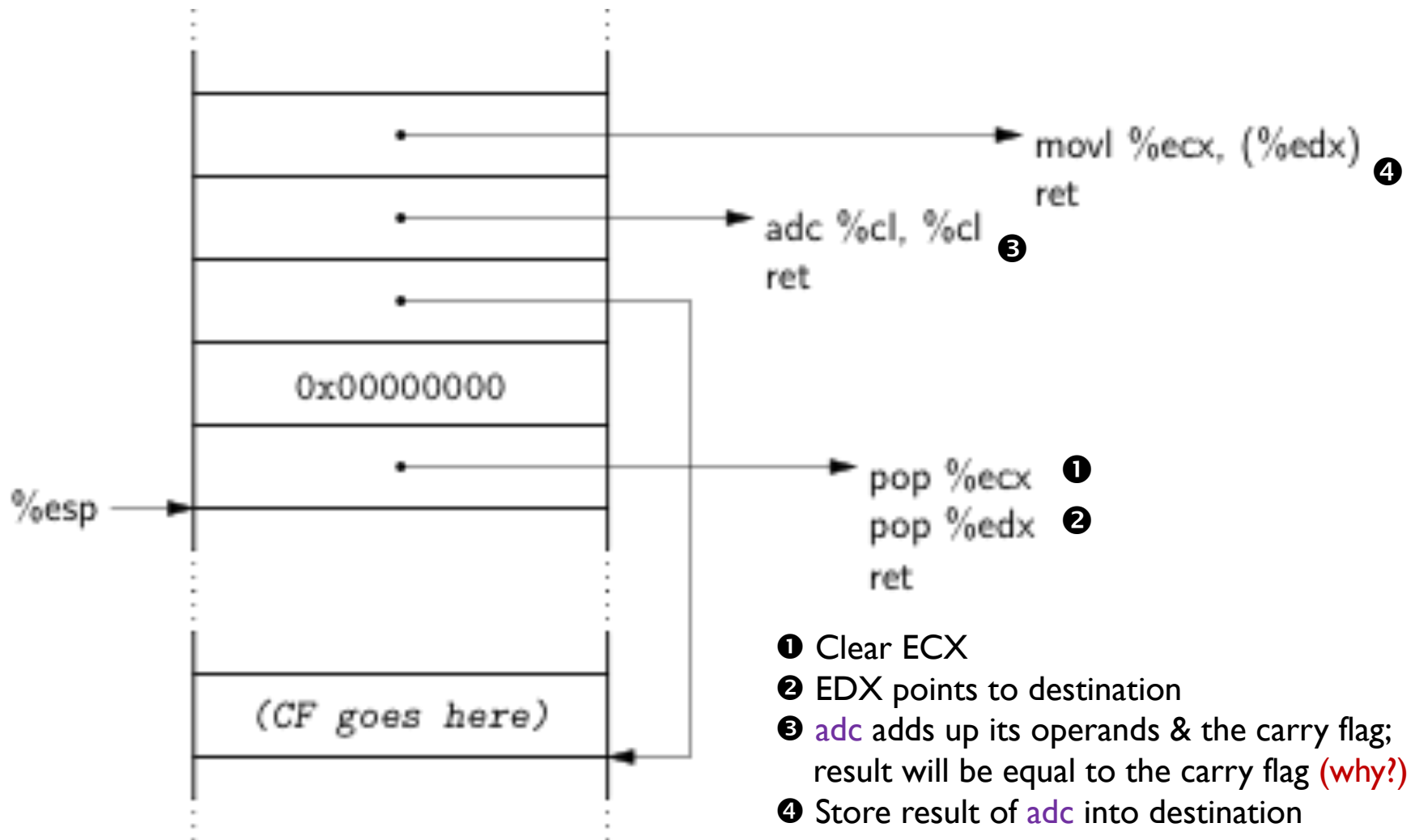


Phase 1: Perform Comparison

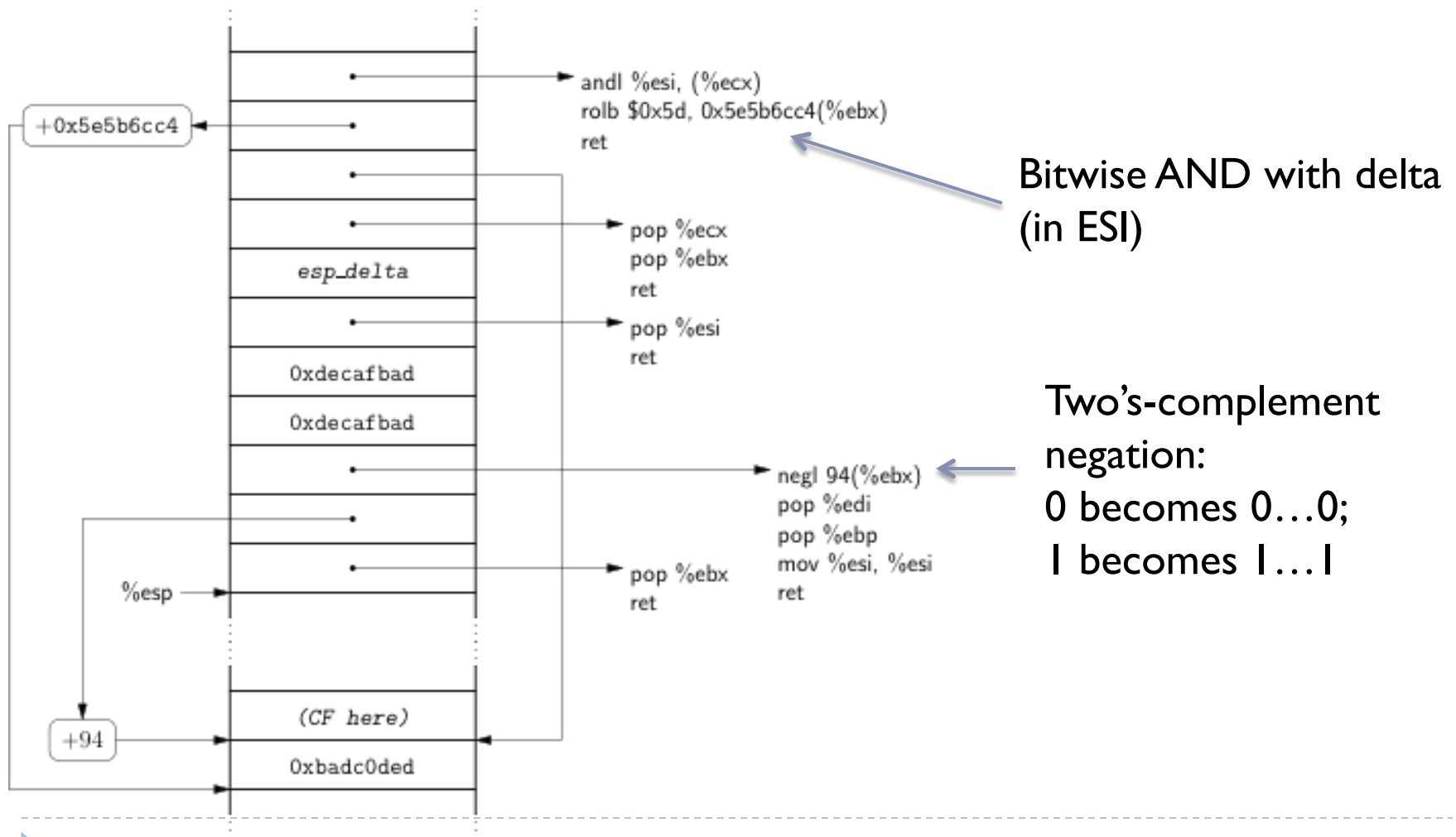


- ▶ **neg** calculates two's complement
 - ▶ Replaces the value of operand with its two's complement – equivalent to subtracting the operand from 0.)
 - ▶ As a side effect, **sets carry flag (CF)** if the argument is nonzero
- ▶ Use this to test for equality
- ▶ **sub** is similar, use to test if one number is greater than another

