Lec 08: Rethinking Memory Safety

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Recap

- Memory safety attacks and mitigations
  - Can we move beyond memory corruption?

Defending against memory safety vulnerabilities

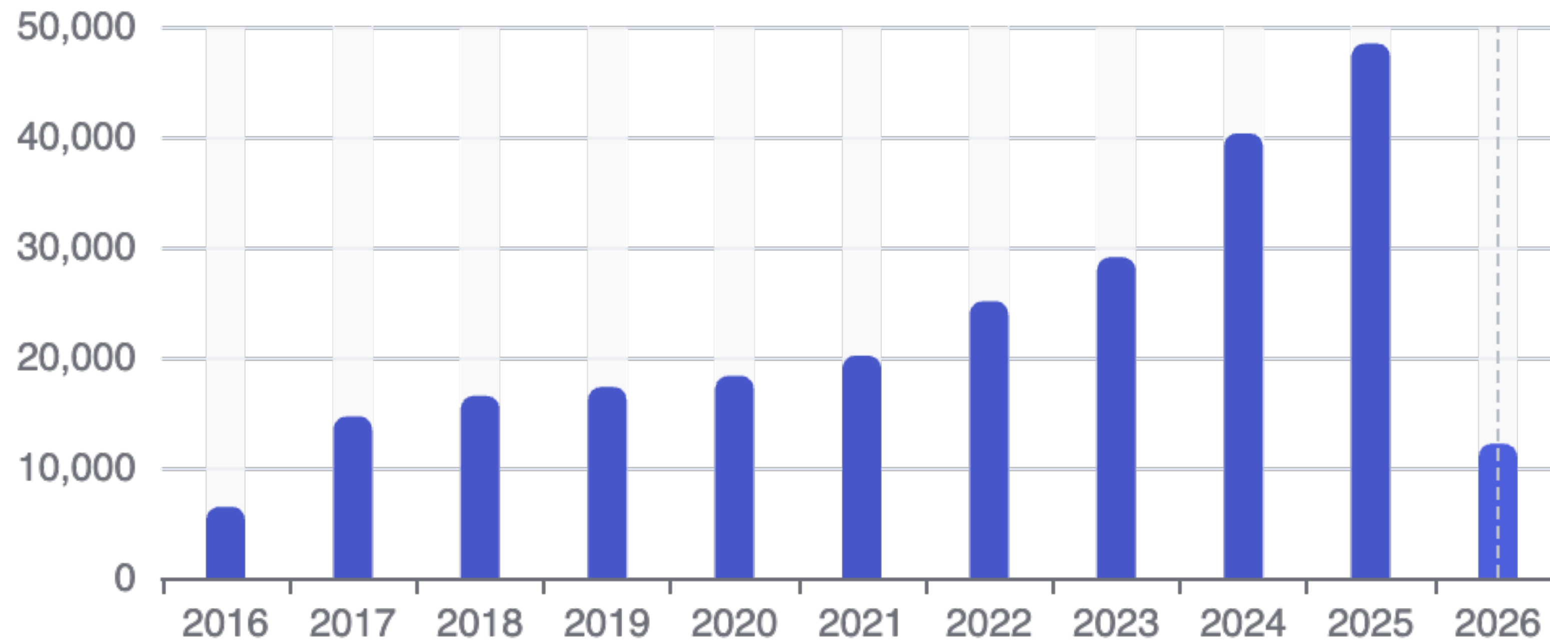
POSTECH

- Approaches
  - Write memory-safe code (Lecture 03)
  - Inspect code or binary for vulnerabilities and fix them (Lecture 05)
  - Add system- or compiler- level mitigations (Lecture 07)
    - NX (OS + MMU)
    - ASLR (OS)
    - Stack Canaries (Compiler)

Defending against memory safety vulnerabilities

POSTECH

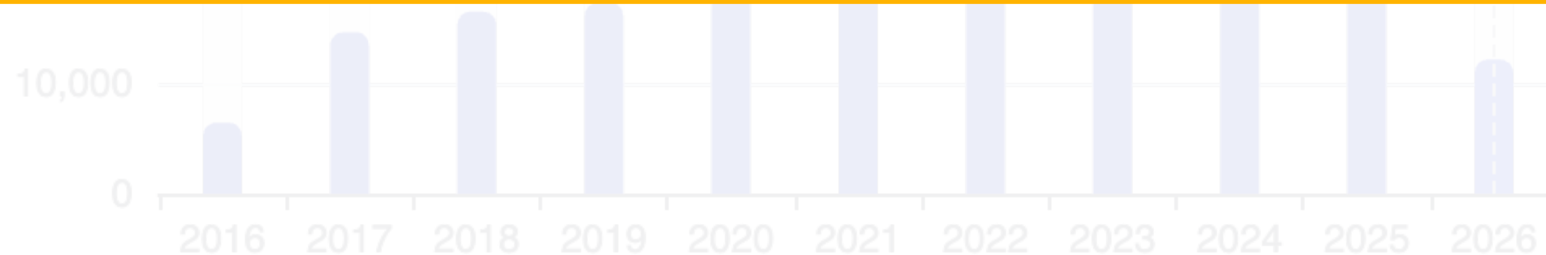
- Despite all these efforts, vulnerabilities continue to emerge



https://www.cvedetails.com/browse-by-date.php

Defending against memory safety vulnerabilities

- Despite all these efforts, vulnerabilities continue to emerge



Instead of mitigating bugs,
can we eliminate them altogether?

