Function Pointers, Vulnerabilities

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 10 due tonight on Gradescope
- Final exam: 7-9 pm Dec 12, Physics 338 (different room!)
  - Cumulative, see practice tests
- Remember to fill out course evaluations
  - 5 pts extra credit on final exam if completed by

    Wednesday, Dec 10 at 5pm!

Example Code

Consider the following code:

```
void apply(double (*f)(double), double *l, unsigned n) {
  for(int i=0; i<n; i+=1)
    l[i] = f(l[i]);
}
```

What are its parameters? How do we call it?

It takes three parameters:

1. A function pointer: `double (*f)(double)`
   This means *f is a pointer to a function that takes a double and returns a double*.
2. A pointer to a list of doubles: `double *l`.
3. The length of the list: `unsigned n`.

Inside the function, we simply loop from `0` to `n - 1`, and we replace each element `l[i]` with `f(l[i])`.
So this function applies **whatever function we pass in** to every element of the array.

Example Code

```
int main() {
  double vals[5] = { M_PI, M_E, 2130, 1, 0 };
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(sqrt, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(sin, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
  apply(cos, vals, 5);
  for(int i=0; i<5; i+=1) printf("%f\t", vals[i]);
  puts("");
}
```

We start with an array of five doubles, including `M_PI` and `M_E` from the math library.

We print the array so we can see the initial values.

Then we call:

```
apply(sqrt, vals, 5);
```

Notice the important detail:
We pass **`sqrt` without parentheses**.
That means we are *not calling* `sqrt` here. We are passing **the address of the function**.

`apply` will call it later for each element in the array.

We repeat the same process with:

```
apply(sin, vals, 5);
apply(cos, vals, 5);
```

Each time, the array gets transformed in place using the function we passed in.

So the point of this example is:
we can write a generic function that applies any math function to an array, as long as it takes and returns a double.

Function Pointers

```
void apply(double (*f)(double), double *l, unsigned n) {
  for(int i=0; i<n; i+=1)
    l[i] = f(l[i]);
}
```

`double (*f)(double)` means:

- `*f` – `f` is a pointer
- `(double)` – with a single `double` argument
- `double` – that returns a `double`
- `( )` – to make it parse as `*f` instead of `double *`

`double (*f)(double)` means:

- `*f` → f is a pointer
- `(double)` → it takes one double argument
- `double` → it returns a double
- the extra `( )` around `*f` are required because of operator precedence; otherwise the compiler would think we are declaring a function returning a pointer.

- Once we have a function pointer, calling it is easy.
  We just write: `f(x);`

  Exactly the same way we call a normal function.

- One more note: math functions like `sqrt`, `sin`, and `cos` are in the math library, so we must compile with:

  ```
  clang file.c -lm
  ```

  `-lm` means 'link with the math library'.
  C does not link it automatically.

Function Pointers

```
(const char *)(*fv)(const char *) = findVowel;
```

A **function pointer** is a pointer that references code

- In assembly, the address of the function is just a label
  - Follow calling conventions
  - Push return address
  - Jump to that label
- C tries to hide that with this function pointer syntax
- Be aware of operator precedence!

Conceptually, a function pointer is very simple: it is a pointer that references code.

At the assembly level, a function is just a label—an address in memory. Calling a function means jumping to that address, following the standard calling conventions, and returning when done.

C hides this behind syntax like `double (*f)(double)`, but the meaning is straightforward:
you store the address of a function in a variable, and later you jump to it when you call `f`.

There is one thing to be careful about: **operator precedence**.
This is why the parentheses around `*f` are required.

Function pointers exist because C does not have first-class functions or closures.
But they still let us pass behavior—like `sin`, `sqrt`, or `cos`—into generic code, which can be a very powerful idea.

