RedLeaf
Isolation and Communication in a Safe Operating System

OSDI’ 20

University of California, Irvine

VMware Research

TOC

1. Overview

Overview

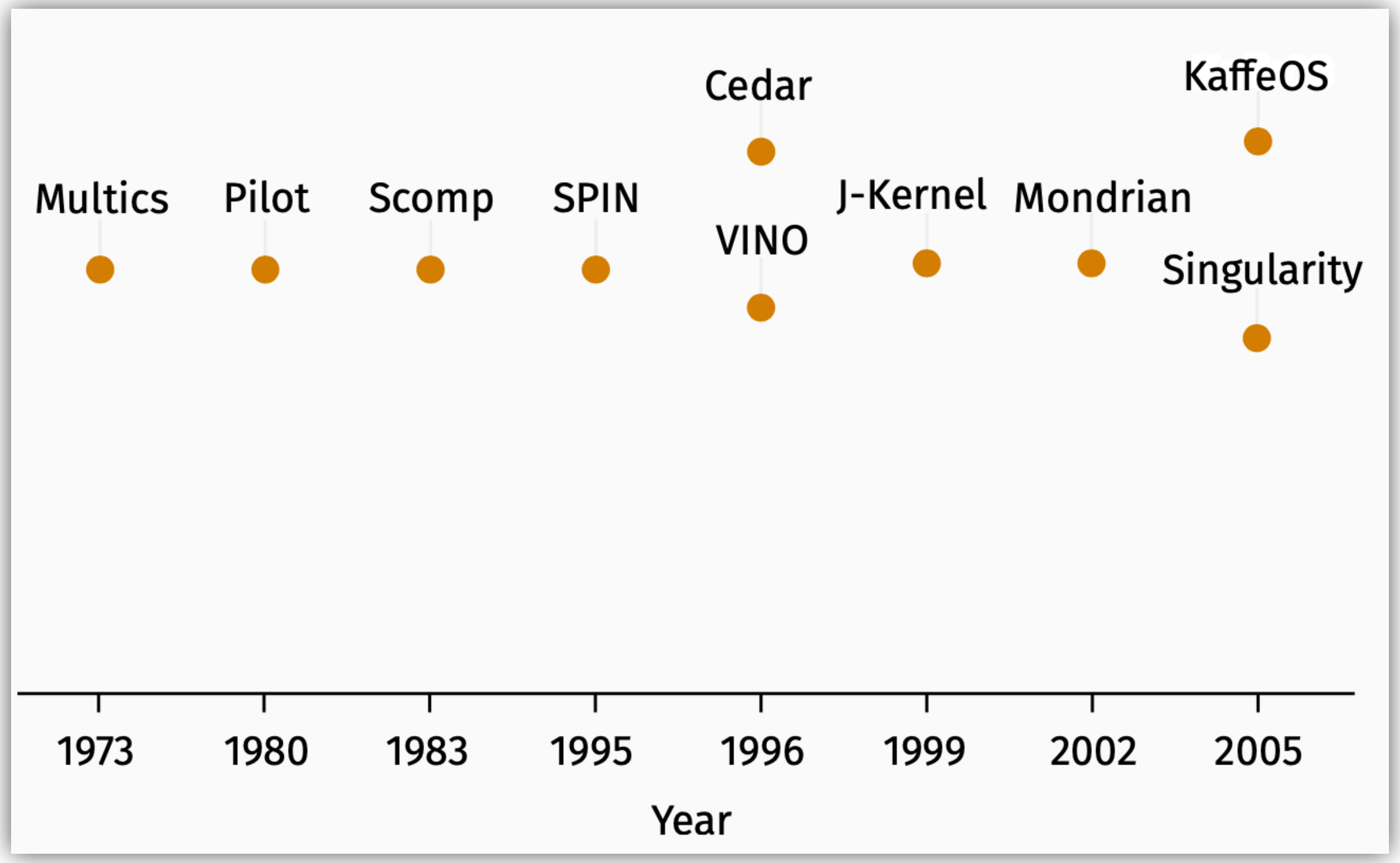
- RedLeaf OS with novel **isolation** mechanism
  - NO costly hardware-based isolation
  - Relies on **type and memory safety of Rust**
  - **IDL** that supports cross-domain call proxying
  - (Engineering) POSIX-subset, NVMe, 10Gbps Intel ixgbe network

