Rust, TEE, and OS

Mingshen

Questions

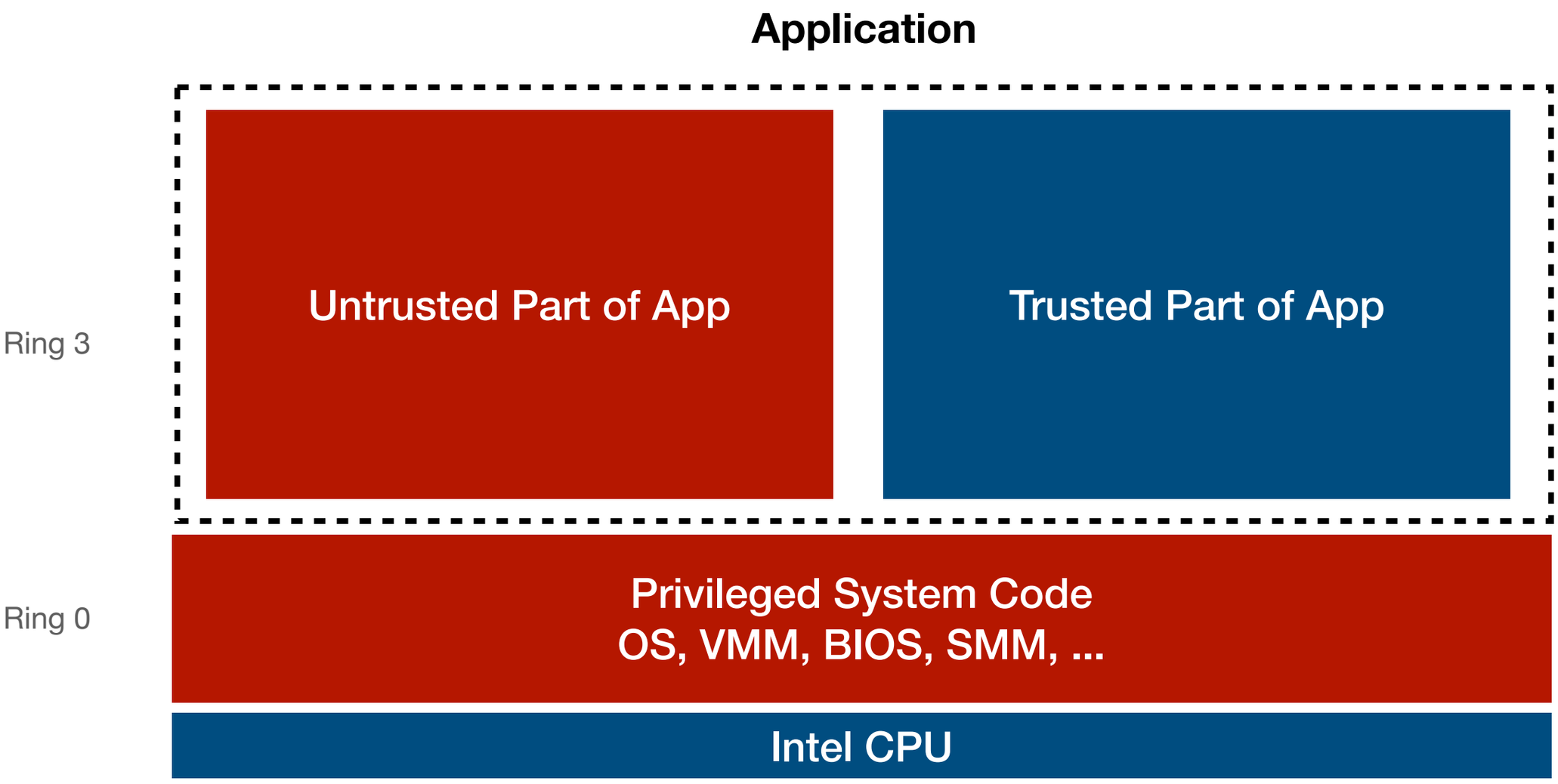
- What is TEE?
- Why having an OS in TEE?
- Does language matter?