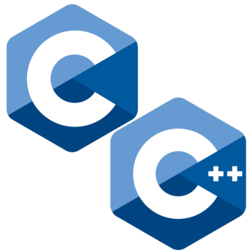
https://www.cvedetails.com/browse-by-date.php

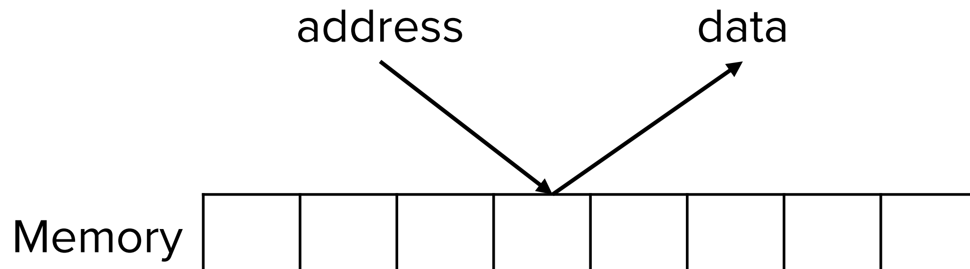
Memory-Safe Languages

Memory-unsafe vs. -safe languages

POSTECH

Memory-unsafe languages



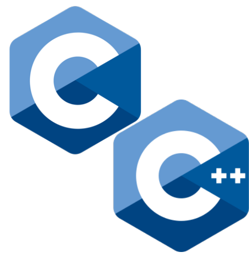


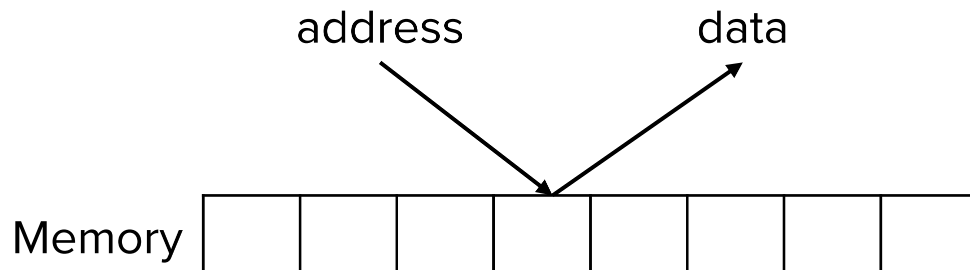
No abstraction of memory access

Memory-unsafe vs. -safe languages

POSTECH

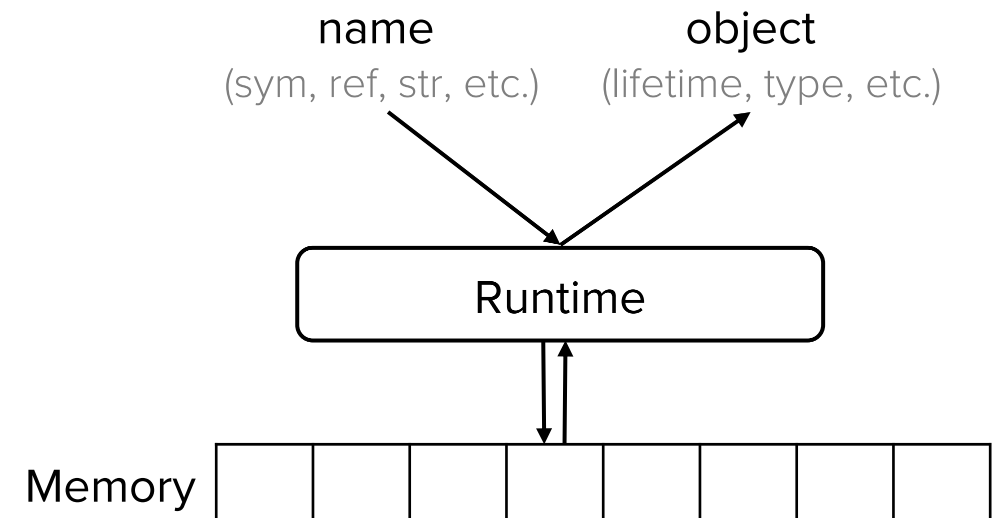
Memory-unsafe languages





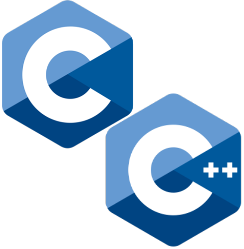
No abstraction of memory access

Memory-safe languages



Object-based abstraction of low-level memory

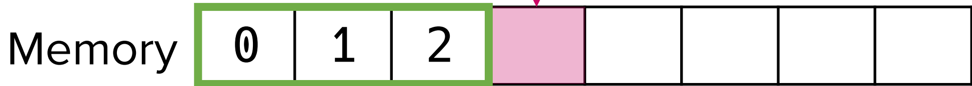
Memory-safety bugs – Spatial bugs



```c
char A[3] = {0, 1, 2};
A[3] = 3;
```

```c
*(&A[0] + 3) = 3
```

Memory



&A[0]

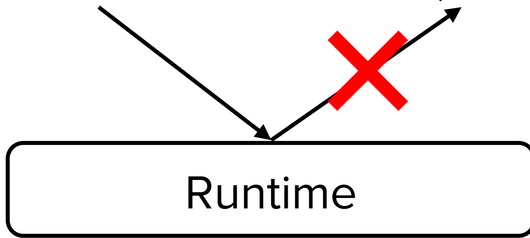
<Undefined behavior>

```python
A = [0, 1, 2]
A[3] = 3
```

```python
list.__setitem__(A, 3, 3)
```

symbol: A

object:
list, len=3, [0, 1, 2]

