

# CS 453/698: Software and Systems Security

## **Module: An In-depth Study of Memory Errors**

Lecture: Exploit mitigation

Meng Xu (*University of Waterloo*)

Winter 2025









# Themes of mitigation

- Principle of least privileges (PoLP)
  - reduce permissions unless absolutely needed
- Reference monitoring / program herding
  - keep an eye on the program while it is executing
- Moving-target defense







# DEP a.k.a., $W \oplus X$

# DEP a.k.a., $W \oplus X$

DEP – Data Execution Prevention

$W \oplus X$  – Write exclusive-or eXecute

You can either **write data** **OR** **execute code** in a memory region,  
but **never both**.

# DEP a.k.a., $W \oplus X$

DEP – Data Execution Prevention

$W \oplus X$  – Write exclusive-or eXecute

You can either **write data** **OR** **execute code** in a memory region,  
but **never both**.

Implementation: `gcc -z execstack`.



## Motivation for type-based heap allocation

A more realistic use-after-free (UAF) exploit:

```
1 struct 0 {
2     int (*oper)(void);
3     long id;
4 };
```

```
1 void bar(long id) {
2     struct 0 *x =
3         malloc(sizeof(struct 0));
4
5     x->oper = __safe_function_2;
6     x->id = id;
7     struct 0 *q = x;
8     free(x);           // q is dangling
9
10    /* later in the code */
11    q->oper();
12 }
```









# CFI: introduction

Control-Flow Integrity (CFI) is a classic example of **runtime reference monitor** in software security.

# CFI: introduction

Control-Flow Integrity (CFI) is a classic example of **runtime reference monitor** in software security.

CFI is also sometimes referred to as **program shepherding**

*monitoring control flow transfers during program execution to enforce a security policy — from [a paper in USENIX Security'02](#).*

# Basic use cases of CFI

```
1 void f1();
2 void f2();
3 void f3();
4 void f4(int, int);
5
6 void foo(int usr) {
7     void (*func)();
8
9     if (usr == MAGIC)
10         func = f1;
11     else
12         func = f2;
13
14     // forward edge CFI check
15     CHECK_CFI_FORWARD(func);
16     func();
17
18     // backward edge CFI check
19     CHECK_CFI_BACKWARD();
20 }
```





# Basic use cases of CFI

```
1 void f1();
2 void f2();
3 void f3();
4 void f4(int, int);
5
6 void foo(int usr) {
7     void (*func)();
8
9     if (usr == MAGIC)
10         func = f1;
11     else
12         func = f2;
13
14     // forward edge CFI check
15     CHECK_CFI_FORWARD(func);
16     func();
17
18     // backward edge CFI check
19     CHECK_CFI_BACKWARD();
20 }
```

## Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

## Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

## Option 3: allow functions with type signature `void (*)()`

- f1, f2, f3



## Example: Microsoft Return-flow Guard (RFG)

RFG was our compatible, ABI compliant, performant software shadow stack

## Compile Time

NOP's added to the prolog & epilog of all functions

Metadata added to the image to locate the prolog and epilog NOP bytes

## Runtime

## Process Start

- 1TB shadow stack region created
- Region cannot be queried
- A/V's in region are fatal
- FS segment points to the shadow stack of the current thread

Image  
Load

- If process enables RFG: patch NOP's with RFG prolog/epilog

## Function

- Prolog: Push return address to shadow stack
- Epilog: Fast fail if return address on stack and shadow stack are mismatched

Parent Function	Child Function
[...] //Prior code	
call ChildFunction	
	mov rax, [rsp]
	mov fs:[rsp], rax
	[...] //Child code
	mov rcx, fs:[rsp]
	cmp rcx, [rsp]
	jne _fast_fail
	ret
0xABCD: [...] //Remainder of parent function	

If attacker changes the return address at these points RFG is defeated

## RFG relies on a secret: the shadow stack's virtual address

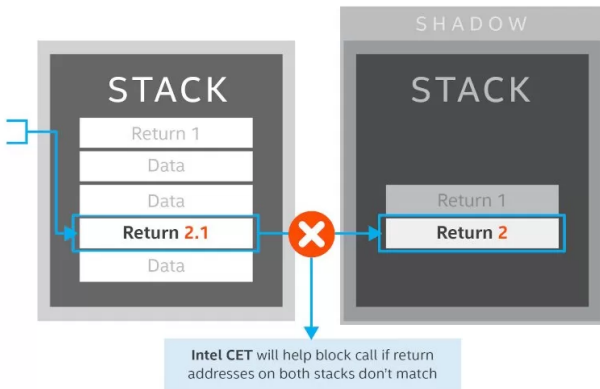
Illustration taken from [Microsoft Talk: The Evolution of CFI Attacks and Defenses](#)



# Back-edge protection: shadow stack

## SHADOW STACK (SS)

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.