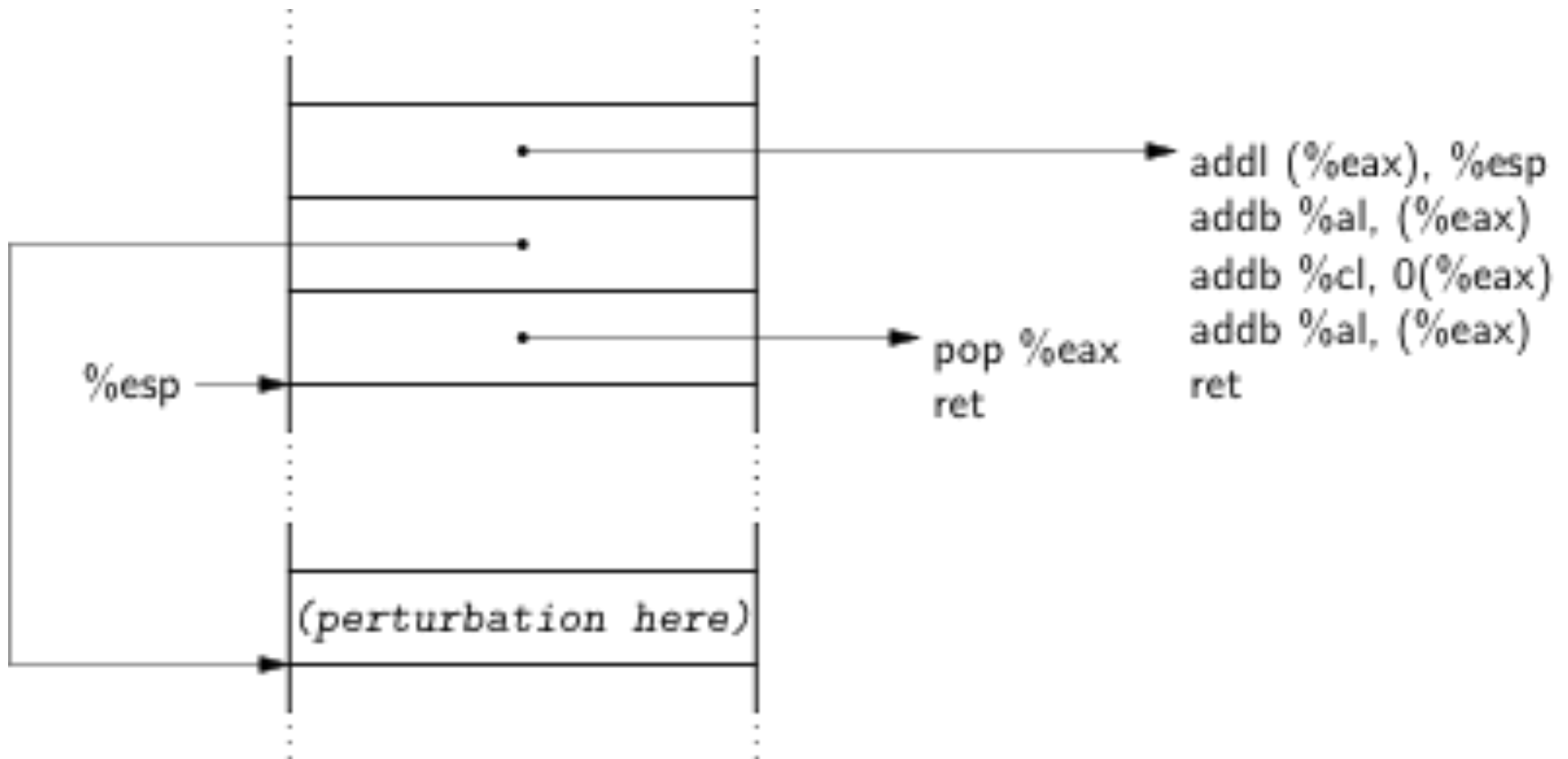
Phase 2: Store 1-or-0 to Memory



Phase 3: Compute Delta-or-Zero

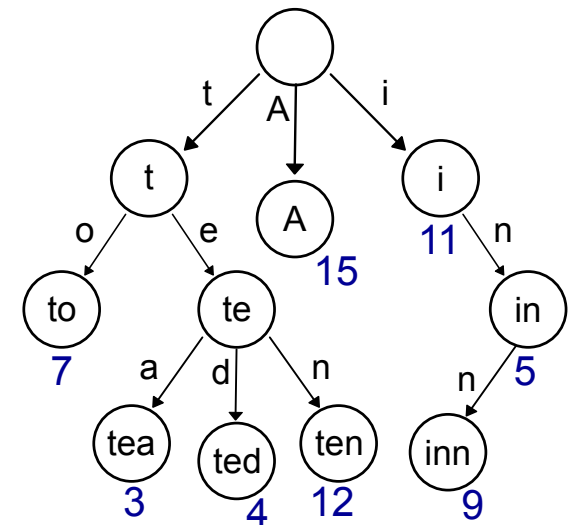


Phase 4: Perturb ESP by Delta



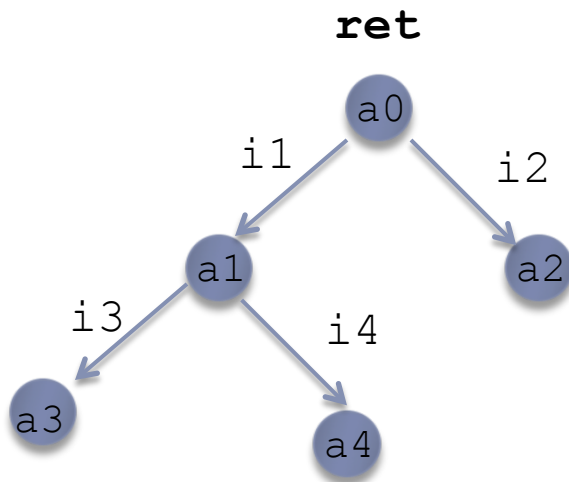
Finding Instruction Sequences

- ▶ Any instruction sequence ending in RET is useful
- ▶ Algorithmic problem: recover all sequences of valid instructions from libc that end in a RET
- ▶ At each RET (C3 byte), look back:
 - ▶ Are preceding i bytes a valid instruction?
 - ▶ Recur from found instructions
- ▶ Collect instruction sequences in a **trie**



A Gadget Trie

This trie collects the following gadgets where the a's are addresses and the i's are instructions:



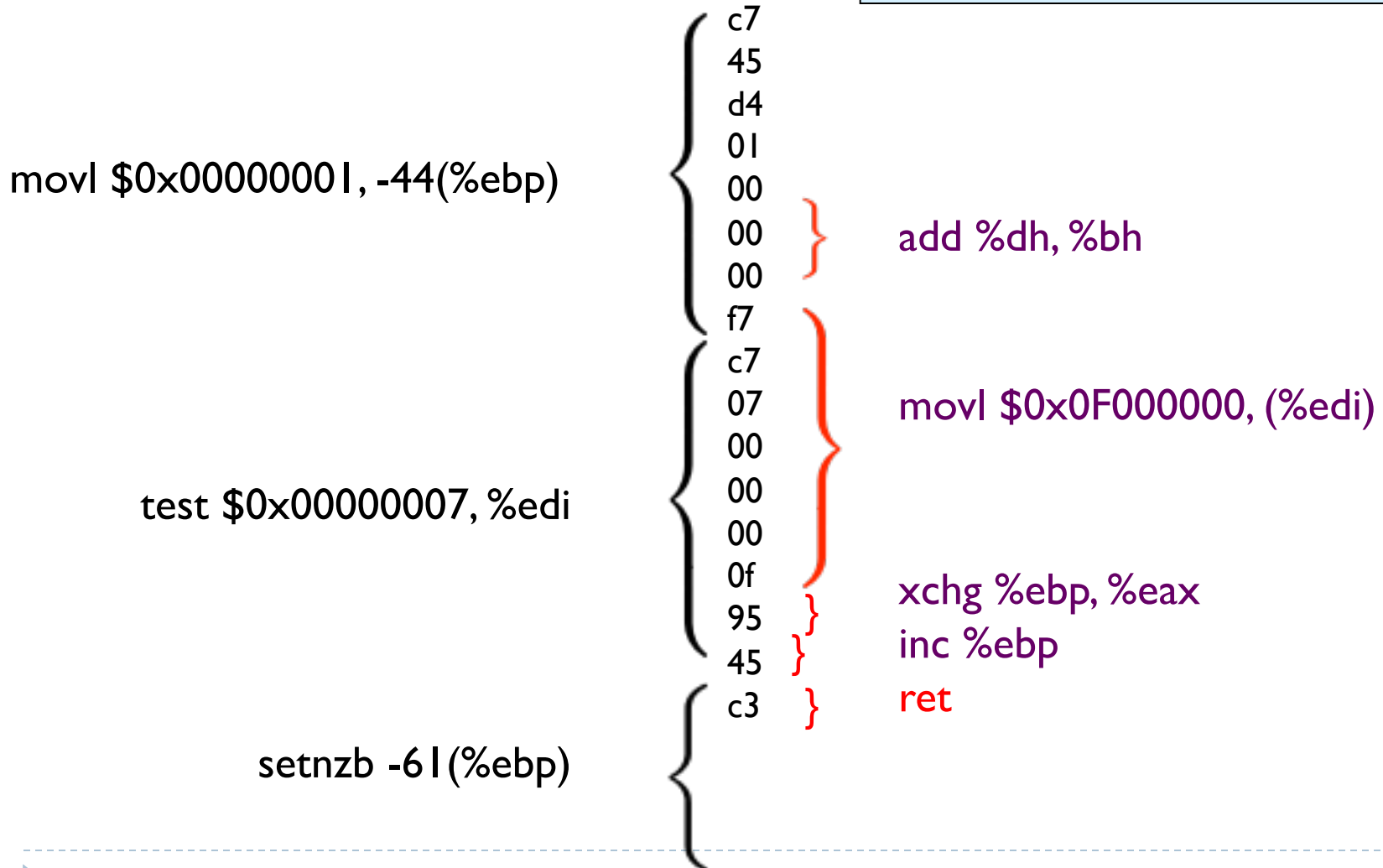
```
a3:    i3 ; i1 ; ret
a1:    i1 ; ret
a4:    i4 ; i1 ; ret
a2:    i2 ; ret
```

“Foriest” of Tries: there’s one of these tries calculated for each return found in binary.



Unintended Instructions

Actual code from ecb_crypt()



x86 Architecture Helps

- ▶ **Register-memory machine**
 - ▶ Plentiful opportunities for accessing memory
- ▶ **Register-starved**
 - ▶ Multiple sequences likely to operate on same register
- ▶ **Instructions are variable-length, unaligned**
 - ▶ More instruction sequences exist in libc
 - ▶ Instruction types not issued by compiler may be available
- ▶ **Unstructured call/ret ABI**
 - ▶ Any sequence ending in a return is useful



SPARC: the Unix86

- ▶ Load-store RISC machine
 - ▶ Only a few special instructions access memory
- ▶ Register-rich
 - ▶ 128 registers; 32 available to any given function
- ▶ All instructions 32 bits long; alignment enforced
 - ▶ No unintended instructions
- ▶ Highly structured calling convention
 - ▶ Register windows
 - ▶ Stack frames have specific format



ROP on SPARC

- ▶ Testbed: Solaris 10 libc (1.3 MB)
 - ▶ Use instruction sequences that are suffixes of real functions
 - ▶ Dataflow within a gadget
 - ▶ Structured dataflow to dovetail with calling convention
 - ▶ Dataflow between gadgets
 - ▶ Each gadget is memory-memory
 - ▶ Turing-complete computation!
 - ▶ Interesting “When Good Instructions Go Bad: Generalizing ROP to RISC” for details (same authors)
 - ▶ Also interesting: “Escape from R.O.P.: ROP w/o Returns”
-