2. Background

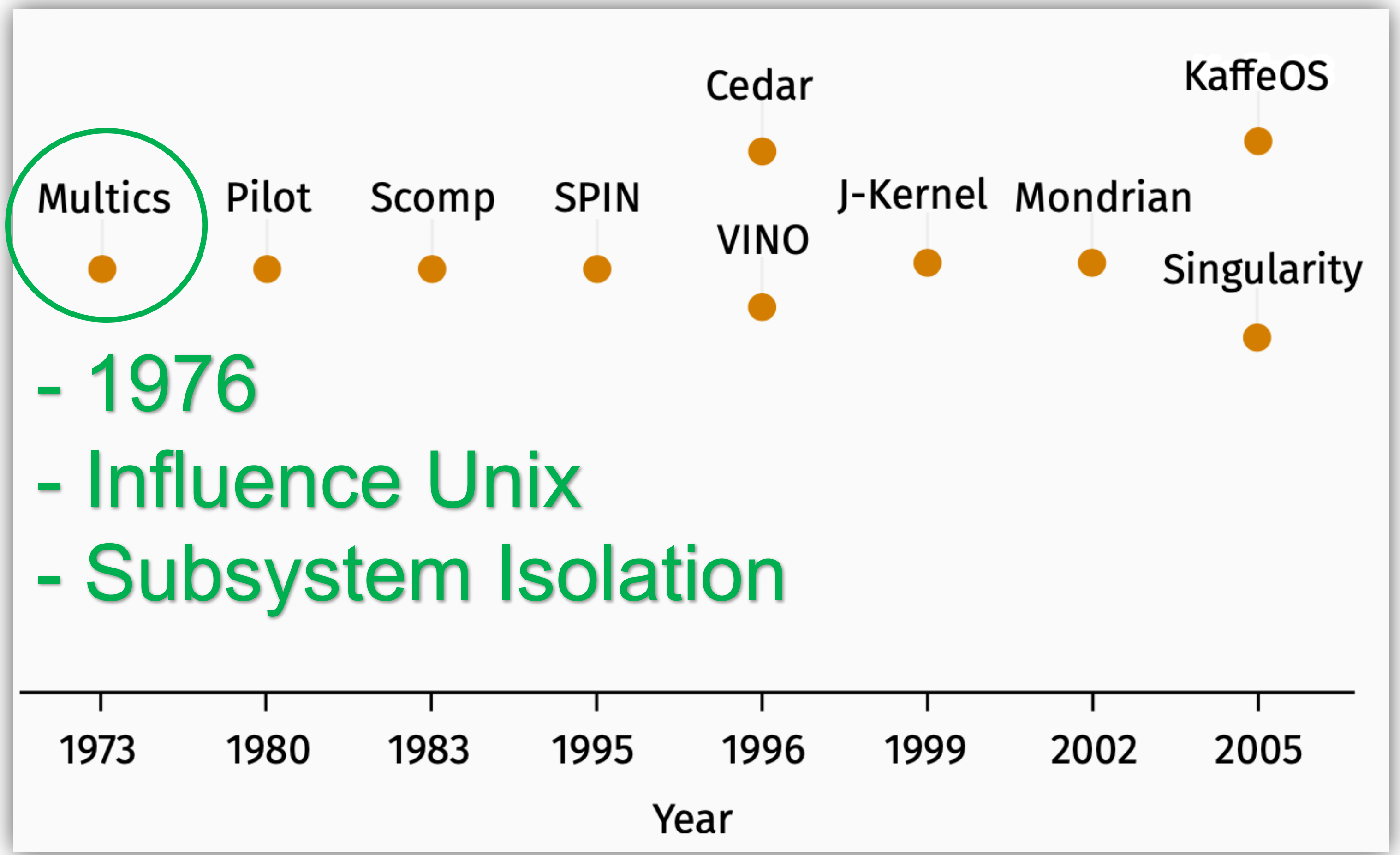
2.1 如何评估一个安全的系统

- Domains: a unit of resources and info (本文中的主要分析对象)
- Domains can be **cleanly** terminated
- The **faults and crashes** in one domain do not affect other domains
- Shared objects cause many problems !

