CS107 Lecture 20: Two-Minute Etudes

Each function below is shown with one or two lines of C replaced by blanks, alongside the complete x86-64 **`gcc`**-generated assembly (including the instructions for the missing lines).

This time every function calls another (or maybe even itself). Remember: The first six arguments travel in **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, and **`%r9`**, in that order, and the return value comes back in **`%rax`**.

Problem: Reverse Compilation, Function Call and Return Edition

a. The **`rope`** function, just like **`jump`**, takes three **`long`**s as parameters. Most of **`jump`** just rearranges the parameters passed through the **`rope`** call. The **`subq`** and **`addq`** instructions are in place to keep **`%rsp`** 16-byte aligned for the **`call`** instruction.

C Code:

```
long jump(long a, long b,
          long c) {
  long skip = ____________;
  return 2 * skip;
}
```

x86:

```
jump:
   subq    $8, %rsp
   movq    %rdi, %rax
   movq    %rdx, %rdi
   leaq    1(%rsi), %rdx
   movq    %rax, %rsi
   call    rope
   addq    %rax, %rax
   addq    $8, %rsp
   ret
```

b. **`state`** takes a single **`long`** and returns another one. Two intermediate values need to survive a **`call`** instruction, so **`golden`** relies on two caller-owned registers.

C Code:

```
long golden(long warrior) {
  long w = state(warrior);
  return _________________;
}
```

x86:

```
golden:
   pushq   %rbp
   pushq   %rbx
   subq    $8, %rsp
   movq    %rdi, %rbx
   call    state
   movq    %rax, %rbp
   leaq    1(%rbx), %rdi
   call    state
   addq    %rbp, %rax
   addq    $8, %rsp
   popq    %rbx
   popq    %rbp
   ret
```

c. **tot** takes a single parameter of type **long ***. The address of some variable local to **tater** is passed as a parameter to **tot**, so that local needs a home on the stack.

C Code:

```
long tater(long waffle) {
  long fries = ___________;
  _______________________;
  return fries;
}
```

x86:

```
tater:
   subq    $24, %rsp
   leaq    (%rdi,%rdi,2), %rax
   movq    %rax, 8(%rsp)
   leaq    8(%rsp), %rdi
   call    tot
   movq    8(%rsp), %rax
   addq    $24, %rsp
   ret
```

d. A little recursion never hurt anyone, right? Notice the base case never touches the stack pointer.

C Code:

```
long involute(long self) {
  if (_____________)
    return 0;
  return ___________________;
}
```

x86:

```
involute:
   testq   %rdi, %rdi
   jle     .L3
   pushq   %rbx
   movq    %rdi, %rbx
   leaq    -1(%rdi), %rdi
   call    involute
   addq    %rbx, %rax
   popq    %rbx
   ret
.L3:
   movl    $0, %eax
   ret
```

e. **marie** takes two **long**s and returns a **long** as well. **doris** is compiled so that one of its paths ends in **jmp** instead of **call**. Fill in the blanks, of course, but also answer: Why can **gcc** get away with that? [Note: this is tricky, but really neat.]

C Code:

```
long doris(long dirk,
           long evan) {
  if (_____________)
    return _____________;
  return dirk + evan;
}
```

x86:

```
doris:
   cmpq    %rsi, %rdi
   jg      .L4
   leaq    (%rdi,%rsi), %rax
   ret
.L4:
   jmp     marie
```

Answers:

a. rope(c, a, b + 1)
b. w + state(warrior + 1)
c. 3 * waffle and tot(&fries)
d. self <= 0 and self + involute(self - 1)
e. dirk > evan and marie(dirk, evan) Why? doris has nothing to do after marie returns, and marie's ret jumps straight back to doris's caller.