TEE

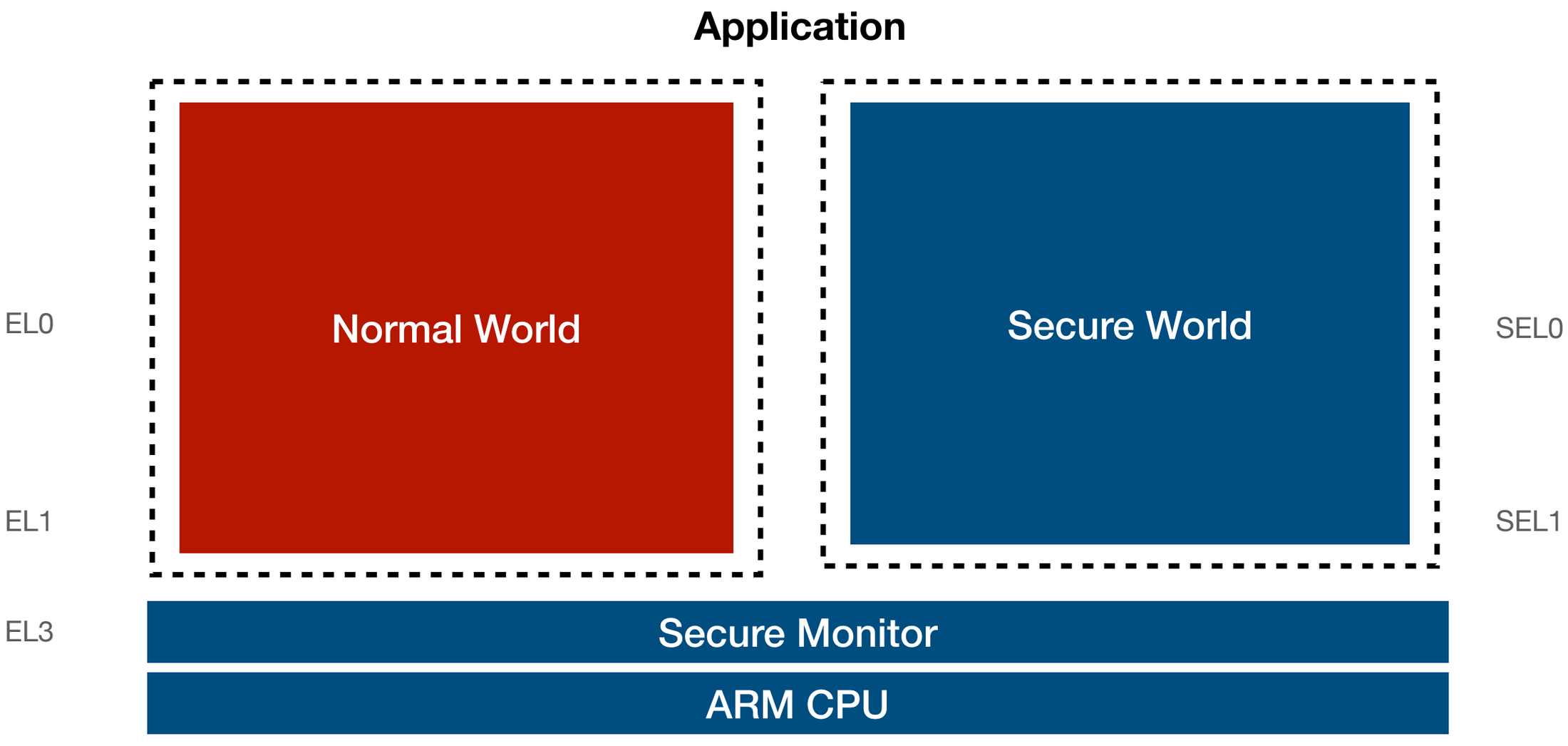
TEE Implementations

- Intel SGX
- AMD SEV
- ARM TrustZone, CCA
- RISC-V Keystone, Penglai