# CET: shadow stack

- For every regular stack CET adds a shadow stack region, which is indexed via a new register %ssp.
- Regular memory stores (executed from any ring) are not allowed in shadow stack region

When enabled,

- Each time a call instruction gets executed, in addition to the return address being pushed onto the regular stack, a copy of it is also pushed (automatically) onto the shadow stack.
- Each time a ret instruction gets executed, the return addresses pointed by %rsp and %ssp are (automatically) popped from the two stacks, and their values are compared together.

# CET: Indirect Branch Tracking (IBT)

CET introduces a new (4-byte) instruction, i.e., `endbr`, which becomes the **only** allowed target of indirect `call/jmp` instructions.

In other words, forward-edge transfers via (indirect) `call` or `jmp` instructions are pinned to code locations that are “marked” with an `endbr`; else, an exception (`#CP`) is raised.

## IBT example

```
1 void main() {
2     int (*f) {};
3     f = foo;
4     f();
5 }
6
7 int foo() {
8     return 0;
9 }
```

```

1  <main>:
2  movq    $0x4004fb, -8(%rbp)
3  mov     -8(%rbp), %rdx
4  call    *%rdx
5  :
6  retq
7
8  <foo>:
9  endbr64
10 :
11 mov     rax, 0
12 :
13 retq

```

## IBT example

```

1  void main() {
2      int (*f) {};
3      int (*g) {};
4      f = foo;
5      g = bar;
6      f();
7      g();
8  }
9
10 int foo() {
11     return 0;
12 }
13
14 int bar() {
15     return 1;
16 }

```

```

1  <main>:
2  movq    $0x4004fb, -16(%rbp)
3  mov     -16(%rbp), %rdx
4  call    *%rdx
5  mov     -8(%rbp), %rdx
6  call    *%rdx
7  :
8  retq
9
10 <foo>:
11 endbr64
12 :
13 mov     rax, 0
14 :
15 retq
16
17 <bar>:
18 endbr64
19 :
20 mov     rax, 1
21 :
22 retq

```

## Security boundaries of CFI-protected programs

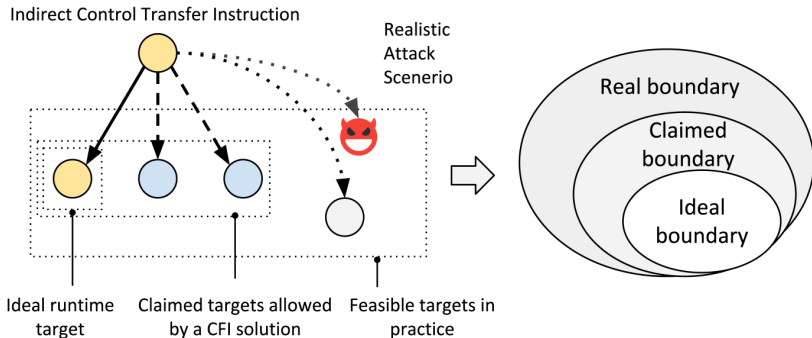


Figure from a paper published in ACM CCS'20

# Pointer integrity

**Goal:** ensures **pointers** in memory remain **unchanged**.

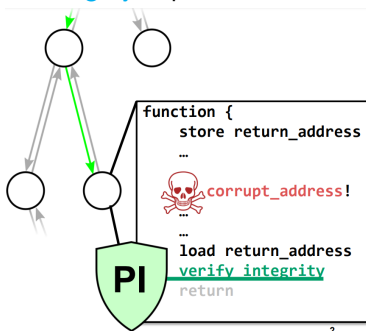




# Pointer integrity

**Goal:** ensures **pointers** in memory remain **unchanged**.

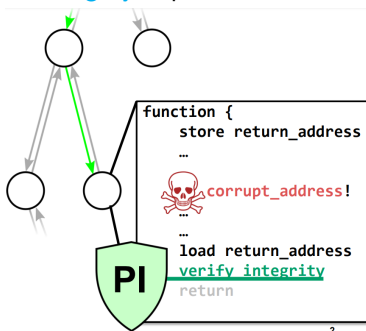
- i.e., the **value of the pointer** remains unchanged, not the memory content referred to by this pointer.
- Perfect **code pointer integrity** implies control-flow integrity (CFI).



