Assembly Crash Course

Control Flow

Yan Shoshitaishvili
Arizona State University

Computers Make Decisions

```
if (authenticated) {
    leetness = 1337;
}
else {
    leetness = 0;
}
```

So far, we've just shunted data around.

But how do we make decisions?

What to Execute?

First, let's look at how computers execute instructions.

Recall: Assembly instructions are direct translations of binary code.

This binary code lives in *memory*.

Example:

	0x400800			
Program Binary Code	`pop rax`	`pop rbx`	`add rax, rbx`	`push rax`	

This is (in hex):

	0x400800	0x400801	0x400802	0x400805	
Program Binary Code	`58`	`5b`	`48 01 d8`	`50`	

Control Flow: Jumps

CPUs execute instructions in sequence *until told not to*.

One way to interrupt the sequence is with a **`jmp`** instruction:

```
  mov cx, 1337
  jmp STAY_LEET
  mov cx, 0
STAY_LEET:
  push rcx
```

	0x400800			STAY_LEET
Program Binary Code	`mov rcx, 0x1337`	`jmp STAY_LEET`	`mov rcx, 0`	`push rcx`

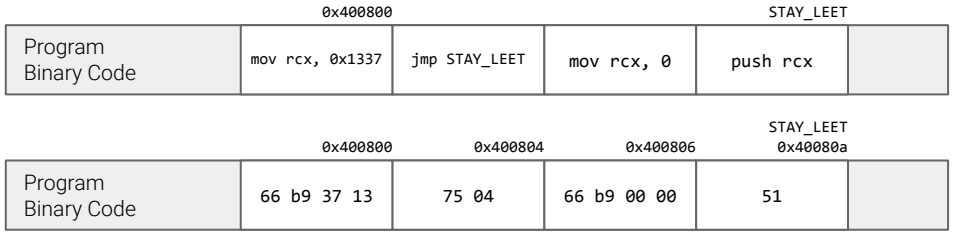
	0x400800	0x400804	0x400806	STAY_LEET 0x40080a
Program Binary Code	`66 b9 37 13`	`eb 04` (skip 4 bytes)	`66 b9 00 00`	`51`

`jmp` skips X bytes and then resumes execution!
But that's still not enough for decisions...

Control Flow: *Conditional* Jumps!

Jumps can rely on conditions!

```
mov cx, 1337
jnz STAY_LEET
mov cx, 0
STAY_LEET:
push rcx
```



je	jump if equal
jne	jump if not equal
jg	jump if greater
jl	jump if less
jle	jump if less than or equal
jge	jump if greater than or equal
ja	jump if above (unsigned)
jb	jump if below (unsigned)
jae	jump if above or equal (unsigned)
jbe	jump if below or equal (unsigned)
js	jump if signed
jns	jump if not signed
jo	jump if overflow
jno	jump if not overflow
jz	jump if zero
jnz	jump if not zero

`jnz` is "jump if not zero", but if **what** is not zero?

Control Flow: Conditions

Conditional jumps check Conditions stored in the "flags" register: **`rflags`**.

Flags are updated by:

Most arithmetic instructions.
Comparison instruction cmp (**`sub`**, but discards result).
Comparison instruction test (**`and`**, but discards result).

Main conditional flags:

Carry Flag: was the 65th bit 1?
Zero Flag: was the result 0?
Overflow Flag: did the result "wrap" between positive to negative?
Signed Flag: was the result's signed bit set (i.e., was it negative)?

Common patterns:

```
cmp rax, rbx; ja STAY_LEET # unsigned rax > rbx. 0xffffffff >= 0
cmp rax, rbx; jle STAY_LEET # signed rax <= rbx. 0xffffffff = -1 < 0
test rax, rax; jnz STAY_LEET # rax != 0
cmp rax, rbx; je STAY_LEET # rax == rbx
```

Thanks to Two's Complement, only the *jumps themselves* have to be signedness-aware.

je	jump if equal	ZF=1
jne	jump if not equal	ZF=0
jg	jump if greater	ZF=0 and SF=OF
jl	jump if less	SF!=OF
jle	jump if less than or equal	ZF=1 or SF!=OF
jge	jump if greater than or equal	SF=OF
ja	jump if above (unsigned)	CF=0 and ZF=0
jb	jump if below (unsigned)	CF=1
jae	jump if above or equal (unsigned)	CF=0
jbe	jump if below or equal (unsigned)	CF=1 or ZF=1
js	jump if signed	SF=1
jns	jump if not signed	SF=0
jo	jump if overflow	OF=1
jno	jump if not overflow	OF=0
jz	jump if zero	ZF=1
jnz	jump if not zero	ZF=0

Looping!

With our conditional jumps, we can implement a loop (think: **`for`**, **`while`**, etc)!

Example: this counts to 10!

```
mov rax, 0
LOOP_HEADER:
inc rax
cmp rax, 10
jb LOOP_HEADER
# now rax is 10!
```

With looping and conditional control flow, we have almost everything we need to write anything we want!

Control Flow: Function Calls!

Assembly code is split into functions with **`call`** and **`ret`**.
`call` pushes **`rip`** (address of the next instruction after the call) and jumps away!
`ret` pops **`rip`** and jumps to it!

Using a function that takes an **`authenticated`** value and returns **`leetness`**:

```
mov rdi, 0
call FUNC_CHECK_LEET
mov rdi, 1
call FUNC_CHECK_LEET
call EXIT

FUNC_CHECK_LEET:
  test rdi, rdi
  jnz LEET
  mov ax, 0
  ret
  LEET:
  mov ax, 1337
  ret

FUNC_EXIT:
  ???
```

```
int check_leet(int authed) {
  if (authed) return 1337;
  else return 0;
}

int main() {
  check_leet(0);
  check_leet(1);
  exit();
}
```

Calling Conventions

Callee and caller functions must agree on argument passing.

Linux x86: push arguments (in reverse order), then call (which pushes return address), return value in eax
Linux amd64: rdi, rsi, rdx, rcx, r8, r9, return value in rax
Linux arm: r0, r1, r2, r3, return value in r0

Registers are *shared* between functions, so calling conventions should agree on what registers are protected.

Linux amd64.
`rbx`, `rbp`, `r12`, `r13`, `r14`, `r15` are "callee-saved"
(the function you call keeps their values safe on the stack).
Other registers are up for grabs
(within reason; e.g., `rsp` must be maintained). Save their values (on the stack)!