2.2 History of Isolation

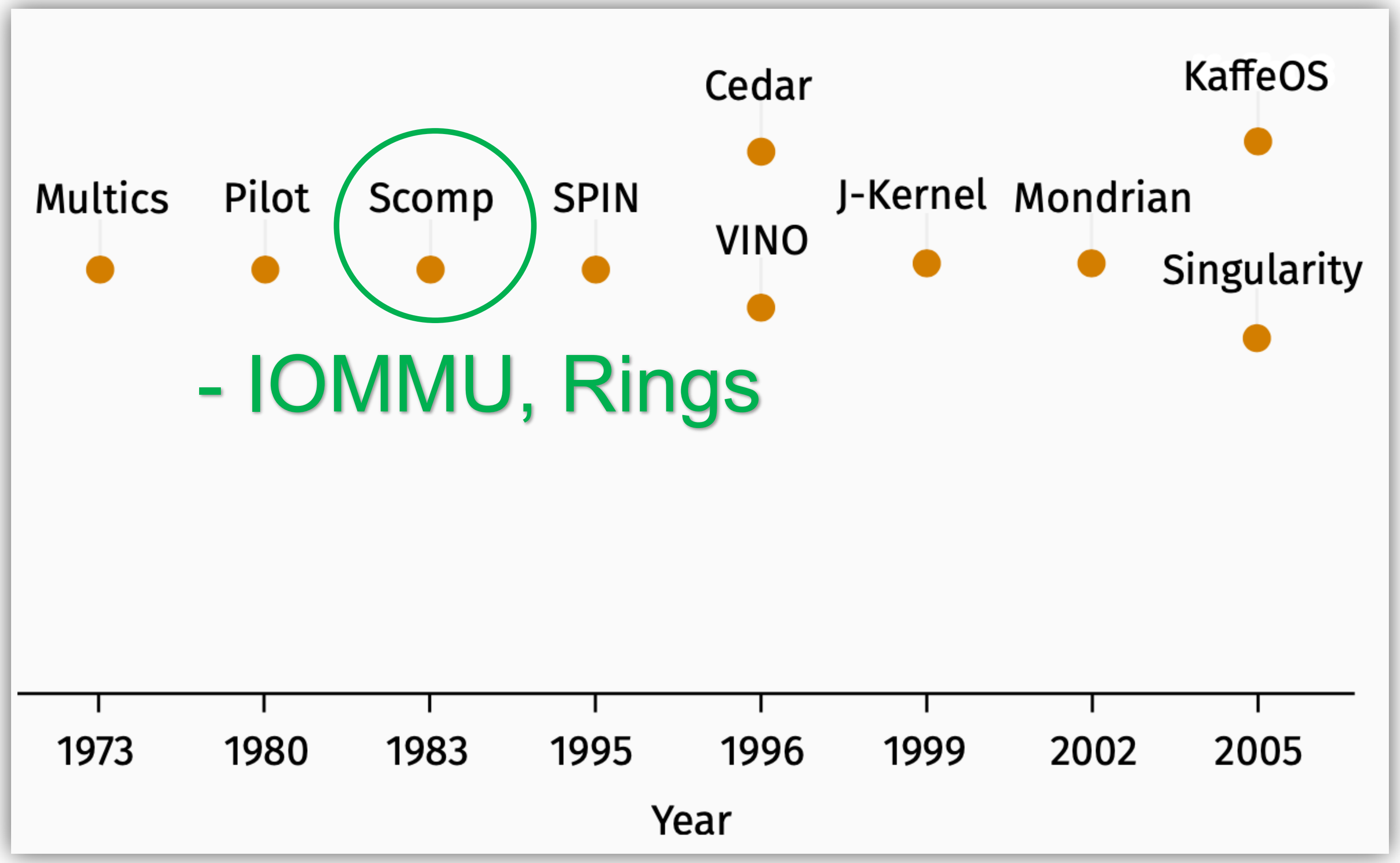


2.2 History of Isolation

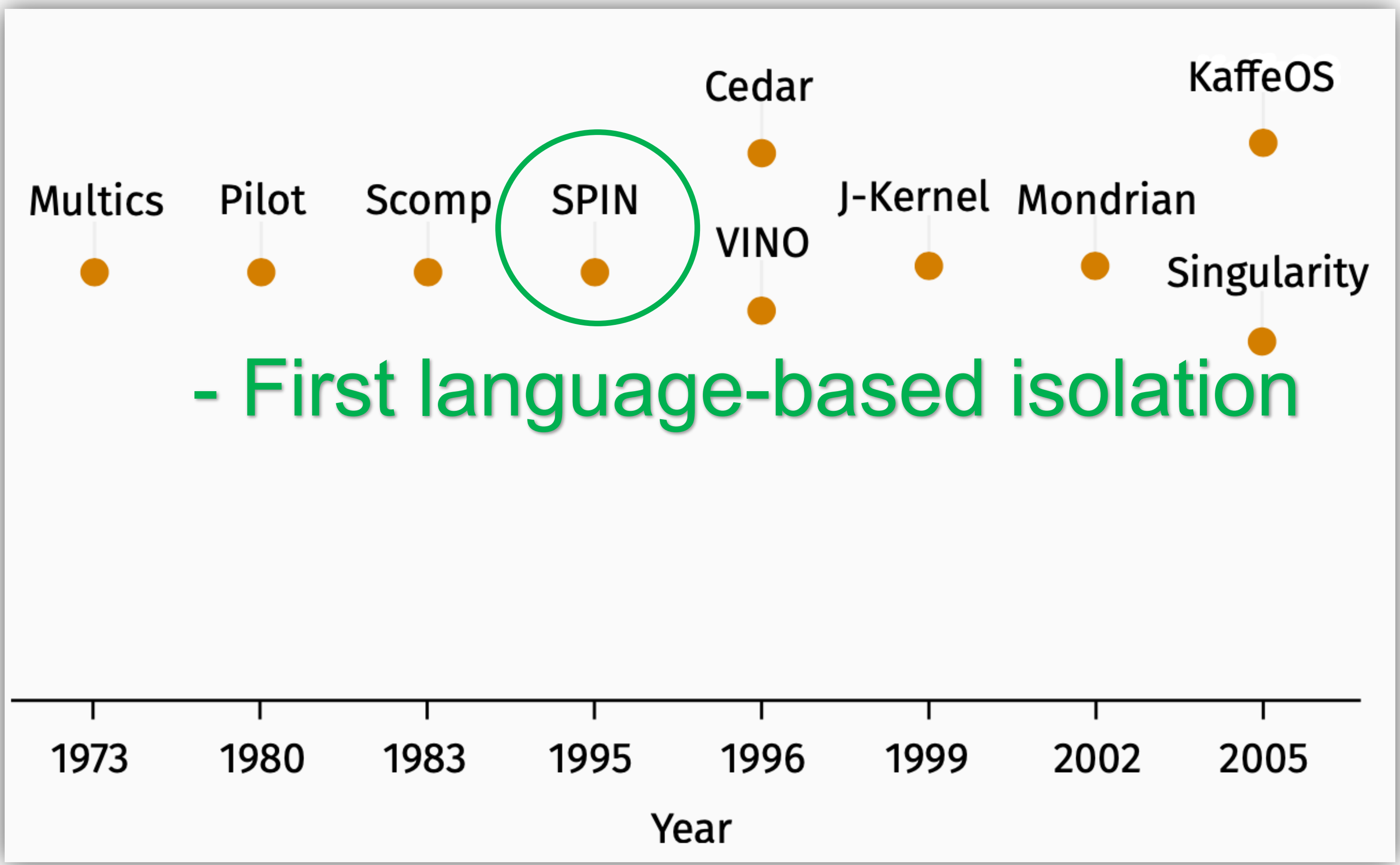


- 1976