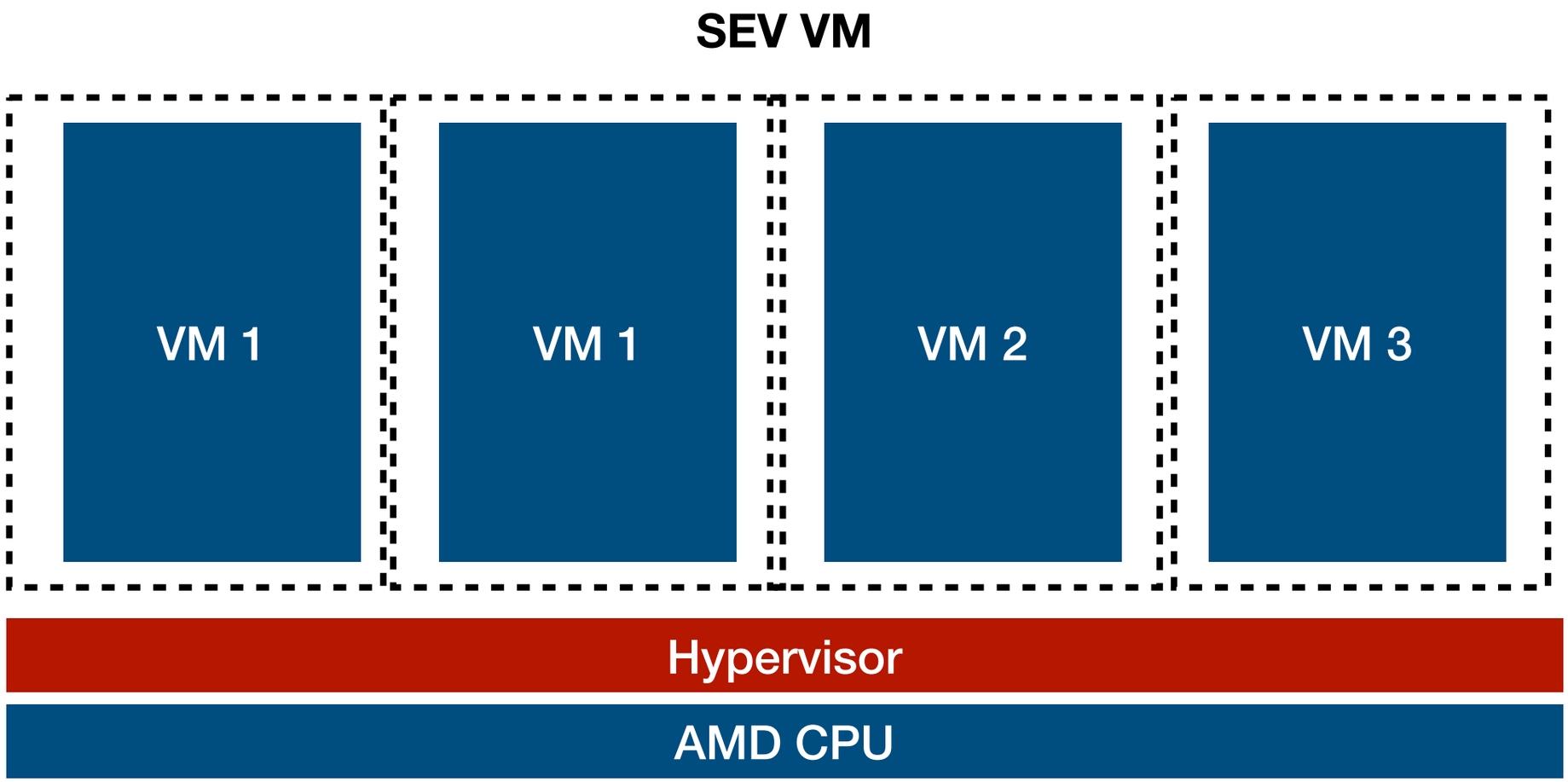
Intel SGX


Application
Untrusted Part of App
Trusted Part of App
Ring 3
Ring 0
Privileged System Code
OS, VMM, BIOS, SMM, ...
Intel CPU