# Pointer integrity

**Goal:** ensures **pointers** in memory remain **unchanged**.

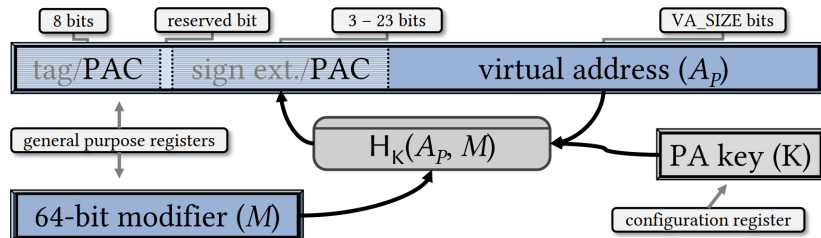
- i.e., the **value of the pointer** remains unchanged, not the memory content referred to by this pointer.
- Perfect **code pointer integrity** implies control-flow integrity (CFI).



- Data pointer integrity is also important (e.g., against data-only attacks and data-oriented programming) and can be (partially) achieved via Pointer Authentication.

# Overview of Arm Pointer Authentication (PA)

Available since Armv8.3-A instruction set architecture (ISA) when the processor executes in 64-bit Arm state (AArch64)



PA consists of a set of instructions for creating and authenticating **pointer authentication codes (PACs)**.















# Why entropy in security?

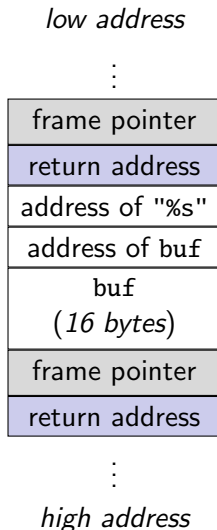
Nondeterminism is useful in software security when

- it has no impact on the intended finite state machine BUT
- limits attackers' abilities to program the weird machine.

**In the rest of this lecture:** we will examine some standard / deployed practices of safely introducing nondeterminism to boost system and software security.

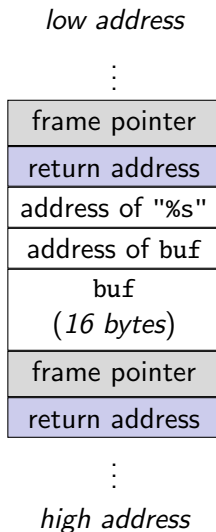
## Recap: stack overflow

```
1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4 }
```



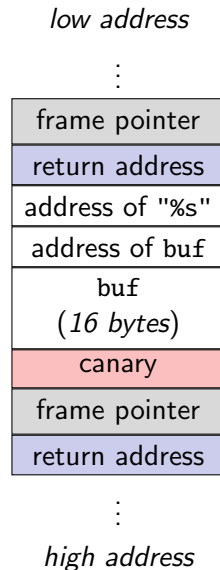
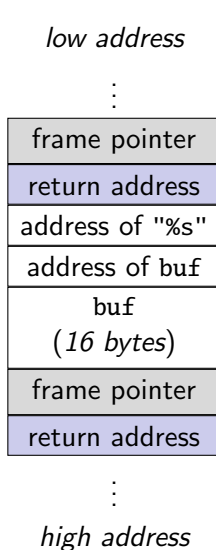
## Stack canary intuition

```
1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4 }
```