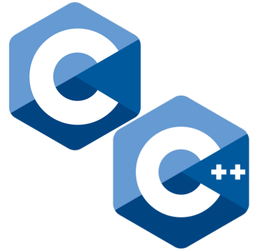


Runtime

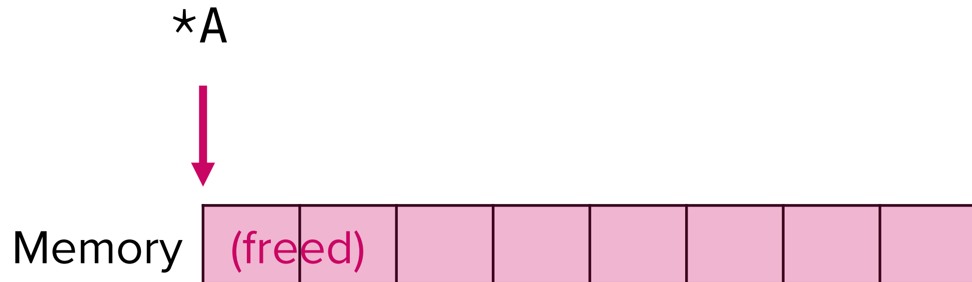
<Terminate>

IndexError: list assignment index out of range

Memory-safety bugs – Temporal bugs

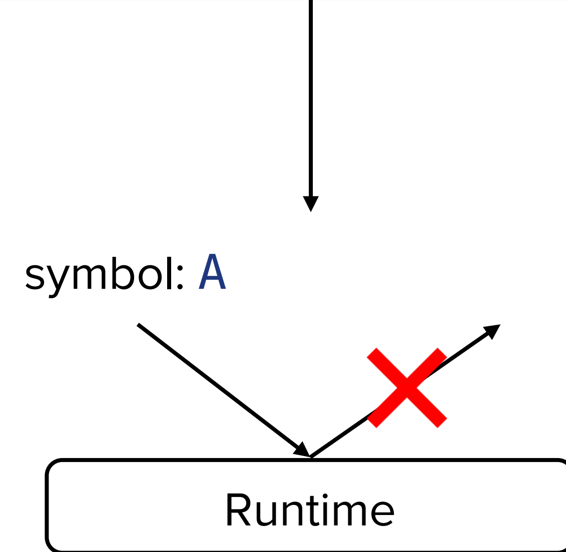


```
A = malloc(64);
free(A);
*A;
```



<Undefined behavior>

```
A = [0, 1, 2]
del A
A
```



<Terminate>

NameError: name ‘A’ is not defined

Memory-safe languages

POSTECH

- Memory-Safe languages enforce runtime checks that prevent invalid memory accesses
  - e.g., bounds checking, managed references, lifetime checking, …
- By design, they eliminate common memory-safety bugs
  - e.g., buffer overflows, use-after-free, double free
- Examples: Python, C#, Go, Rust, Java
  - Most modern languages except C and C++

So why don't we use memory-safe languages everywhere?

Memory-safe languages are slow

POSTECH

- Major criticism for memory-safe languages: performance
- Example: Memory allocation latency
  - C/C++:
    - Memory allocation (malloc/free) is fast, direct, and predictable
  - Python, Java, JavaScript, … (memory-safe languages):
    - Allocate and access memory via abstraction (instantiation of an object)
    - The garbage collector may pause execution unpredictably and introduce substantial latency
      (can be problematic for operating systems and real-time systems)

Memory-safe languages are slow

POSTECH

- How much slower?
  - Language speed benchmark:
    - https://programming-language-benchmarks.vercel.app/problem/helloworld

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Hello world!");
  } else {
    printf("Hello world %s!", argv[1]);
  }
  return 0;
}
```

Hello world in C

```python
import sys

if __name__ == '__main__':
  if len(sys.argv) > 1:
    n = sys.argv[1]
  else:
    n = ''

  print(f'Hello world {n}!’)
```

Hello world in Python

Memory-safe languages are slow

POSTECH

- How much slower?
  - C vs. Python

lang	code	time	stddev	peak-mem	time(user)	time(sys)	compiler/runtime
c	1.c	1.0ms	0.1ms	1.3MB	0ms	0ms	zigcc 0.14.1
c	1.c	1.2ms	0.1ms	2.1MB	0ms	0ms	gcc 15.1.0
c	1.c	1.6ms				0ms	clang 14.0.0-1ubuntu1.1
python	1.py	11ms				0ms	pyston 3.8.12
python	1.py	13ms	0.8ms	8.8MB	0ms	0ms	cpython 3.13.5
python	1.py	29ms	0.8ms	52.2MB	10ms	10ms	pypy 3.11.13

~10x slowdown with Python!

Memory-safe languages are slow

POSTECH

- How much slower?
  - C vs. Java

lang	code	time	stddev	peak-mem	time(user)	time(sys)	compiler/runtime
c	1.c	1.0ms	0.1ms	1.3MB	0ms	0ms	zigcc 0.14.1
c	1.c	1.2ms	0.1ms	2.1MB	0ms	0ms	gcc 15.1.0
c	1.c	1.6ms	0.7ms	2.1MB	0ms	0ms	clang 14.0.0-1ubuntu1.1
java	1.java	62ms					graal/jvm 17.0.8
java	1.java	69ms	4.7ms	43.9MB	78ms	10ms	openjdk 21
java	1.java	73ms	1.9ms	44.7MB	74ms	16ms	openjdk 23
java	1.java	186ms	3.9ms	42.9MB	82ms	132ms	openjdk/zgc 21

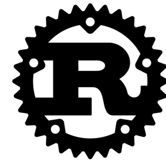
Java is worse. ~60x slowdown ☹

Memory-safe languages are slow

- How much slower?

Memory-safety indeed is important.
However, 10x to 60x slowdown cannot be justified!

Can memory-safe language be fast?



Rust

Rust to the rescue

- Rust
  - Performance comparable to C/C++
    - Key idea: Memory safety without garbage collection
  - Guarantees memory safety at compile time
    - If a program compiles, it is memory-safe
- Modern systems increasingly adopt Rust
  - Google: Android and infrastructure components in Rust
  - Microsoft: Rust used in Windows security components
  - Mozilla: Developed Rust for safe systems programming

Rust's performance is comparable to C/C++

POSTECH

- How comparable?
  - C vs. Rust

lang	code	time	stddev	peak-mem	time(user)	time(sys)	compiler/runtime
c	1.c	1.0ms	0.1ms	1.3MB	0ms	0ms	zigcc 0.14.1
c	1.c	1.2ms	[illegible]	[illegible]	[illegible]	0ms	gcc 15.1.0
rust	1.rs	1.2ms	[illegible]	[illegible]	[illegible]	0ms	rustc 1.88.0
rust	1.rs	1.2ms	0.1ms	1.9MB	0ms	0ms	rustc 1.90.0-nightly
c	1.c	1.6ms	0.7ms	2.1MB	0ms	0ms	clang 14.0.0-1ubuntu1.1

Negligible performance loss!

How does Rust guarantee memory safety with low overhead?

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Ownership:
    - Each value in Rust has an **owner** and there can be **one owner at a time**
    - When the owner goes out of scope, the value is dropped

```
{
   let mut x = String::from("Hello!");

   let y = String::from("World!");

   x = y;
}
```

x owns "Hello!"

y owns "World!"

Ownership transfer: x now owns "World!" and doesn't own "Hello!" anymore

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Ownership (~= memory management)
    - Each value in Rust has an **owner** and there can be **one owner at a time**
    - When the owner goes out of scope, the value is dropped