Vulnerabilities...
...and when to report them

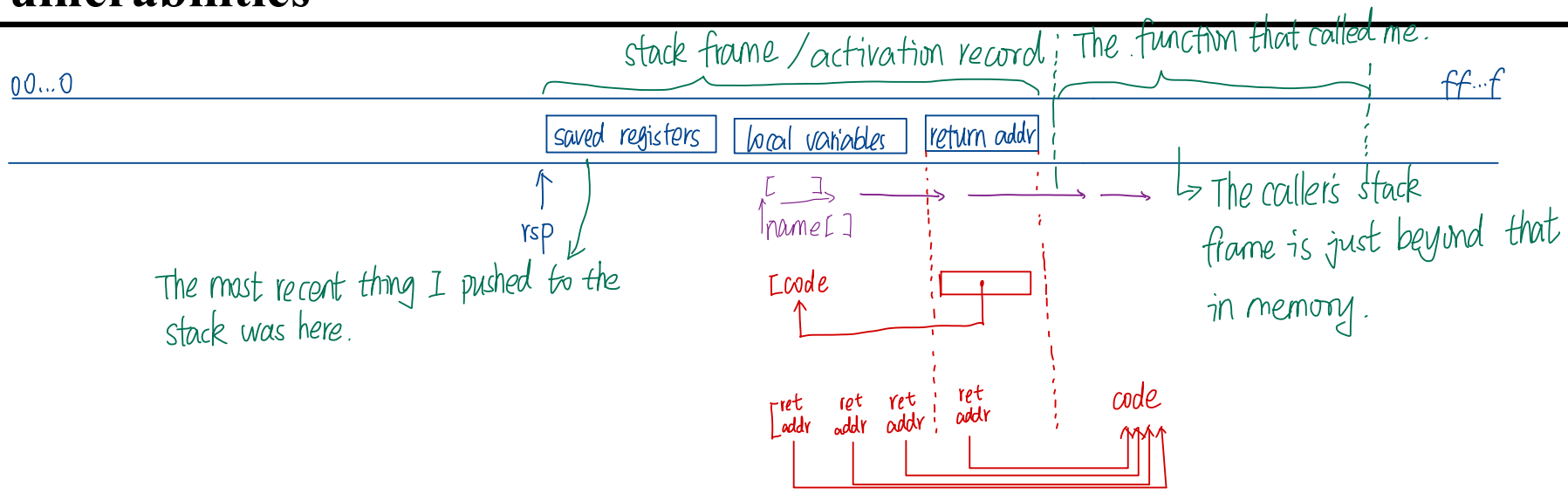
Memory

Common Memory Problems (from reading)

- Memory leak
- Uninitialized memory
- Accidental cast-to-pointer
- Wrong use of 'sizeof'
- Unary operator precedence mistakes
- Use after free
- Stack buffer overflow
- Heap buffer overflow
- Global buffer overflow
- Use after return
- Uninitialized pointer
- Use after scope

Vulnerabilities

So what's going on here? Let's draw a memory. horizontally.



1. What happens if a program asks for your name but doesn't check the length?

Imagine the program allocates:

```
char name[20];
```

But then it reads **100 bytes** into it.

Where do the extra 80 bytes go?

They overwrite whatever comes next:

- other local variables
- saved registers
- the return address
- the caller's frame

This is the fundamental danger of buffer overflows."

2. Overwriting the return address

"The return address is just a number — it is the address the CPU jumps to when the function returns.

If an attacker can overwrite the return address, the attacker can choose **where the program jumps next**."

3. Classic attack: Inject code + jump to it

"This is the traditional stack–based exploit:

1. Write malicious machine instructions into the buffer (as input).
2. Keep writing until you reach the return address.
3. Overwrite the return address with the starting address of that buffer.

When the function returns, it jumps directly into the attacker's code.

This is how early exploitation worked:
'put code on the stack, then return into it.'"

4. Problem: How do attackers know exactly how far to overwrite?

"You might not know the exact number of bytes between the buffer and the return address.

Attackers solved this by writing *tons* of return addresses, over and over, so eventually one of them lands exactly on the real return address.

Because of 8-byte stack alignment, eventually one of those overwrite attempts matches the correct alignment."