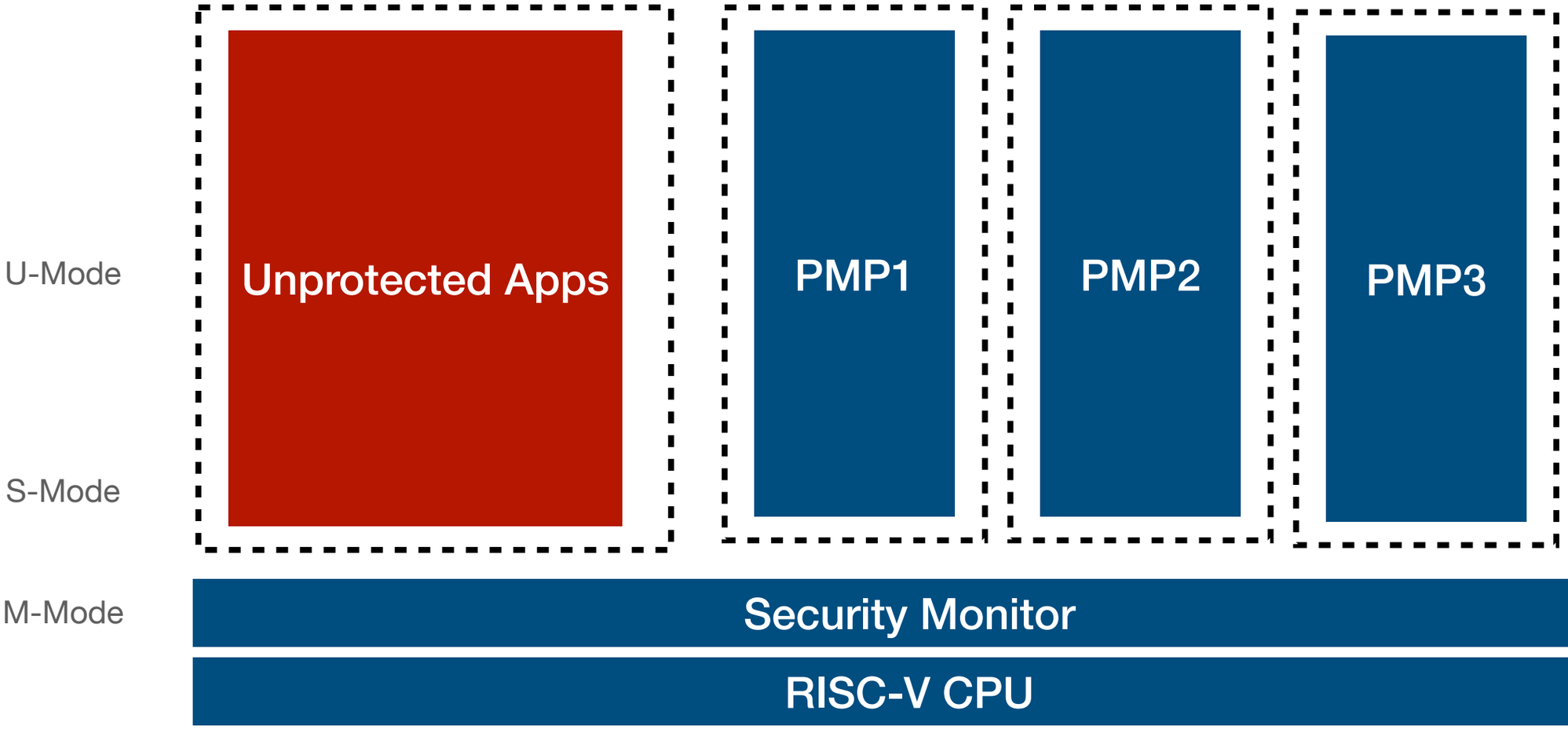
ARM TrustZone



AMD SEV



RISC-V PMP


U-Mode
S-Mode
M-Mode
Unprotected Apps
PMP1
PMP2
PMP3
Security Monitor
RISC-V CPU

TEE

Data Confidentiality

Data Integrity

Code Integrity

Code Confidentiality

Authenticated Launch

Programmability

Attestability

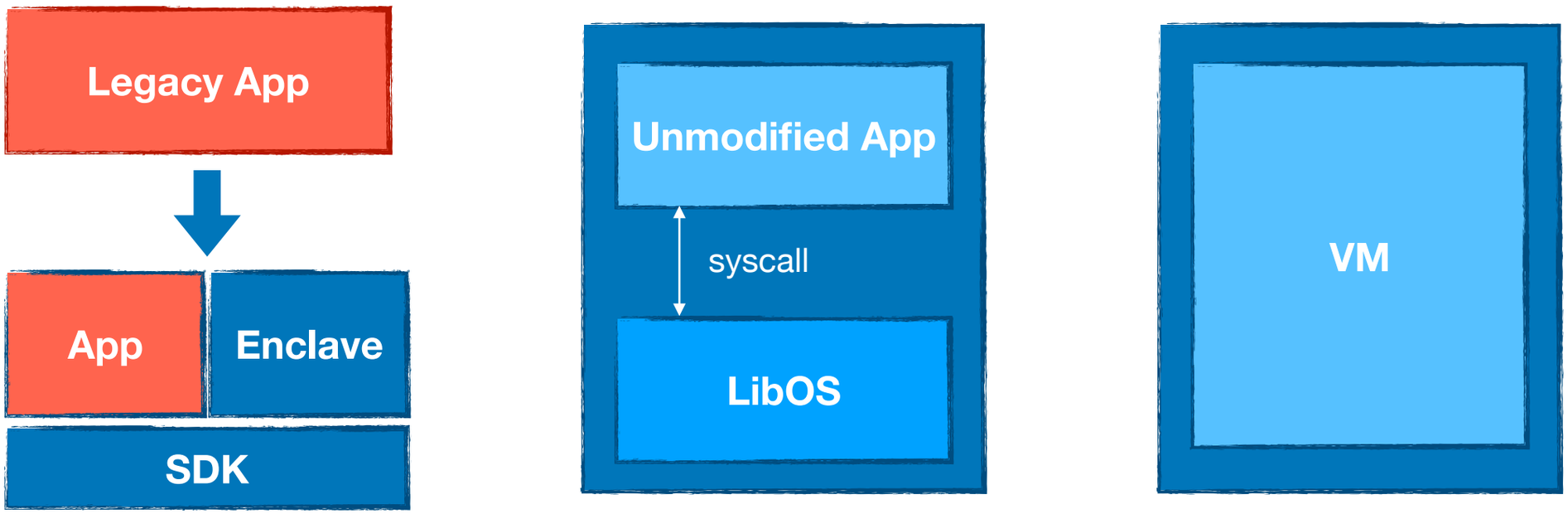
Recoverability

Applications of TEE

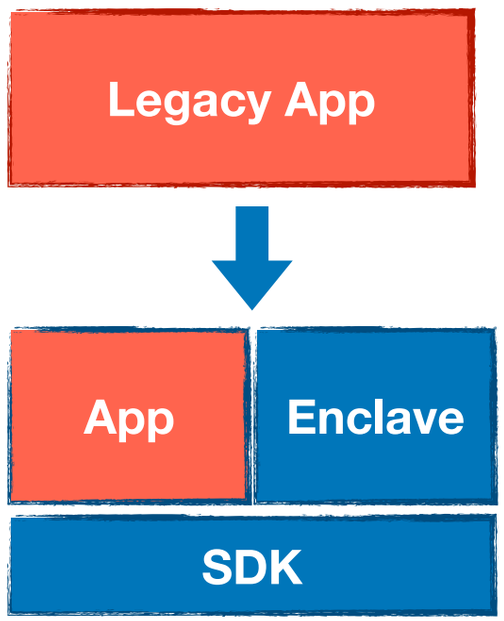
- Digital Right Management (DRM)
- Biometrics Authentication
- **Privacy-Preserving Computation**