- Influence Unix
- Subsystem Isolation

2.2 History of Isolation

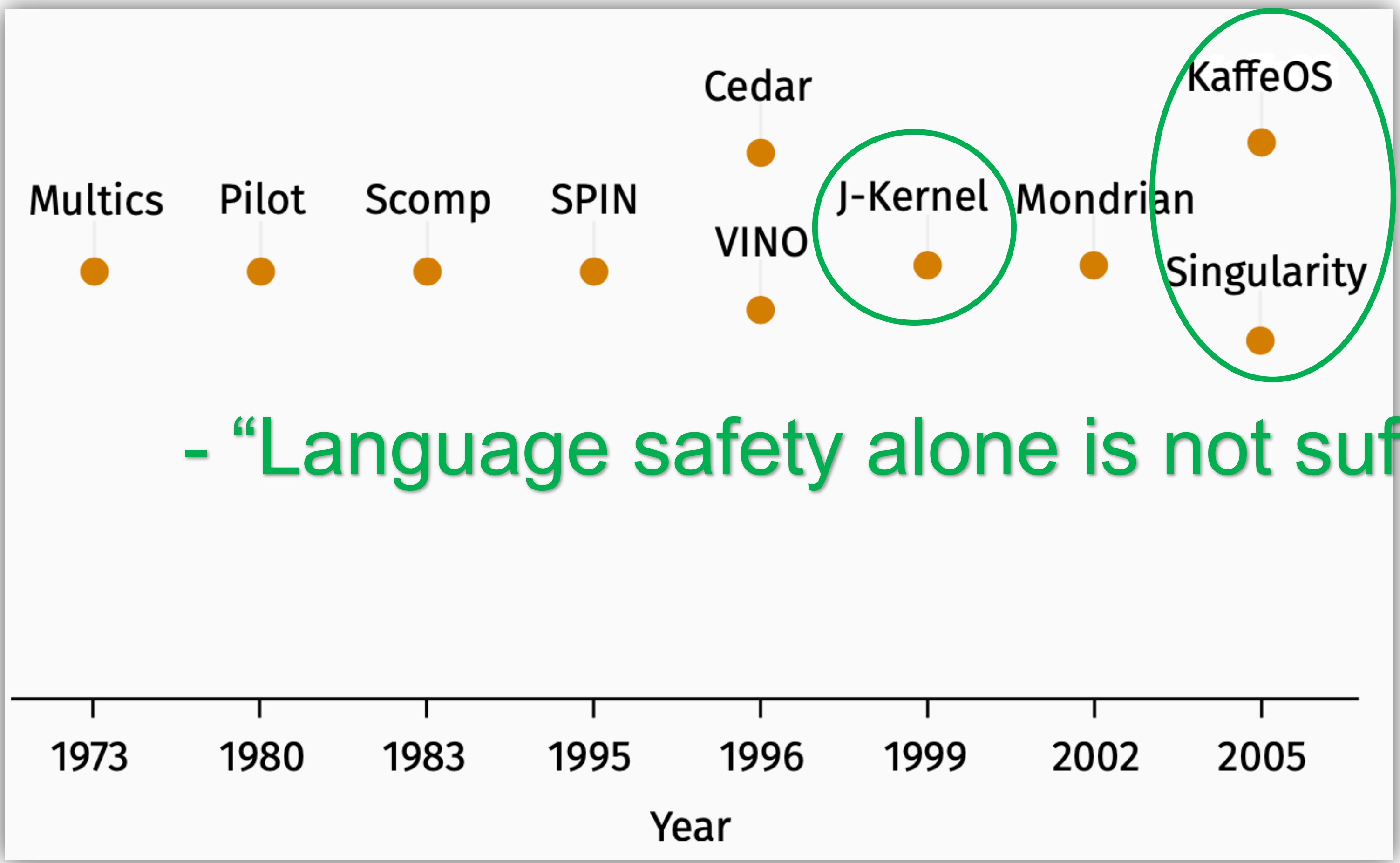


2.2 History of Isolation