```
{
  // x <- alloc("Hello!")
  let mut x = String::from("Hello!");
  // y <- alloc("World!")
  let y = String::from("World!");
  // drop(x): free("Hello!")
  x = y;
}
// drop(x): free("World!")
```

Assignment → Memory allocation

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Ownership (~= memory management)
    - Each value in Rust has an **owner** and there can be **one owner at a time**
    - When the owner goes out of scope, the value is dropped

```
{
  // x <- alloc("Hello!")
  let mut x = String::from("Hello!");
  // y <- alloc("World!")
  let y = String::from("World!");
  // drop(x): free("Hello!")
  x = y;
}
// drop(x): free("World!")
```

Assignment → Memory allocation

Transfer → Memory deallocation

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Ownership (~= memory management)
    - Each value in Rust has an **owner** and there can be **one owner at a time**
    - When the owner goes out of scope, the value is dropped

```
{
   // x <- alloc("Hello!")
   let mut x = String::from("Hello!");
   // y <- alloc("World!")
   let y = String::from("World!");
   // drop(x): free("Hello!")
   x = y;
}
// drop(x): free("World!")
```

Assignment → Memory allocation

Transfer → Memory deallocation

x goes out of scope → Memory deallocation

Key ideas of Rust's memory safety

- Ownership and borrowing
  - Ownership (~= memory management)
    - Each value in Rust has an **owner** and there can be **one owner at a time**

Users don't need to manually manage resources. The compiler automatically inserts the necessary memory management operations.

```
    let y = String::from("World!");
    // drop(x): free("Hello!")
    x = y;
}
// drop(x): free("World!")
```

Transfer → Memory deallocation

x goes out of scope → Memory deallocation

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Ownership system is great for resource management
  - However, it can impose too much burden on developers
    - e.g., Any access of x results in compilation error

```
{
  let x = String::from("Hello!");

  let y = x;

  x;
}
```

x owns "Hello!"

y now owns "Hello!"

Compilation error: x does not own any value!

→ In practice, even basic function calls become cumbersome under strict ownership ☹

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - A variable can **borrow** a value from an owner

```
{
  let x = String::from("Hello!");

  let y = &x;

  x;
}
```

x owns "Hello!"

y borrows a value from x
(y owns a shared reference to x)

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #1**: There can exist multiple shared references
    - Shared references allow concurrent reading

```
{
  let x = String::from("Hello!");
  let y = &x;
  let z = &x;
  x; // "Hello!"
  y; // "Hello!"
  z; // "Hello!"
}
```

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #2**: There should exist only one mutable reference
    - Mutable references allow writing

```
{
  let mut x = String::from("Hello!");
  let y = &mut x;
  let z = &mut x;
}
```

Compilation error: more than one mutable references exist

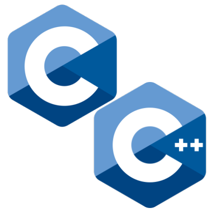
Key ideas of Rust's memory safety

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #1**: There can exist multiple shared references
    - **Rule #2**: There should exist only one mutable reference
    - **Rule #3 (Safety rule)**: Rule #1 **xor** Rule #2 is applicable
      - We can create many shared references or one mutable reference, but we cannot create both at the same time!

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #3 (Safety rule)**: Rule #1 **xor** Rule #2
      - Key to enabling secure and performant APIs
      - Example 1: Stale iterator



```
vector<int> v = {...};
for (auto it = v.begin(); it != v.end(); it++) {
  v.push_back(-1);
}
```

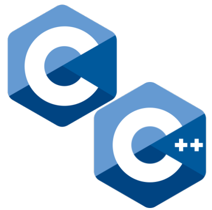
What's the issue?

If a C++ container is modified during iteration,
existing iterators become invalid,
leading to undefined behavior (potential memory safety issues) when dereferenced

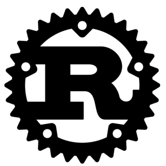
Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #3 (Safety rule)**: Rule #1 **xor** Rule #2
      - Key to enabling secure and performant APIs
      - Example 1: Stale iterator



```
vector<int> v = {...};
for (auto it = v.begin(); it != v.end(); it++) {
  v.push_back(-1);
}
```



```
let mut v = vec!{...};
for it in v.iter() {
  v.push(-1);
}
```

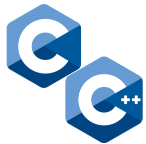
Compilation error: cannot borrow 'v' as mutable ...

The iterator `it` is a shared reference created by `v.iter()`.
`v.push()` requires a mutable borrow of v → Violation of Rule #3.

Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #3 (Safety rule)**: Rule #1 **xor** Rule #2
      - Key to enabling secure and performant APIs
      - Example 2: Aliasing



```
void *memcpy(void *dest, void *src, size_t n);
```

If `dest` and `src` overlap, it may lead to memory safety issues.
However, C compilers do not check for overlapping pointers.
→ It is user's responsibility to follow `memcpy( )`'s safety semantics.

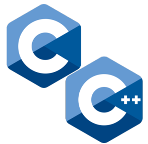
```
MEMCPY(3)                                   Linux Programmer's Manual

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The memcpy() function copies n bytes from memory area src to memory area dest.  The memory areas must not overlap.
```

(`$ man memcpy`)

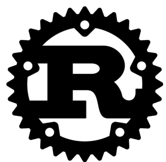
Key ideas of Rust's memory safety

POSTECH

- Ownership and borrowing
  - Borrowing (a.k.a. references)
    - **Rule #3 (Safety rule)**: Rule #1 **xor** Rule #2
      - Key to enabling secure and performant APIs
      - Example 2: Aliasing



```
void *memcpy(void *dest, void *src, size_t n);
```



mutable ref   shared ref