5. Compiler defense: Stack Canary

“To defend against overwriting the return address, compilers insert a random value — a ‘canary’ — right before the return address.

Memory layout becomes:

```
[ local variables ]
[ random canary ]
[ return address ]
```

Before the function returns, the compiler-generated prologue checks that the canary is unchanged.

- If it changed → abort with ‘stack smashing detected’.
- If it’s intact → normal return.

To exploit the overflow, the attacker must guess the canary — extremely difficult.”

Vulnerabilities

Vulnerability: a program for which something like this could happen (security holes)

- Ex: stack buffer overflow possibility
- Not necessarily malicious (like when we talked about backdoors) might not have been on purpose

Exploit: a way to use a vulnerability or backdoor that has been created

- Ex: the magic long word to type into our program

Vulnerabilities

Modern systems prevent executing code on the stack

"This classic attack doesn't usually work today.

Why?

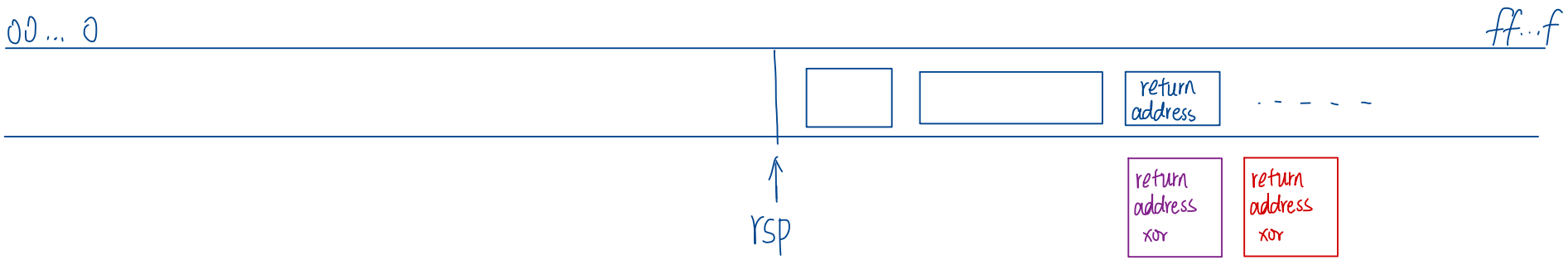
Because modern operating systems mark the stack as:

- readable
- writable
- **not executable**

This is the NX bit (No-eXecute).

If you try to execute instructions stored on the stack:
→ immediate segmentation fault."

Vulnerabilities




```
xorl %eax, %eax
movq %rdi, %rsi
      ⋮
(The code "I" want to do)
```

```
xorl %eax, %eax
ret
(functions pre-compiled)
movq %rdi, %rsi
ret
```

Vulnerabilities

The modern workaround: ROP (Return-Oriented Programming)

"So if we cannot execute our own code, we must use code that **already exists**.

This leads to Return-Oriented Programming (ROP).

The idea is:

- The program and libc already contain thousands of tiny instruction sequences ending in `ret`.
- Each small sequence is called a *gadget*.
- If you can control the return address, you can chain these gadgets together:
  - first return to a gadget that clears a register
  - then return to a gadget that loads a value
  - then return to a gadget that calls a function

By chaining dozens of gadgets, you can build an arbitrary computation **without injecting any code**.

You only manipulate return addresses."

OS defense: ASLR (Address Space Layout Randomization)

"To defeat ROP, modern operating systems randomize memory layout:

- the stack address
- the heap address
- the location of shared libraries
- the base address of the program

This means attackers cannot predict where gadgets are located.

Without knowing the address of libc functions or gadgets, building a ROP chain becomes much harder."

“Even though the stack is non-executable, attackers can still run malicious behavior by chaining together short pieces of code that already exist in memory.

The idea is this:

Your program and the standard library come with thousands of precompiled functions. Inside those functions are many tiny instruction sequences like:

```
xor rax, rax
ret
```

or

```
mov rdi, rsi
ret
```

These small instruction sequences are called **gadgets**. They end with a `ret`, which is crucial.