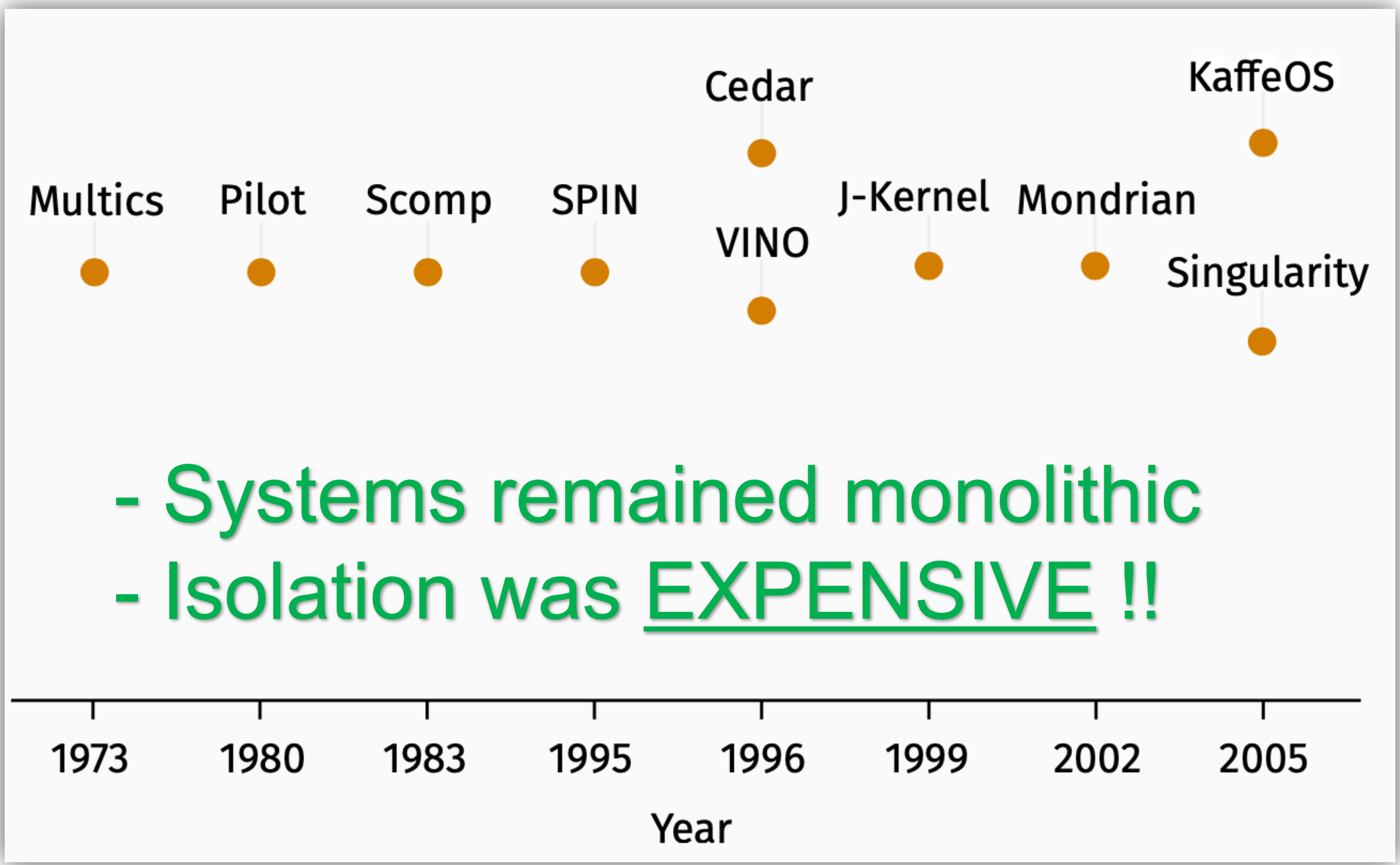
- First language-based isolation

2.2 History of Isolation



- “Language safety alone is not sufficient”