Confidential Computing

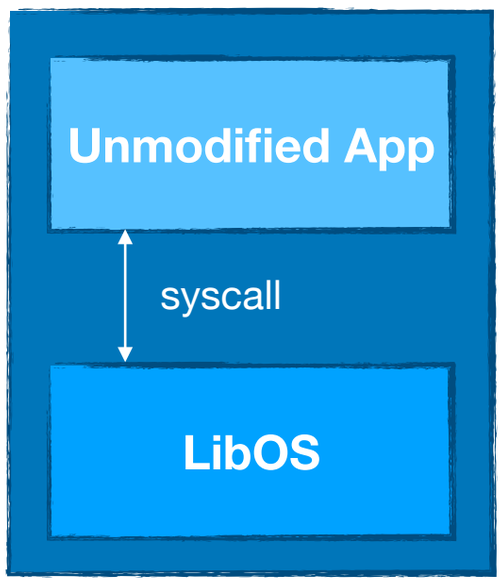
TEE Development Model



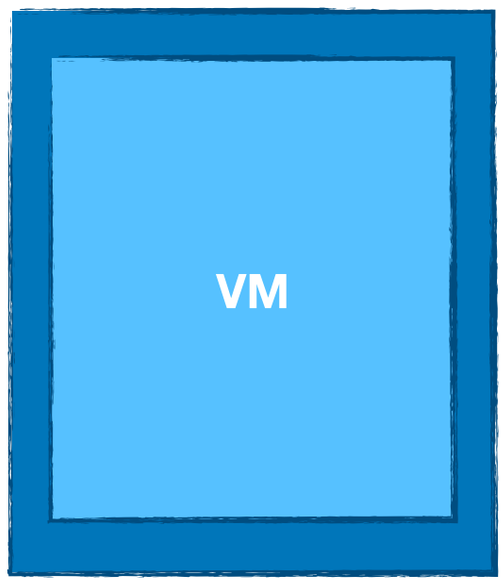
TEE Development Model



+ small TCB
- difficult to develop



+ medium TCB
- easy to develop



+ large TCB
- easy to develop

Security vs Usability

Does programming language matter?

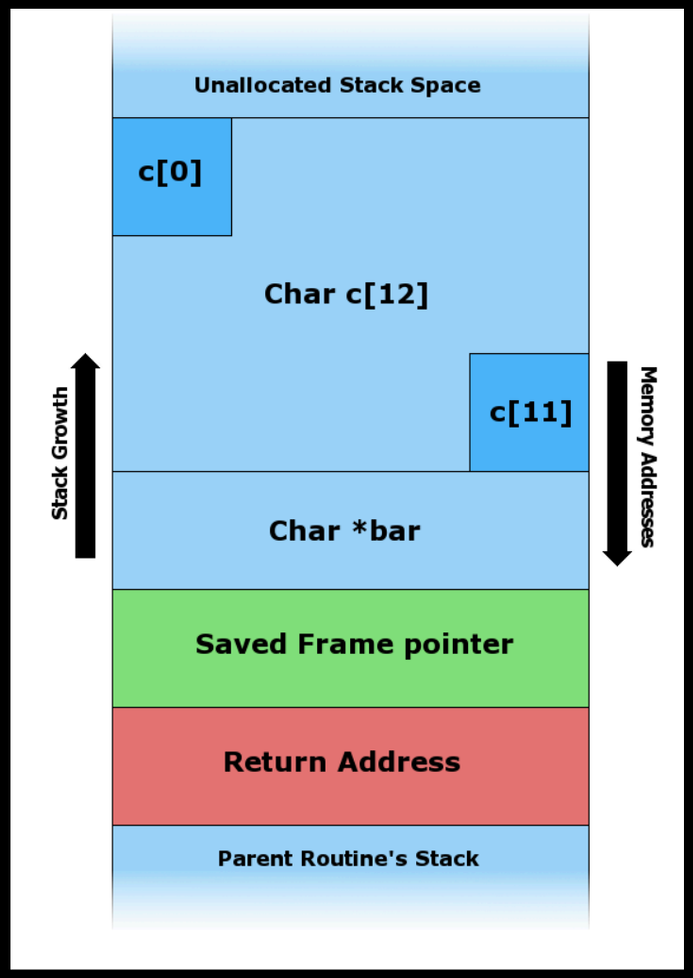
- Yes. Memory safety in TEE is extremely important!