So an attacker does the following:

1. Find a gadget that does the first thing they want

For example, ‘xor rax, rax’.
This gadget exists at some address in the code segment (text segment).

The attacker overwrites the function’s return address so that when the function returns, it jumps into the *middle of another function*, exactly where that gadget is located.

2. After the gadget runs, it executes `ret`

A `ret` instruction pops the next return address off the stack.
But the attacker controls the stack now.

So what happens?

- The attacker places another address on the stack, pointing to another gadget.
- When the first gadget finishes, `ret` jumps to the next gadget.

3. Repeat

The attacker chains:

- XOR gadget
- MOV gadget
- ADD gadget
- etc.

Each time:

1. A gadget executes a tiny instruction sequence.
2. The final `ret` jumps to the next address the attacker placed on the stack.

Eventually, the attacker can build a full “program” made entirely of:

✔️ existing code
✔️ ending in `ret`
✔️ placed in a custom sequence by manipulating the stack

This is called **Return-Oriented Programming (ROP)**.

Warning

Anytime you can modify memory the programmer did not expect you to be able to modify, there's something you can do to give yourself power or rights the programmer didn't mean to give you

Memory

Common Memory Problems (from reading)

- Memory leak
- Uninitialized memory
- Accidental cast-to-pointer
- Wrong use of ‘sizeof‘
- Unary operator precedence mistakes
- Use after free
- Stack buffer overflow
- Heap buffer overflow
- Global buffer overflow
- Use after return
- Uninitialized pointer
- Use after scope

Vulnerabilities

What should you do when you find a vulnerability?

Good Practices

Good practices when finding a vulnerability:

1. Tell the owner
2. Wait (a reasonable amount of time for a fix)
3. Publish

1. The Ethical Question

When you find a vulnerability, there are a few obvious possibilities:

- You could exploit it ("cash out").
- You could report it.
- Or you could ignore it.

People who report vulnerabilities are *usually trying to be good*—at least in appearance.
But **not everyone agrees on what "good" means**.

Real-World Example: The Iranian Centrifuge Incident

Someone discovered a vulnerability in the control system for centrifuges in Iran.
They intended to report it "responsibly."
They went to the CIA.

The CIA viewed Iran as a threat, so instead of simply fixing the issue, they:

- Weaponized the vulnerability
- Made the centrifuges spin too fast
- Destroyed the facility

This example shows:

- One side believed it was "good"
- The other side suffered damage and viewed it as "bad"

So ethical interpretations vary.

2. Industry Standard: Responsible Disclosure

Despite the ethical complexity, there *is* an agreed-upon best-practice process.

🔹 **Step 1 — Notify the Owner**

Tell the developer or company (e.g., Google, Apple, Microsoft).
Provide enough detail for them to:

- Reproduce the bug
- Understand the issue
- Fix it

🔹 **Step 2 — Wait a Reasonable Amount of Time**

Give the developer adequate time to fix the bug.

- You've seen how long even small bugs can take to fix.
- Large systems like Chrome take even longer.

There is no single strict number, but it must be **reasonable**.

🔹 **Step 3 — Publish the Vulnerability**

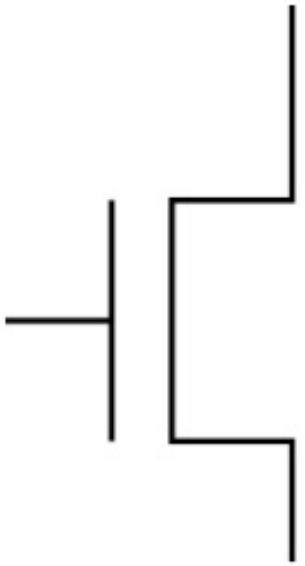
Publish it **whether or not it was fixed**:

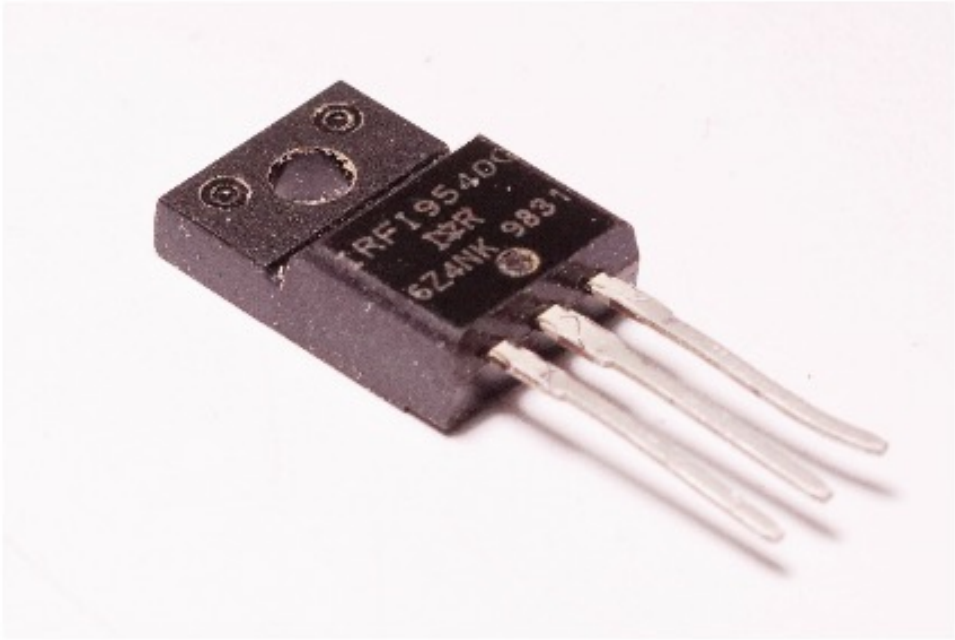
- If not fixed → publishing creates pressure to fix it
- If still not fixed → users can decide not to trust that software
- If fixed → publishing warns users to update immediately

This is why CVEs list things like:
"Heap buffer overflow in Chrome prior to version 112…"
These entries help users know **when to update** and which versions are unsafe.

Where have we been?

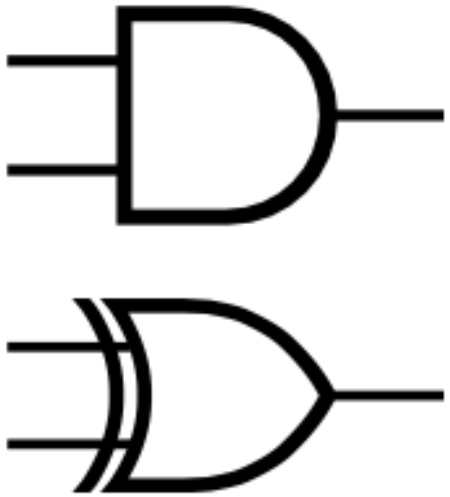
Where are we going?