2.2 History of Isolation

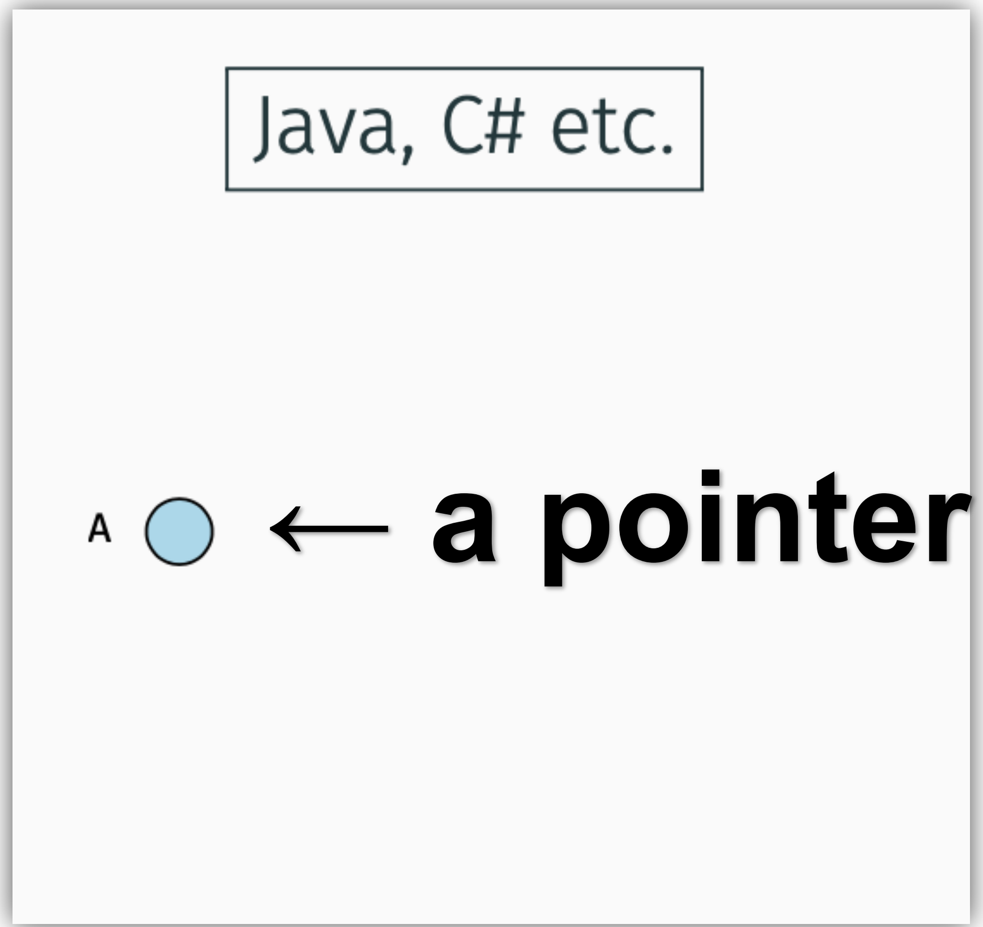


- Systems remained monolithic
- Isolation was EXPENSIVE !!

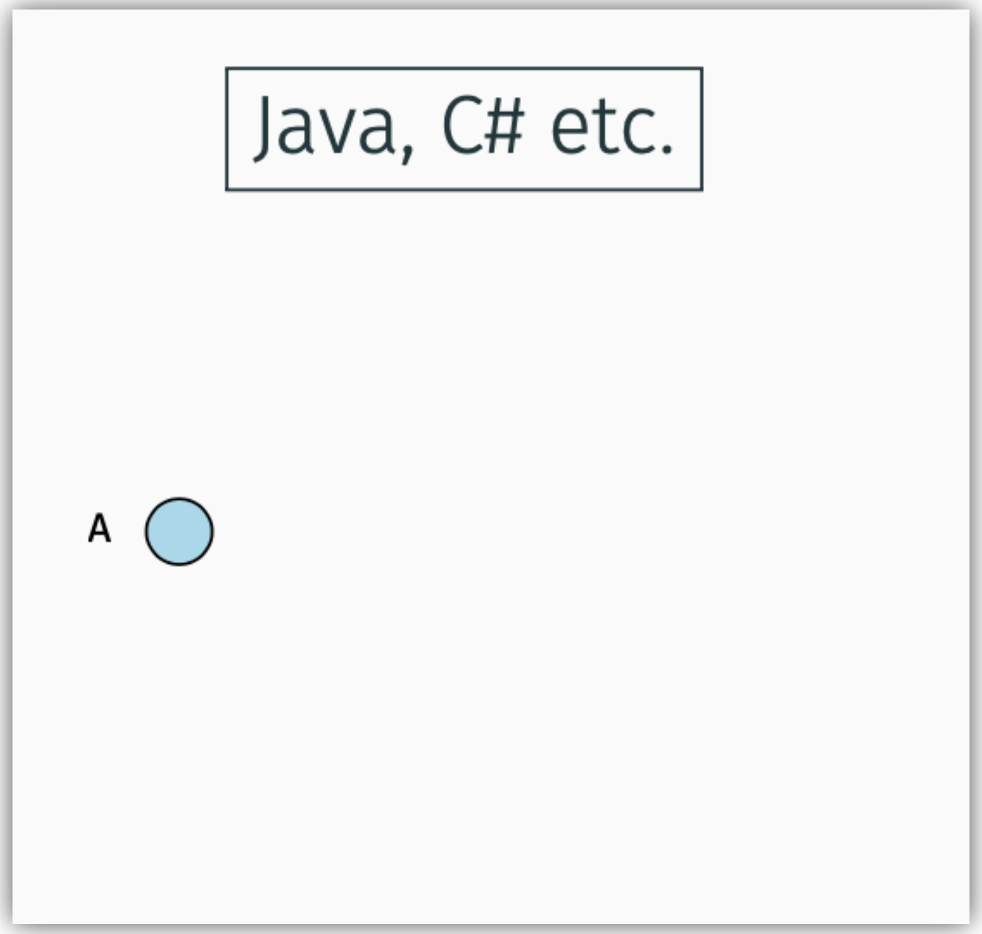
2.3 Isolation Mechanisms & Drawbacks

- Hardware Isolation & Latency
  - Segmentation (46 cycles)
  - Page table isolation (797 cycles)
  - VMFUNC (396 cycles)
  - Memory protection keys (20-26 cycles)
- Language based isolation
  - Compare drivers written (DPDK-style) in a safe high-level language (C, Rust, Go, C#, etc.)
  - Managed runtime and Garbage collection (20-50% overhead on a device-driver workload)