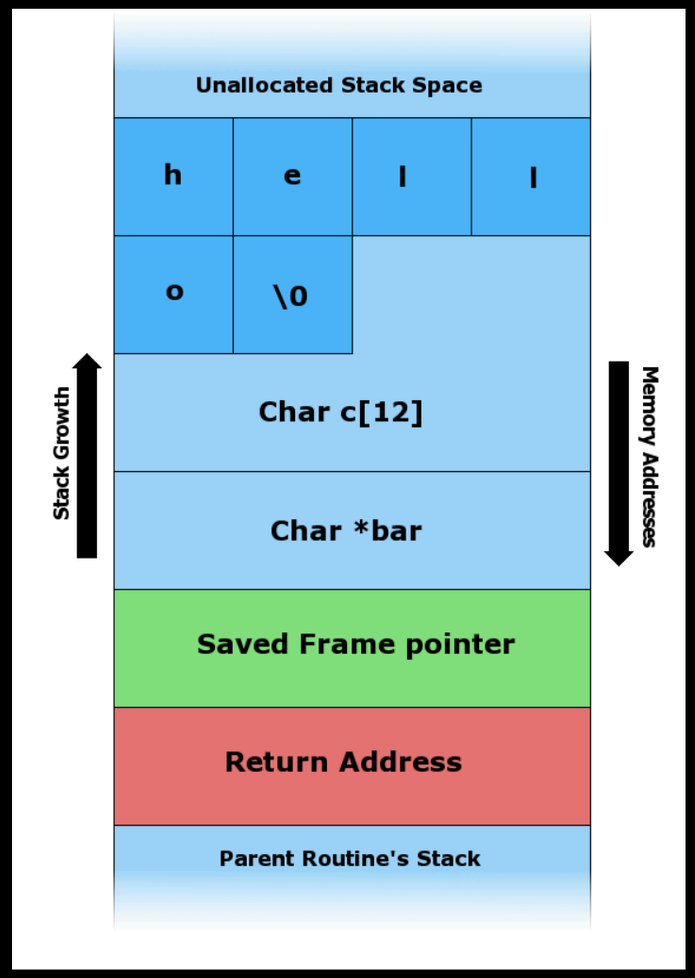
Memory Safety

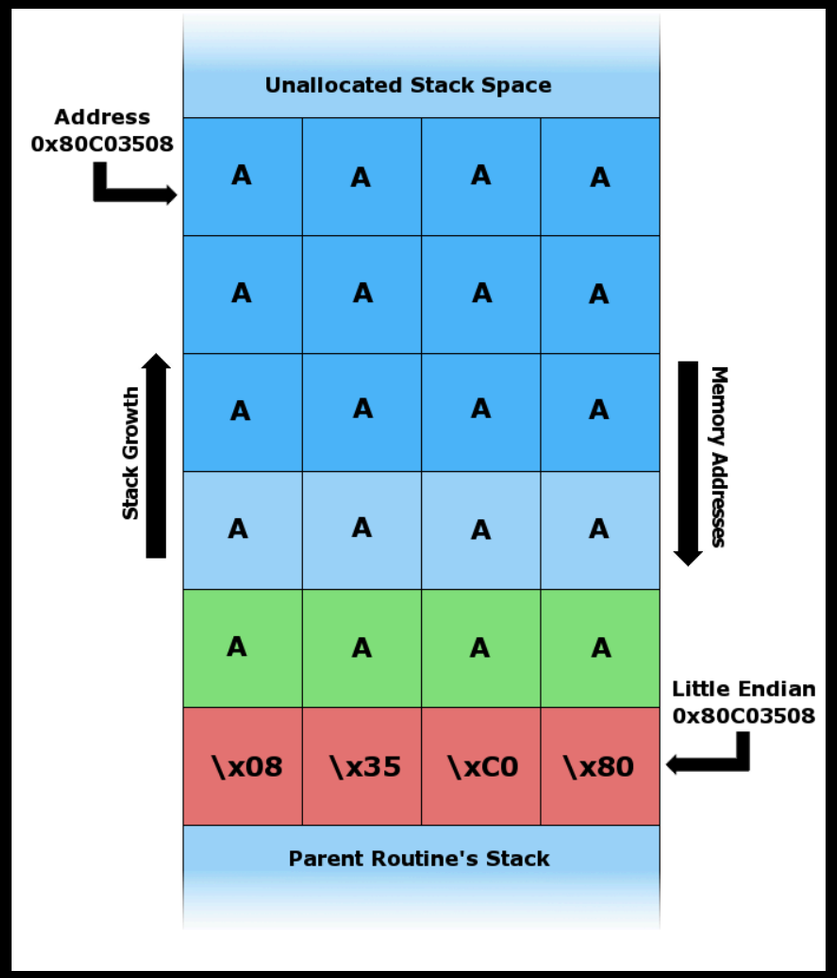
- **Memory corruption** occurs in a computer program when the contents of a memory location are **unintentionally modified**; this is termed violating memory safety.

- **Memory safety** is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as **buffer overflows and dangling pointers**.