## Stack canary intuition

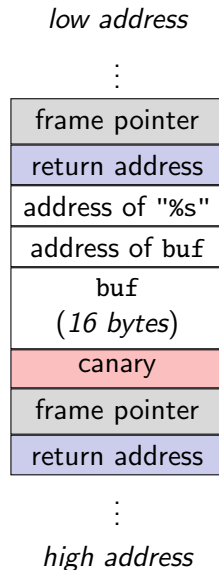
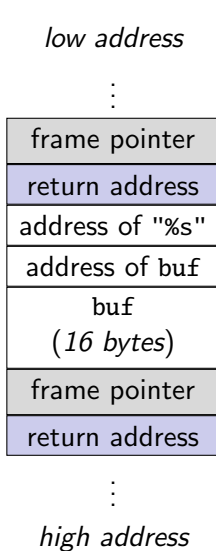
```
1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4 }
```



## Stack canary intuition

```
1 int main() {
2     char buf[16];
3     scanf("%s", buf);
4 }
```

- On function entry, push canary value  $X$  onto stack.
- On function return, check canary value is still  $X$ .



# Original use of canary



**Figure:** Canaries in coal-mining. Credits / Trademark: Alamy Stock Photo

# The default implementation in GCC

```
1 int main() {  
2     char buf[16];  
3     scanf("%s", buf);  
4 }
```

```
1 extern uintptr_t __stack_chk_guard;  
2 noreturn void __stack_chk_fail(void);  
3  
4 int main() {  
5     uintptr_t canary = __stack_chk_guard;  
6  
7     char buf[16];  
8     scanf("%s", buf);  
9  
10    if ((canary = canary ^ __stack_chk_guard) != 0) {  
11        __stack_chk_fail();  
12    }  
13 }
```



# The default implementation in GCC

```
1 int main() {  
2     char buf[16];  
3     scanf("%s", buf);  
4 }
```

```
1 extern uintptr_t __stack_chk_guard;  
2 noreturn void __stack_chk_fail(void);  
3  
4 int main() {  
5     uintptr_t canary = __stack_chk_guard;  
6  
7     char buf[16];  
8     scanf("%s", buf);  
9  
10    if ((canary = canary ^ __stack_chk_guard) != 0) {  
11        __stack_chk_fail();  
12    }  
13 }
```

- The `__stack_chk_guard` and `__stack_chk_fail` symbols are normally supplied by a GCC library called `libssp`.
- You also have the option of specifying your own value for stack canaries.



# Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?

# Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?

# Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?
- When to do the integrity check?
  - on function return? is that enough?

# Design choices of stack canaries

- Which value should we use as canary?
  - deterministic? secret? random?
- What is the granularity of the canary invocation?
  - per function? per execution?
- When to do the integrity check?
  - on function return? is that enough?
- How much randomness is needed?
  - 1 byte? 8 bytes? 64 bytes?

# Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value

# Limitations of stack canary

- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value
- Limited protection for frame pointer and return address only
  - other stack variables are not protected



# Limitations of stack canary

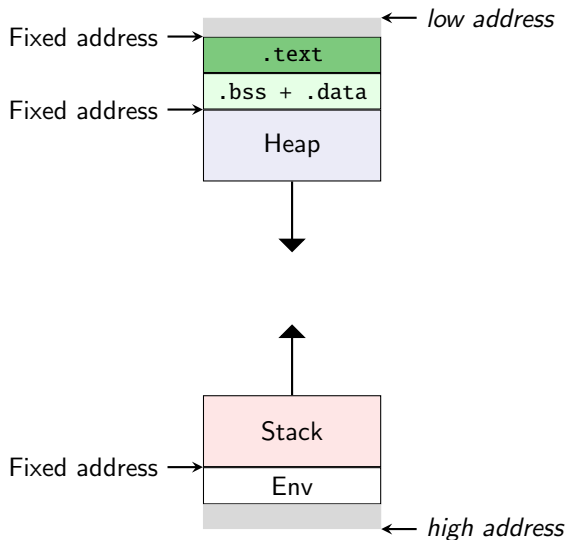
- Vulnerable to information leak
  - e.g., using a buffer over read to retrieve the canary value
- Limited protection for frame pointer and return address only
  - other stack variables are not protected
- Unable to defend against arbitrary writes
  - i.e., non-continuous overrides