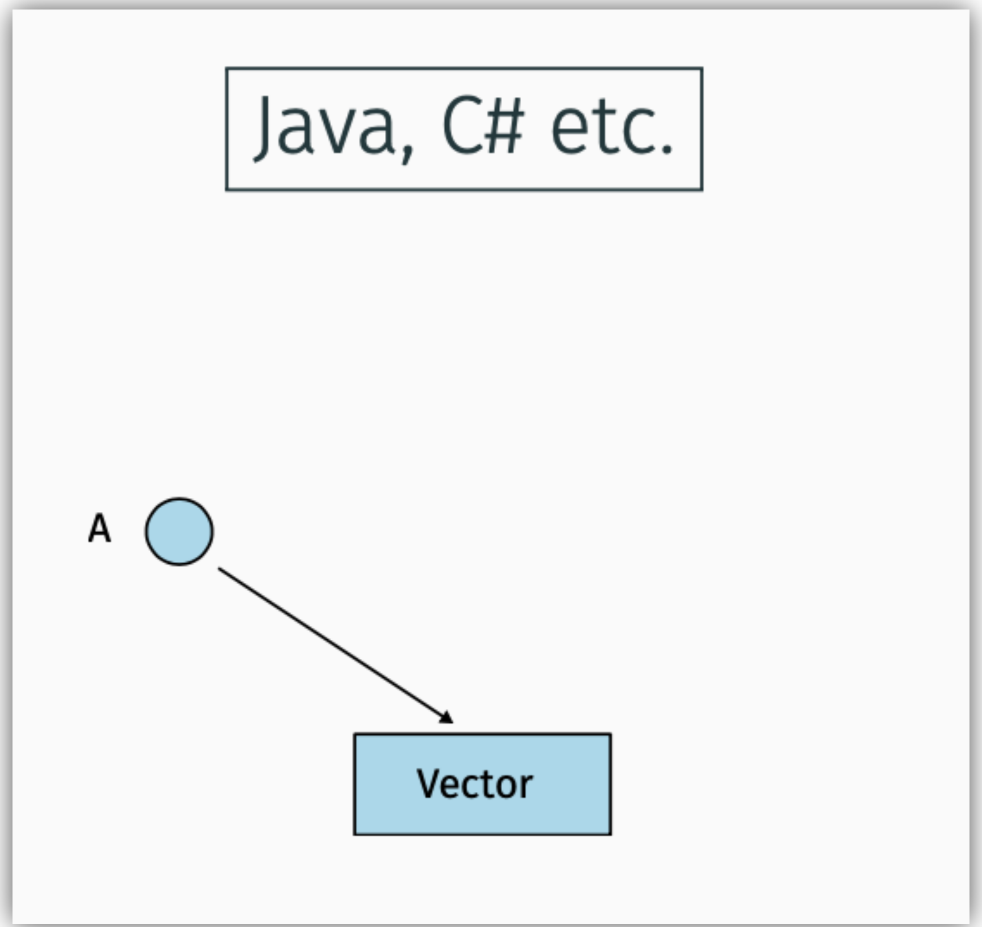
2.4 Traditional Safe Languages



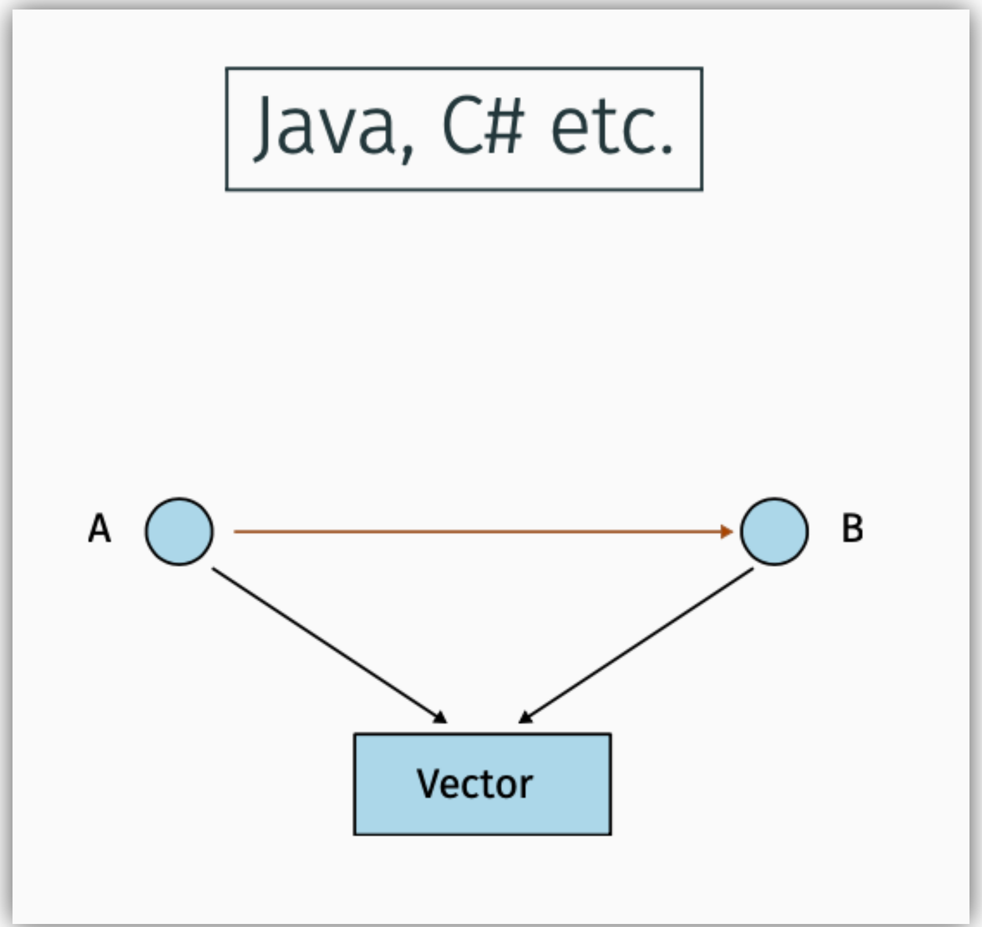