Stack Buffer Overflow





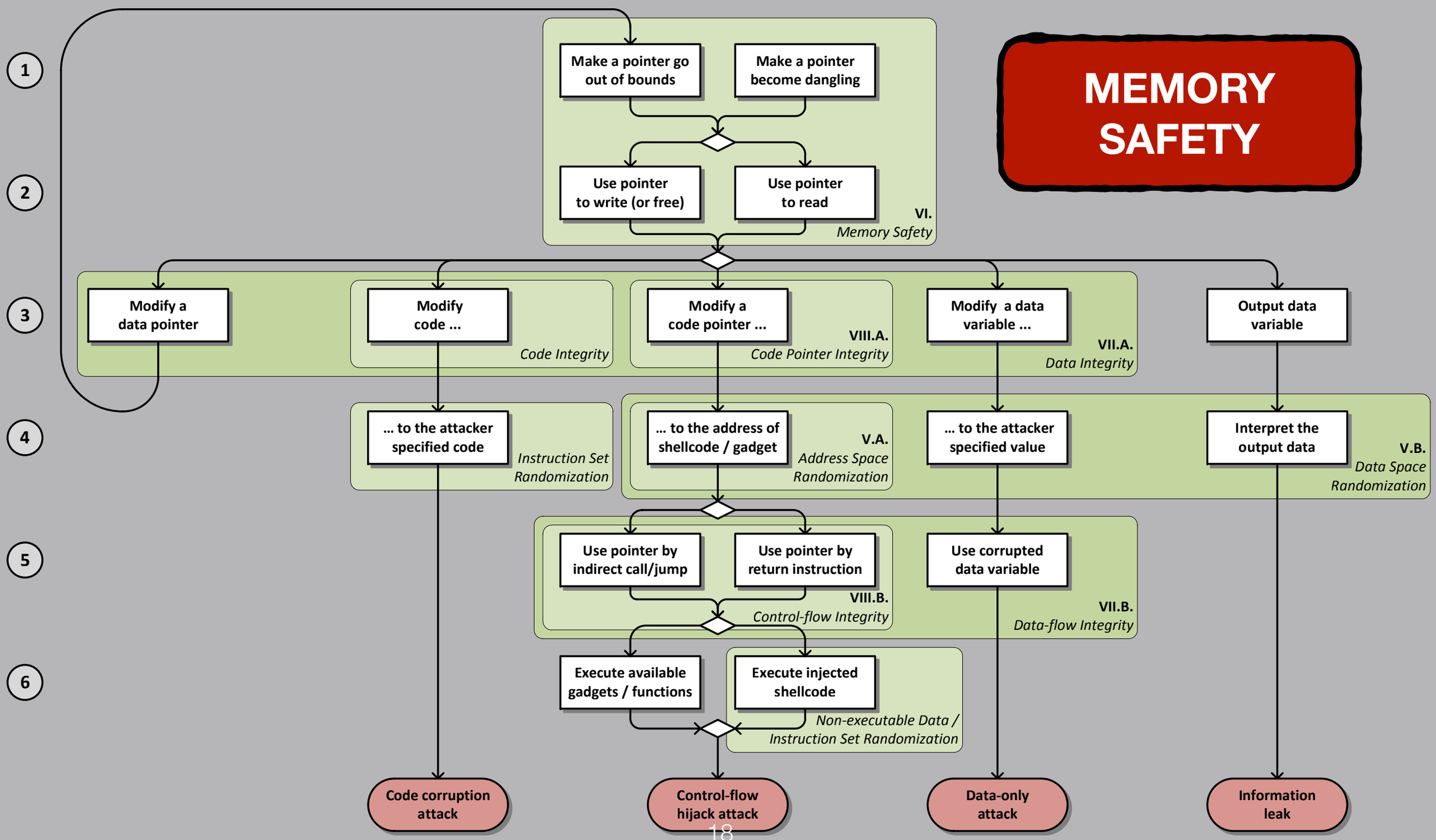


- https://youtu.be/T03idxny9jE

Types of memory errors

- Access errors
  - Buffer overflow
  - Race condition
    - Use after free
    - Segmentation fault
- Uninitialized variables
- Memory leak
  - Double free

SoK: Eternal War in Memory
Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
Proceedings of the 2013 IEEE Symposium on Security and Privacy



Approaches to Mitigate Memory Corruption Errors

- Program analysis like symbolic execution: **KLEE**
- Memory-checking virtual machine: **Valgrind**
- Compiler instrumentation: **AddressSanitizer**
- Fuzzing: **AFL, libFuzzer**
- Formal verification: **Seahorn, Smack, Trust-in-Soft**

Approaches to Mitigate Memory Corruption Errors

- Program analysis like symbolic execution: KLEE
- Memory checking virtual machine: Valgrind
- Compiler instrumentation: AddressSanitizer
- Fuzzing: AFL, libFuzzer
- Formal verification: Seahorn, Smack, Trust-in-Soft

Approaches to Mitigate Memory Corruption Errors

- ~~Program analysis like symbolic execution: KLEE~~
- ~~Memory-checking virtual machine: Valgrind~~
- ~~Compiler instrumentation: AddressSanitizer~~
- ~~Fuzzing: AFL, libFuzzer~~
- ~~Formal verification: Seahorn, Smack, Trust-in-Soft~~
- **Programming languages: Rust**

System Programming

- Memory management
- Error handling
- Static Typing
- Compiling
- ...

Rust

- Rust is a **systems programming language** that runs blazingly **fast**, prevents segfaults, and guarantees **thread safety**.

What causes memory issues?

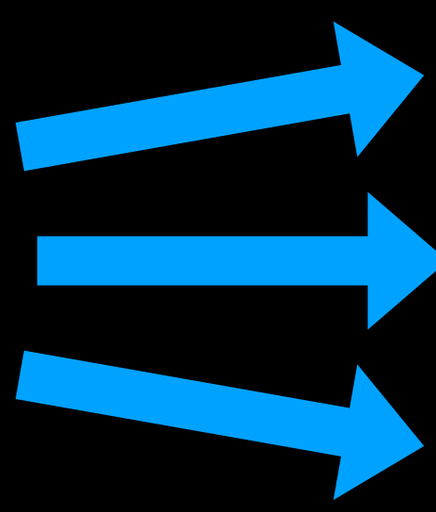
Aliasing + ***~~Mutation~~***

~~Aliasing~~ + ***Mutation***

Aliasing + ***Mutation***

How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetime



- No need for a runtime (C/C++)
- Memory safety (GC)
- Data-race freedom

Ownership and Borrowing

- In Rust, every value has a **single, statically-known, owning path** in the code, at any time.

- Pointers to values have limited duration, known as a "**lifetime**", that is also **statically tracked**.

- All pointers to all values are known **statically**.

Ownership

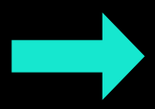
Alice


```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice

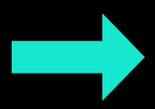
Bob



```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice **Bob**



```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

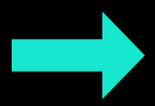
Alice