# Randomize the addresses

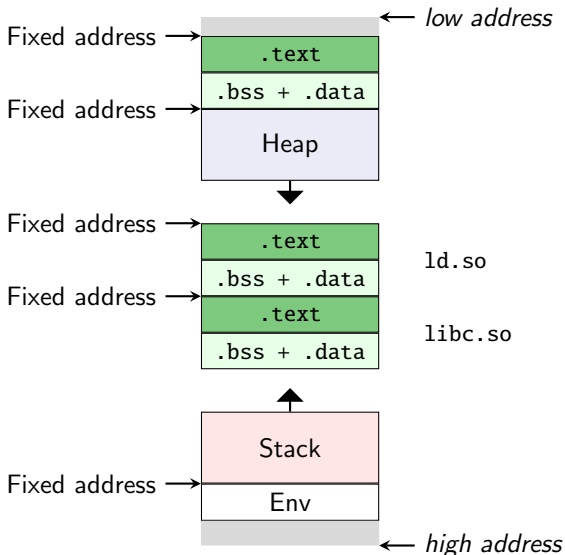
**ASLR** — Address Space Layout Randomization, is a system-level protection that **randomly** arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

**PIE** — Position Independent Executable, is a body of machine code that executes properly **regardless of its absolute address**. This is also known as position-independent code (PIC).