```
fn clone_from_slice(&mut self, src: &[T]);
```

mutable and shared reference for a memory cannot exist at the same time.
→ src and dest never point to overlapping memory region.

Language Landscape

POSTECH

Language	Memory-safe?	No garbage collector?	Systems-level control?
C/C++	❌	✅	✅
Python/Java	✅	❌	❌
Go	✅	❌	Limited
Rust	✅	✅	✅

→ Great! Let's use Rust everywhere, stay safe and performant!

Reality check

POSTECH

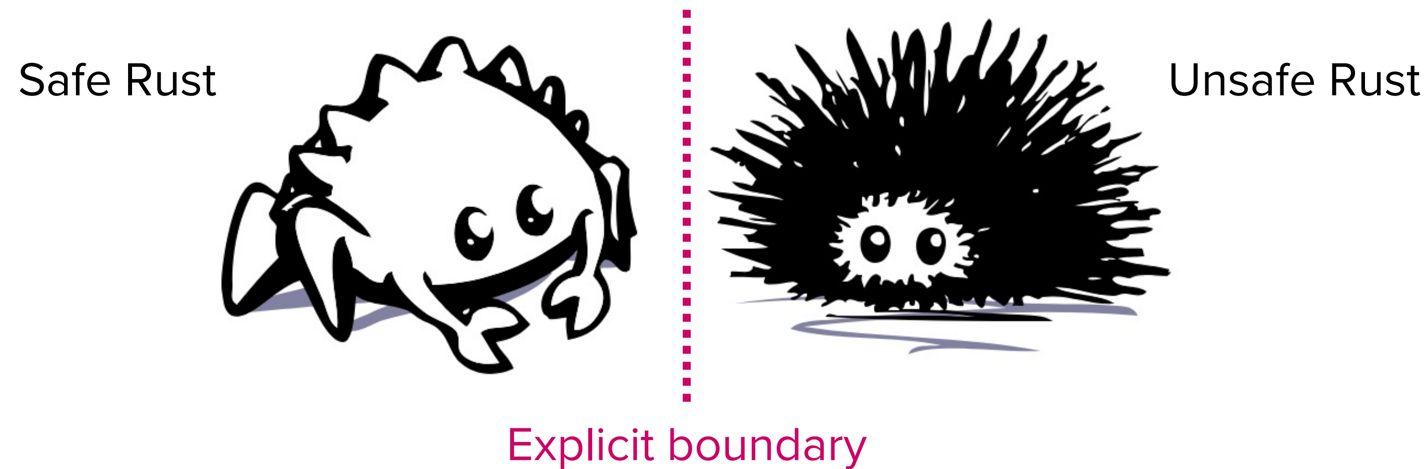
- Is it really possible to build system software with Rust?
- System software typically do:
  - Memory-mapped I/O
  - Hardware abstraction
  - OS interaction (e.g., system calls)

  → Low-level operations cannot be verified by the Rust compiler

Then, system software utilizing these operations will not compile

Solution: Unsafe Rust

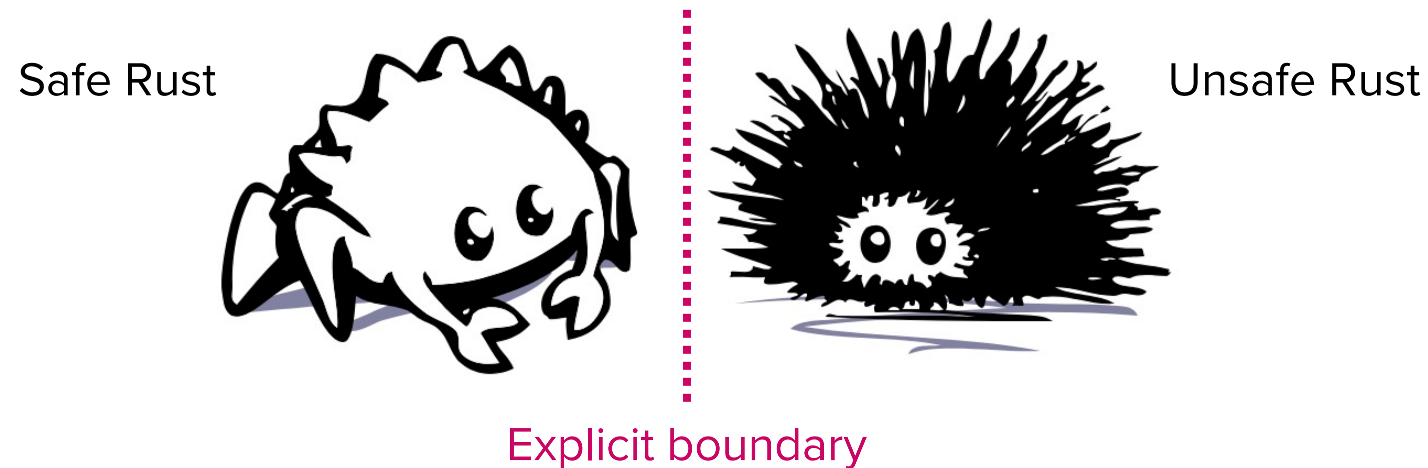
- Rust is a combination of Safe Rust and Unsafe Rust



If a program is written entirely in Safe Rust, the Rust compiler automatically guarantees memory safety

Solution: Unsafe Rust

- Rust is a combination of Safe Rust and Unsafe Rust

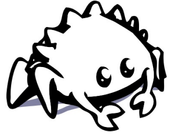


If a program is written entirely in Safe Rust, the Rust compiler automatically guarantees memory safety

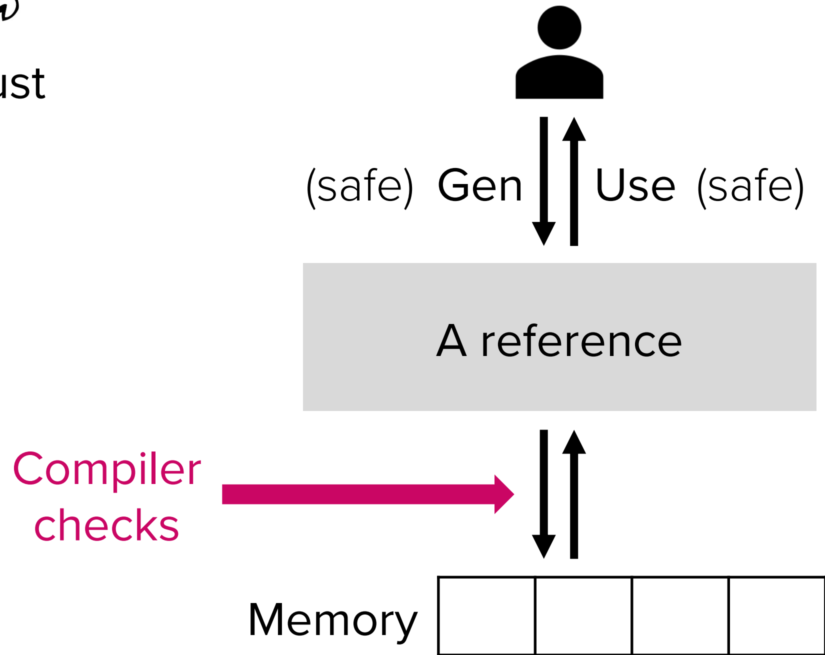
If a program contains Unsafe Rust, the programmer needs to guarantee memory safety (just like C/C++)

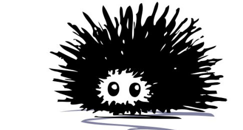
Explicit safety control of Rust