```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

Alice

```
error[E0382]: use of moved value: `alice`
 --> src/main.rs:7:27
  |
4 |         let bob = alice;
  |             --- value moved here
...
7 |     println!("alice: {}", alice[0]);
  |                           ^^^^^ value used here after move
  |
  = note: move occurs because `alice` has type
`std::vec::Vec<i32>`, which does not implement the `Copy` trait
```



```
fn main() {
    let alice = vec![1, 2, 3];;
    {
        let bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Ownership (T)

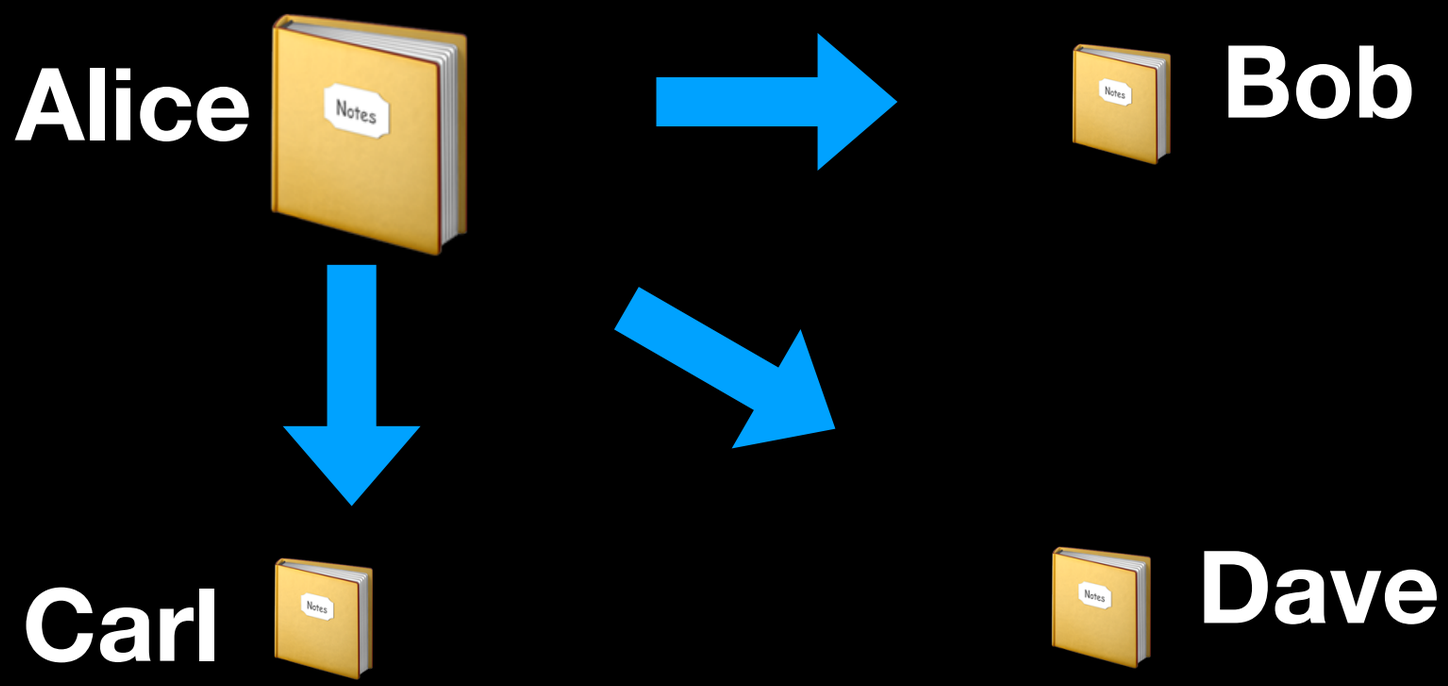
Alice

```
error[E0382]: use of moved value: `alice`
 --> src/main.rs:7:27
  |
4 |         let bob = alice;
  |             --- value moved here
...
7 |     println!("alice: {}", alice[0]);
  |                           ^^^^^ value used here after move
  |
  = note: move occurs because `alice` has type
`std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

```
fn main() {
    let mut alice = vec![1, 2, 3];;
    {
        let mut bob = alice;
        println!("bob: {}", bob[0]);
    }
    println!("alice: {}", alice[0]);
}
```

Shared Borrow (&T)

Aliasing + ~~Mutation~~



Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice **Bob**


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

Alice


```
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

Mutable Borrow (&mut T)

~~Aliasing~~ + **Mutation**

Alice

The lifetime of a borrowed reference should end before the lifetime of the owner object does.

Rust's Ownership & Borrowing

~~Aliasing~~ + ***~~Mutation~~***

- Compiler enforced:
  - Every resource has a unique **owner**
  - Others can **borrow** the resource from its owner (e.g., create an **alias**) with restrictions
  - Owner **cannot** free or mutate its resource while it is borrowed

Ownership & Borrowing

Owership	`T`	"owned"
Exclusive access	`&mut T`	"mutable"
Shared access	`&T`	"read-only"

Use-After Free in C/Rust

C/C++

```
void func() {
    int *used_after_free = malloc(sizeof(int));

    free(used_after_free);

    printf("%d", *used_after_free);
}
```

Rust

```
fn main() {
    let name = String::from("Hello World");
    let mut name_ref = &name;
    {
        let new_name = String::from("Goodbye");
        name_ref = &new_name;
    }
    println!("name is {}", &name_ref);
}
```

Use-After Free in Rust

```
error[E0597]: `new_name` does not live long enough
 --> main.rs:7:5
  |
6 |         name_ref = &new_name;
  |                    --------- borrow occurs here
7 |     }
  |     ^ `new_name` dropped here while still borrowed
8 |     println!("name is {}", &name_ref);
9 | }
  | - borrowed value needs to live until here

error: aborting due to previous error
```

Use Rust for TEE Dev

- Static/strong type system
- Minimal language runtime
- Memory-safety
- Community

Summary

- What is TEE?
- Why having an OS in TEE?
- Does language matter?