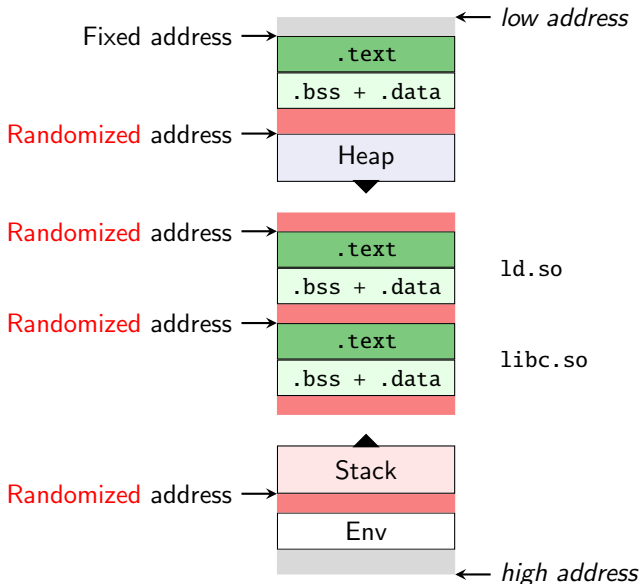
# Base case: static program



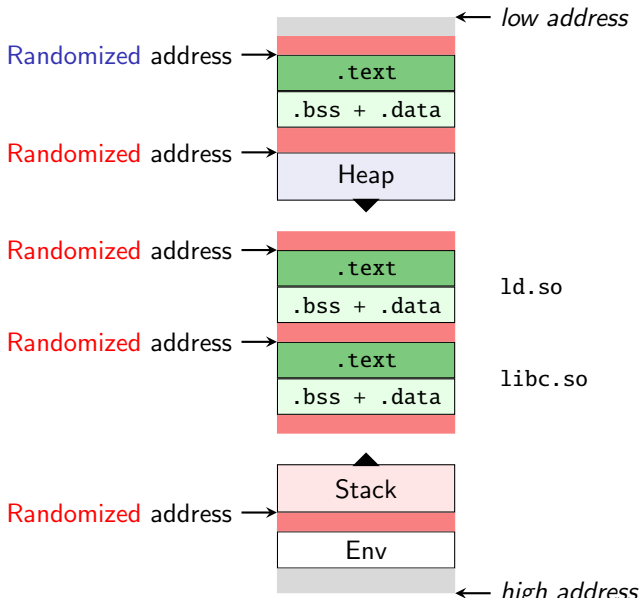
# Static program + shared libraries



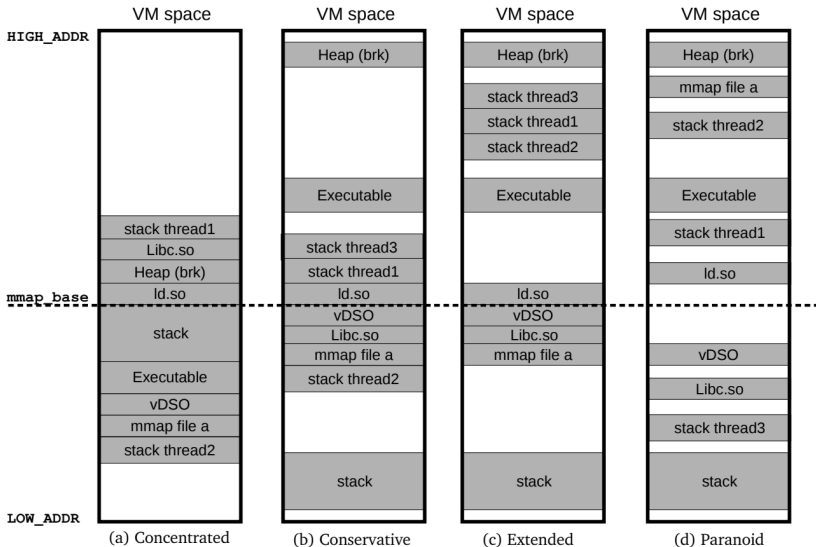
# Static program + shared libraries + ASLR



# Static program + shared libraries + ASLR + PIE



# Paranoid randomization



**Figure:** Different level of randomization proposed by the [ASLR-NG project](#)

# Limitations of ASLR + PIE

- Limited entropy
  - visualized by the [ASLR-NG project](#)



# Limitations of ASLR + PIE

- Limited entropy
  - visualized by the [ASLR-NG project](#)
- Memory layout inheritance
  - Child processes inherit/share the memory layout of the parent.

# Motivation for secure heap allocators

Memory errors are equally (if not more) likely to happen on heap objects (compared with stack objects) which can cause all sorts of unexpected behaviors.

# A heap buffer overflow case

```
1 struct dispatcher {
2     uint64_t counter;
3     int (*action)(uint64_t counter, char *data);
4 }
5
6 int main() {
7     char *p1 = malloc(16);
8     char *p2 = malloc(sizeof(struct dispatcher));
9     p2->counter = 0;
10    p2->action = /* some valid function */;
11
12    scanf("%s", p1);
13    int result = p2->action(p2->counter, p1);
14
15    free(p1);
16    free(p2);
17    return result;
18 }
```

# A heap use-after-free case

```
1 struct dispatcher {
2     uint64_t counter;
3     int (*action)(uint64_t counter, char *data);
4 }
5
6 char *p1;
7
8 void main() {
9     p1 = malloc(16);
10    pthread_create(/* ... */, thread_1);
11    pthread_create(/* ... */, thread_2);
12    /* wait for thread termination */
13 }
```

---

```
1 void thread_1() {
2     scanf("%15s", p1);
3     /* ... compromised here ... */
4     /* use-after-free */
5     free(p1);
6     ((struct dispatcher *)p1)
7     ->action = /* bad function */;
8 }
```

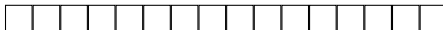
```
1 void thread_2() {
2     char *p2 = malloc(
3         sizeof(struct dispatcher));
4     p2->counter = 0;
5     p2->action = /* good function */;
6     p2->action(p2->counter, p1);
7     free(p2);
8 }
```

# Secure heap allocators

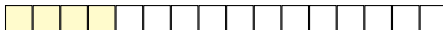
These exploits have **implicit assumptions** on the **layout** of the heap, which can be invalidated by a secure heap allocator.

# Basic allocator example

Initial state:



`p1 = malloc(16);`



`p2 = malloc(sizeof(..));`



`free(p1);`



`p3 = malloc(sizeof(..));`

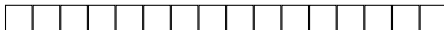


---

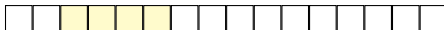
<sup>0</sup>Each square is a 4-byte box

# Allocator + random placement

Initial state:



`p1 = malloc(16);`



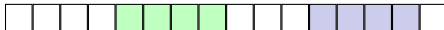
`p2 = malloc(sizeof(..));`



`free(p1);`



`p3 = malloc(sizeof(..));`

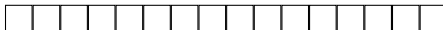


---

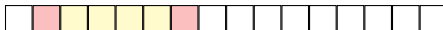
<sup>0</sup>Each square is a 4-byte box

# Allocator + random placement + canary

Initial state:



`p1 = malloc(16);`



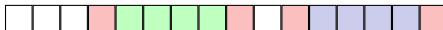
`p2 = malloc(sizeof(..));`



`free(p1);`



`p3 = malloc(sizeof(..));`



---

<sup>0</sup>Each square is a 4-byte box



# Intuition: gene/DNA diversity

**In biology**, maintaining high **genetic diversity** allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.

---

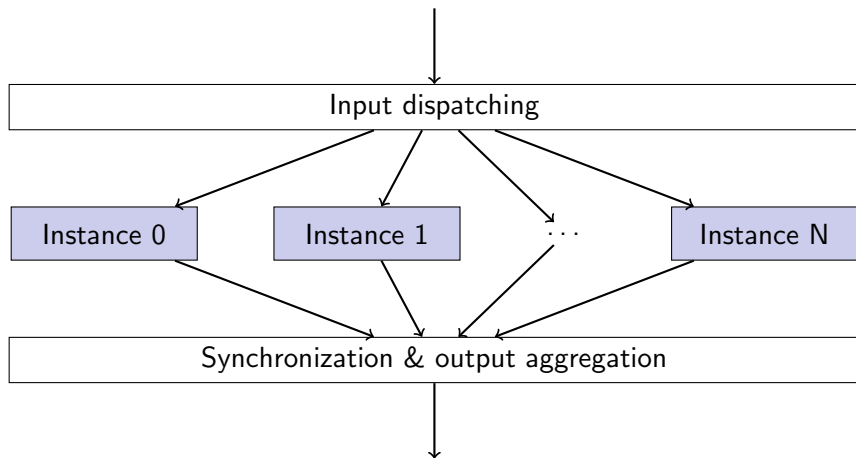
# Intuition: gene/DNA diversity

**In biology**, maintaining high **genetic diversity** allows species to adapt to future environmental changes, survive from deadly diseases, and avoid inbreeding.

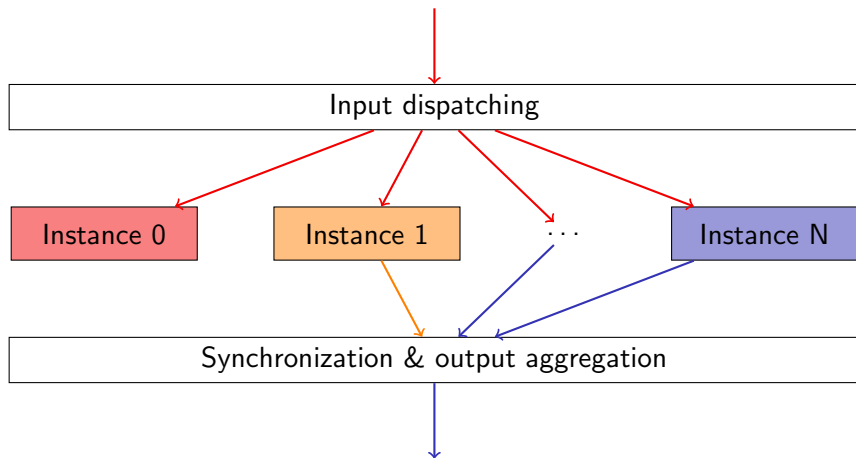
---

Similarly, we expect **software diversity** to protect software systems (especially critical systems) from deadly viruses and attacks while also serving as an early signal of being attacked.

# Core architecture



# Core architecture (under attack)



# Challenges of applying diversity-based defenses

- Source of diversity
- Synchronization of diversified instances

# Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation

# Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation
- N-version programming
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams

# Source of diversity

- Compiler/loader-assisted diversity
  - e.g., direction of stack growth
  - e.g., different canary values
  - e.g., different sanitizer instrumentation
- N-version programming
  - e.g., different language VM (V8 vs SpiderMonkey)
  - e.g., different applications (nginx vs apache web server)
  - e.g., similar applications from independent vendors/teams
- Platform diversity
  - e.g., different libc implementations (glibc vs musl libc)
  - e.g., Adobe Reader on MacOS and Windows
  - e.g., Server programs on Intel and ARM CPUs



## Mode of synchronization

- Online mode (via rendezvous points)
- Offline mode (via record-and-replay)

The key is to synchronize all sources of nondeterminism.

〈 End 〉