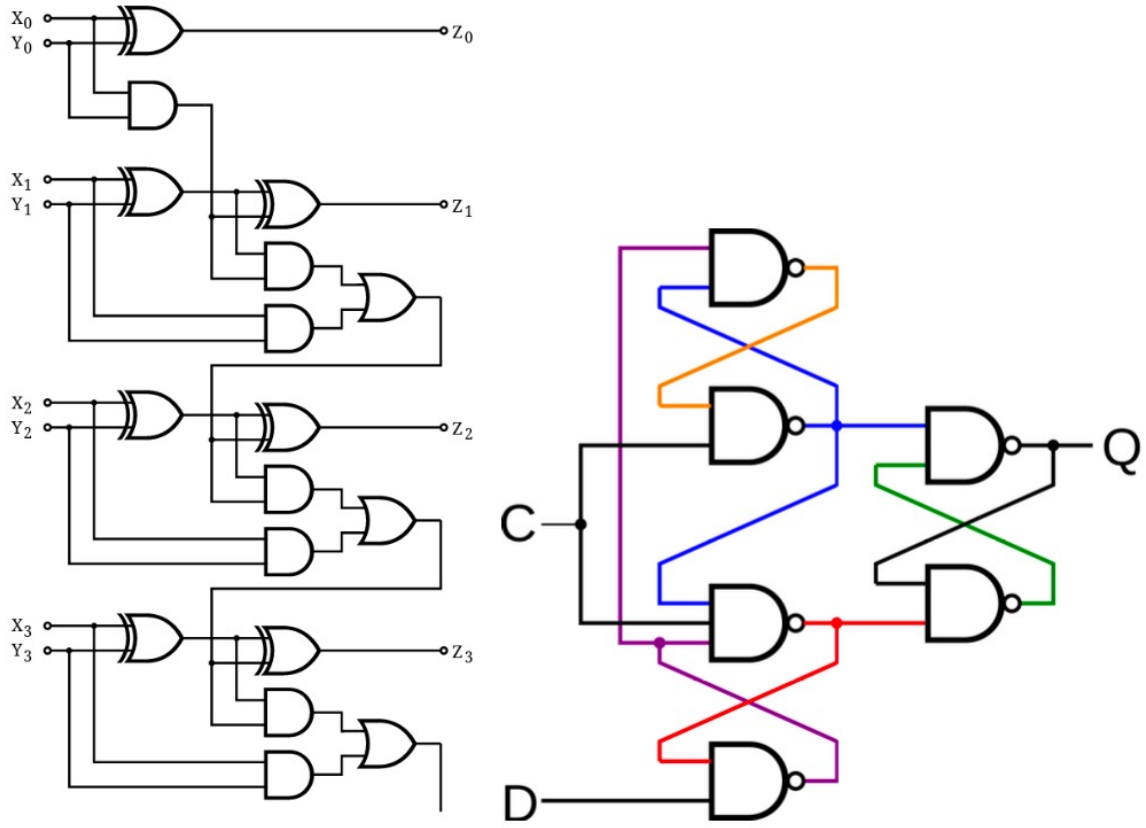


0 and 1

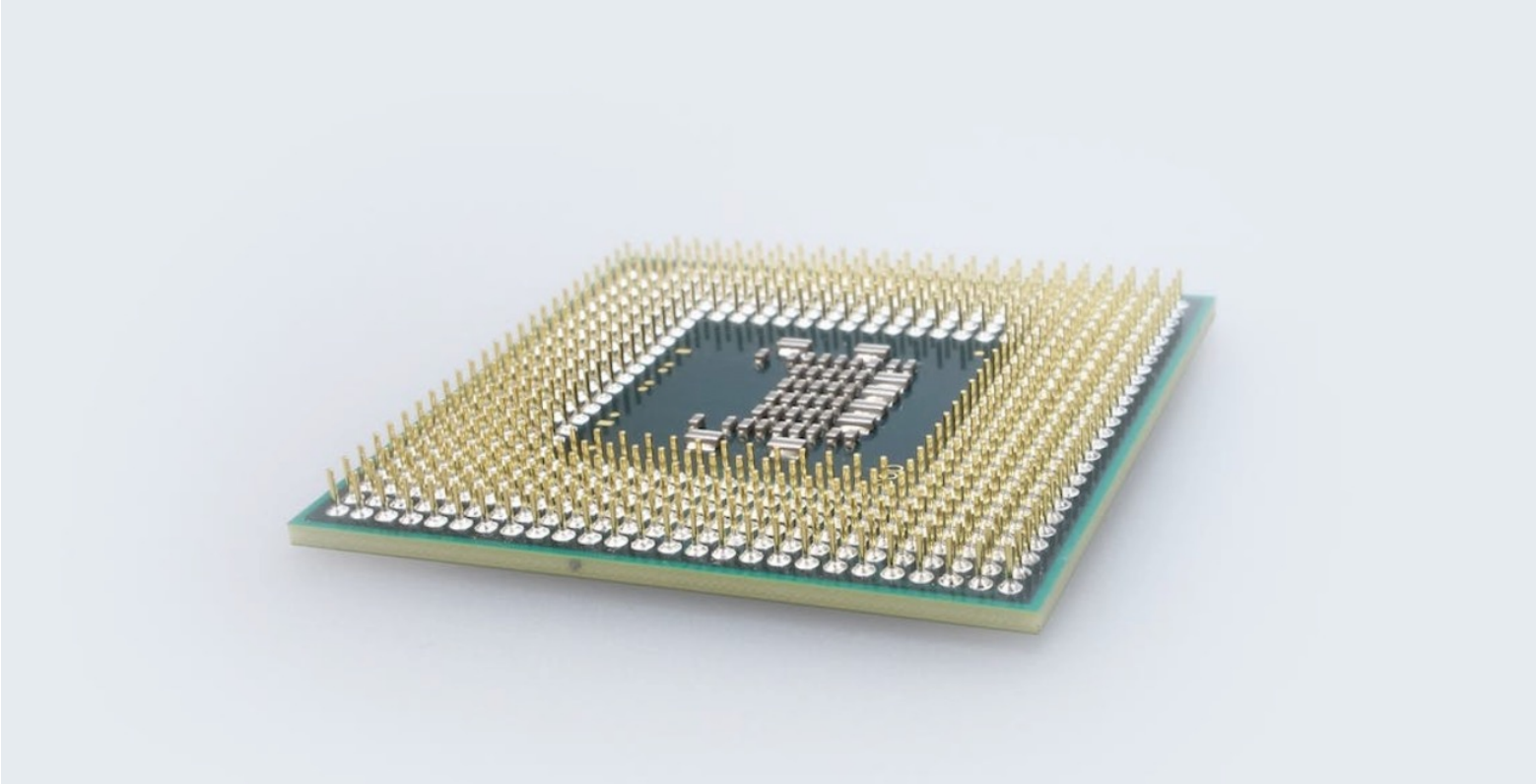
Where are we going?



Where are we going?



Where are we going?



Where are we going?

```
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   31 c0                   xor    %eax,%eax
   6:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   d:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  14:   48 c7 45 f0 04 00 00    movq   $0x4,-0x10(%rbp)
  1b:   00
  1c:   48 8d 4d f8             lea    -0x8(%rbp),%rcx
  20:   48 89 4d e8             mov    %rcx,-0x18(%rbp)
  24:   48 8d 4d f0             lea    -0x10(%rbp),%rcx
  28:   48 89 4d e0             mov    %rcx,-0x20(%rbp)
  2c:   48 8b 4d e8             mov    -0x18(%rbp),%rcx
  30:   48 63 09                movslq (%rcx),%rcx
  33:   48 89 4d d8             mov    %rcx,-0x28(%rbp)
  37:   48 8b 4d e0             mov    -0x20(%rbp),%rcx
  3b:   48 8b 09                mov    (%rcx),%rcx
  3e:   89 4d d4                mov    %ecx,-0x2c(%rbp)
  41:   5d                      pop    %rbp
  42:   c3                      retq
```

Where are we going?

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Where are we going?

Along the way:

- Interact with the terminal and SSH
- Learn basic command-line tools and editors
- Access command-line documentation
- Practice C and using the C standard library
- Learn how to debug with lldb and the address sanitizer
- Discuss related security and social topics
- Think about the next steps of Generative AI

Finale

Along the way:

- Interact with the terminal and SSH
- We have covered a LOT
- Electricity on wires
- Transistors to gates (AND, OR, ...)
- Combined gates to make circuits
- Connected circuits and registers to build a 1-byte computer
- Wrote an ISA for that computer (1-byte instructions, Toy ISA)
- Expanded to x86-64 Assembly (saw the binary)
- Concluded with C (how it compiles and connects with Assembly)

Finale

Thanks for a great semester!