POSTECH

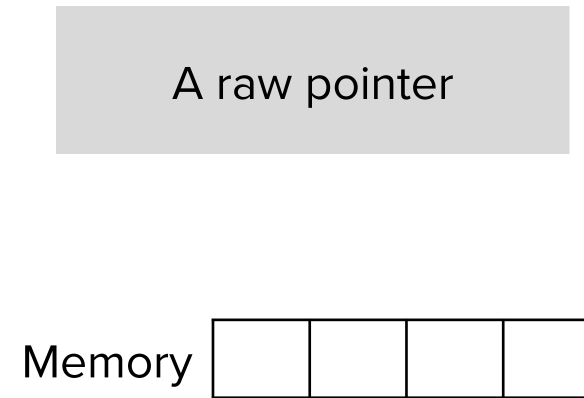


Safe Rust





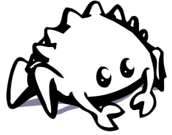
Unsafe Rust



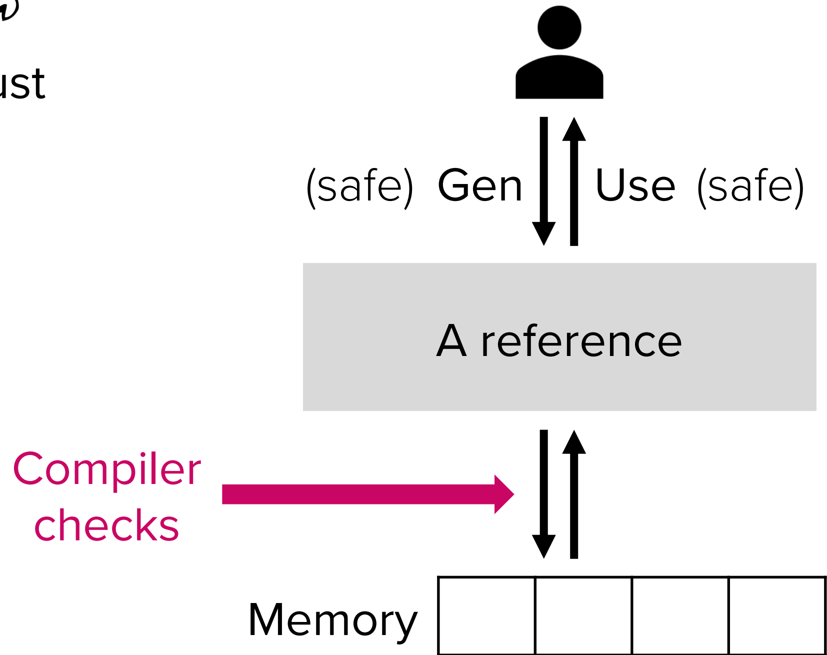
Safety is guaranteed by the compiler

Safety is guaranteed by the developer

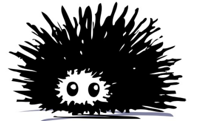
Explicit safety control of Rust



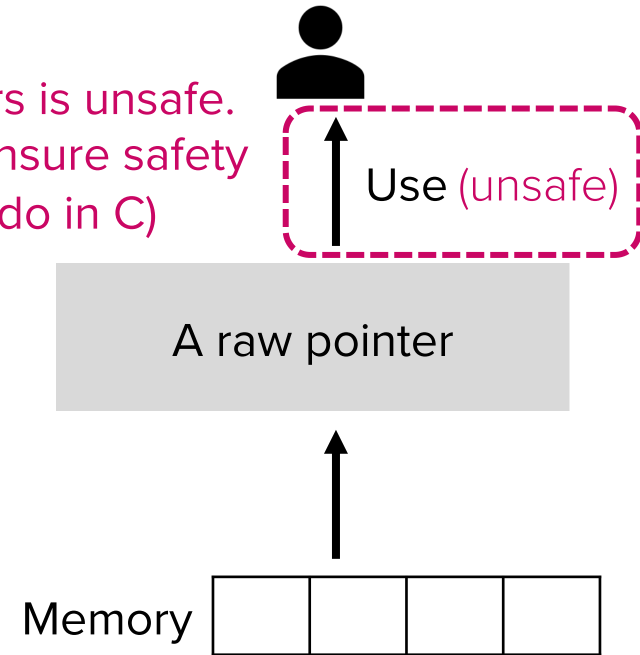
Safe Rust



Using raw pointers is unsafe.
Developer must ensure safety
(like what we do in C)



Unsafe Rust

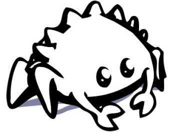


Safety is guaranteed by the compiler

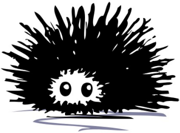
Safety is guaranteed by the developer

Explicit safety control of Rust

POSTECH



Safe Rust



Unsafe Rust

Generating raw pointers is considered safe
(as far as they are not used)

(safe) Gen

(safe) Gen | Use (safe)

A reference

Compiler checks

Memory

(safe) Gen | Use (unsafe)

A raw pointer

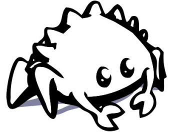
Memory

Safety is guaranteed by the compiler

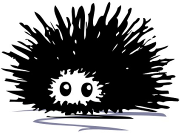
Safety is guaranteed by the developer

Explicit safety control of Rust

POSTECH



Safe Rust



Unsafe Rust

(safe) Gen

(safe) Gen | Use (safe)

A reference

Compiler checks

Memory

(safe) Gen | Use (unsafe)

A raw pointer

Memory

(unsafe) Gen

Creating reference for raw pointers is unsafe

Safety is guaranteed by the compiler

Safety is guaranteed by the developer

Two ways to use Unsafe in Rust

POSTECH

- Unsafe APIs:

```
unsafe fn get_unchecked<T, I>(v: &T, index: I) -> &T::Output
```

→ The caller must ensure that `index` is in bounds of the underlying container.

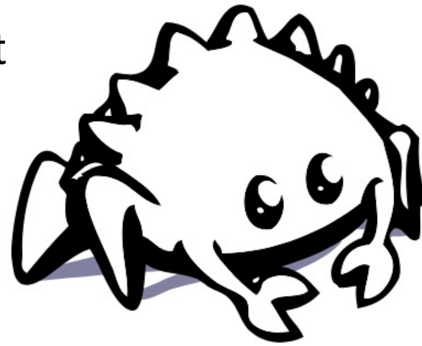
- Safe APIs:

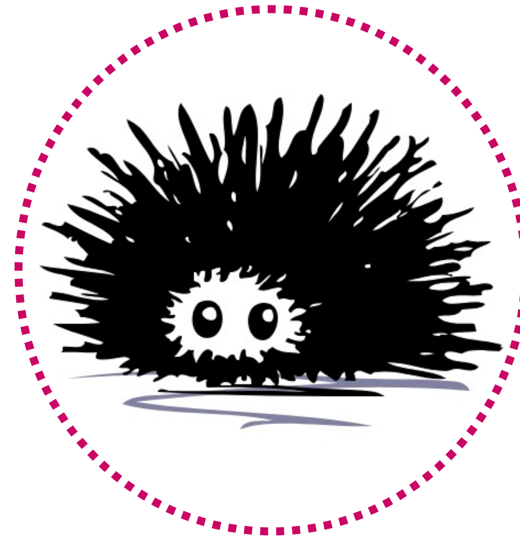
```
fn access(index: usize) {
  assert!(index < self.len());
  unsafe { get_unchecked(index); }
}
```

→ API designers are responsible for memory safety

Rethinking memory safety of Rust

Safe Rust





Unsafe Rust

Memory safety of a Rust program depends on the correctness of all unsafe code it contains!

and that means..

Vulnerabilities in Rust

POSTECH

- CVE-2025-68260:
  - One of the first reported vulnerabilities in the Linux kernel after Rust integration

Linux Kernel Rust Code Sees Its First CVE Vulnerability

Written by Michael Larabel in Linux Kernel on 17 December 2025 at 07:44 AM EST. 114 Comments

The first CVE vulnerability has been assigned to a piece of the Linux kernel's Rust code.

Greg Kroah-Hartman announced that the first CVE has been assigned to a piece of Rust code within the mainline Linux kernel.

This first CVE for Rust code in the Linux kernel pertains to the Android Binder rewrite in Rust. There is a race condition that can occur due to some noted unsafe Rust code. That code can lead to memory corruption of the previous/next pointers and in turn cause a crash.

This CVE for the possible system crash is for Linux 6.18 and newer since the introduction of the Rust Binder driver. At least though it's just a possible system crash and not any more serious system compromise with remote code execution or other more severe issues.

More details on CVE-2025-68260 via the Linux CVE mailing list.

Case study: CVE-2025-68260

POSTECH

- Vulnerability appeared in Android Binder
  - Binder?
    - The core Inter-Process Communication (IPC) mechanism in Android
    - When an app wants to call another app or system service, it uses Binder
  - Death notification
    - In Binder, when a service process crashes unexpectedly, all clients depending on it should be notified via death notification
    - `NodeDeath` struct represents such death notifications
    - `death_list` is a linked list storing all death notifications registered to a particular Binder node

Case study: CVE-2025-68260

POSTECH

- Vulnerability
  - When cleaning up a **`NodeDeath`**, the code needs to remove it from the associated **`death_list`**
  - The vulnerable removal operation:

```
// SAFETY: A `NodeDeath` is never inserted into the death_list
// of any node other than its owner, so it is either in this
// death list or in no death list.
unsafe { node_inner.death_list.remove(self) };
```

The comment above shows the developer's safety assumption

Problem: This assumption does not hold in concurrent scenarios

Case study: CVE-2025-68260

- Vulnerable thread interleaving

(Another unsafe API for cleaning up `death_list`)

Thread A: `remove()`

```
unsafe {
  node_inner.death_list.remove(self)
};
```

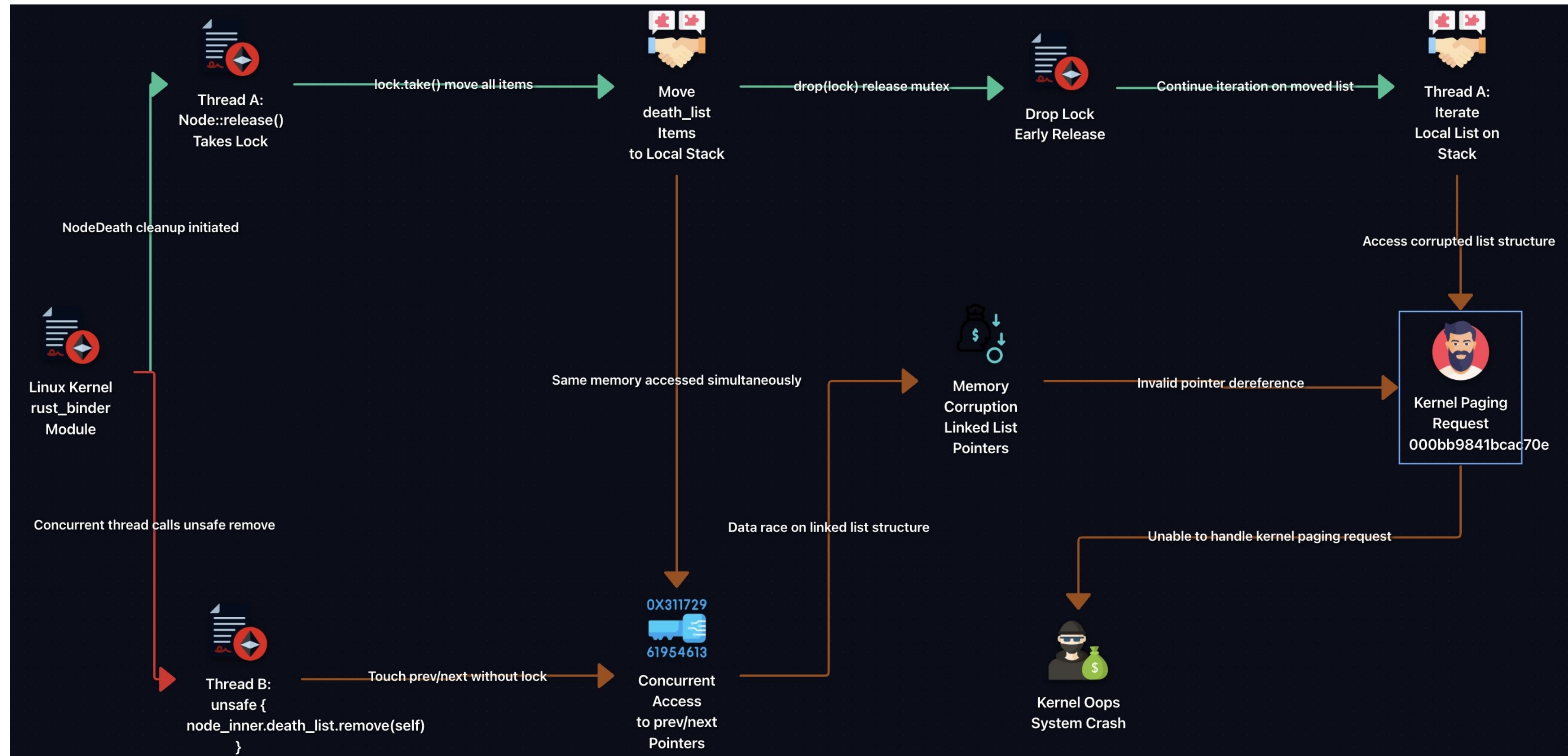
Thread B: `release()`

1. Acquire lock
2. Move all items from `death_list` to a temporary list on the stack
3. Release the lock
4. Iterate through the temporary list, processing each node

Two pointers exist for a single object!

Case study: CVE-2025-68260

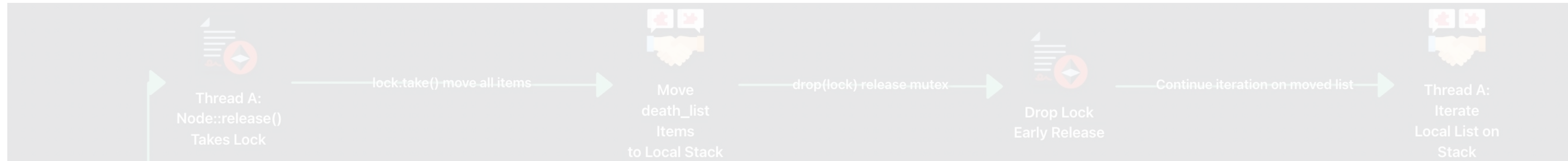
- Race condition (Iamge credit: x.com/forefy)



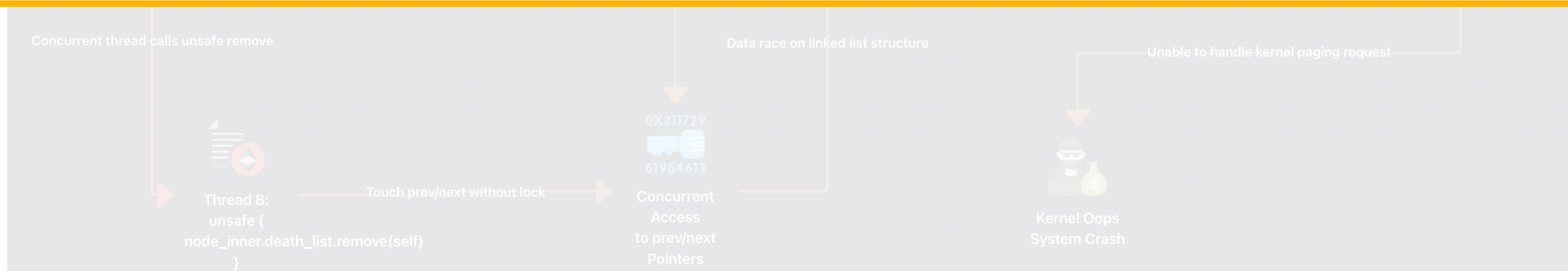
No ownership checking due to unsafe

Case study: CVE-2025-68260

- Race condition (Iamge credit: x.com/forefy)



Read the patch for further details!



No ownership checking due to unsafe

Lessons learned

POSTECH

- Rust's memory safety promise:
  - If your code is 100% safe Rust, it won't have memory safety issues
  - The correctness of unsafe code depends on whether your maintained invariants are correct. If they are, the encapsulated safe API is safe

Language Landscape (Revisited)

POSTECH

Language	Memory-safe?	No garbage collector?	Systems-level control?
C/C++	❌	✅	✅
Python/Java	✅	❌	❌
Go	✅	❌	Limited
Rust	✅ + ❌	✅	✅

→ Rust provides strong, compiler-enforced memory safety, but unsafe code can still violate safety guarantees

Parts 1 and 2 of CSED415 are over

POSTECH

- Attacks typically begin with a memory safety issue
  - Attack surface analysis (Lecture 02)
  - Buggy code is the root of evil (Lecture 03)
- Mitigations exist, but they are not perfect
  - They are designed in a way that their impact on CIA is minimal
  - There is a trade-off between cost and effectiveness

Primary focus so far: System Integrity (ref: Lecture 02)
"A system performs its intended function in an unimpaired manner."

Question: How can we preserve Confidentiality?

Coming up next

POSTECH

- Cryptographic primitives and their applications

Questions?