2.4 Traditional Safe Languages



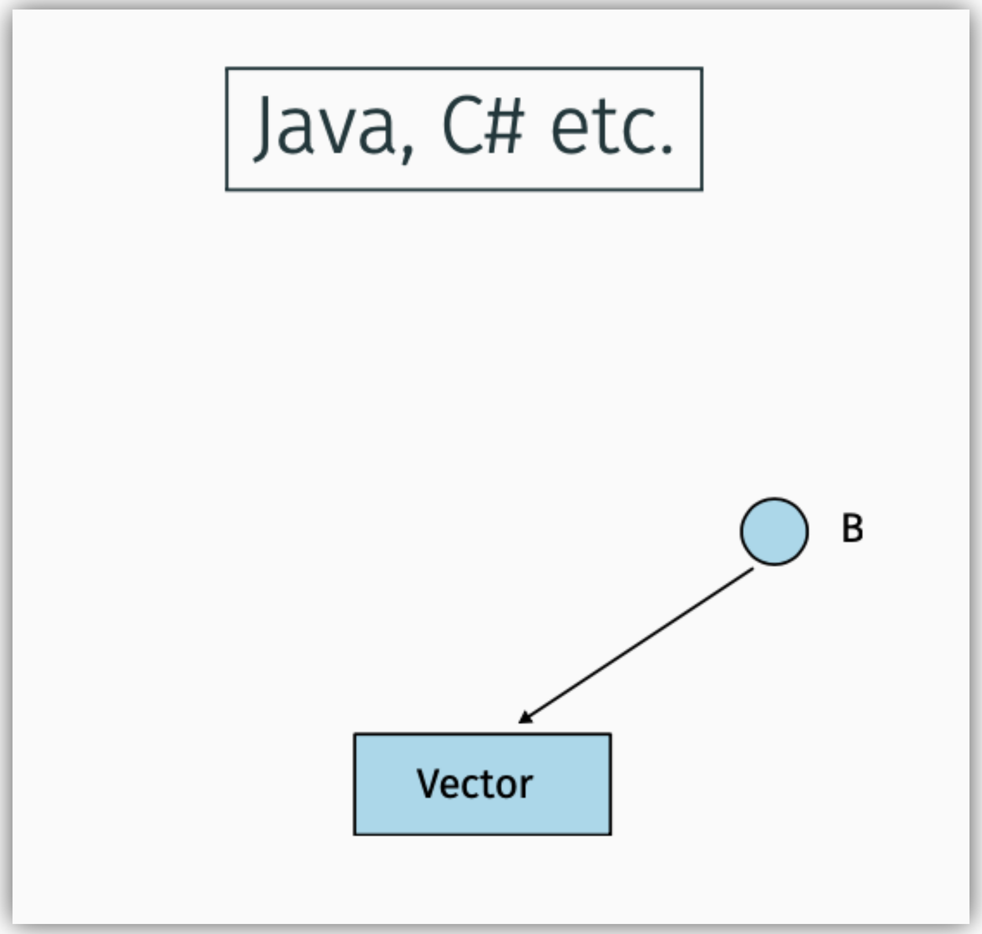
2.4 Traditional Safe Languages



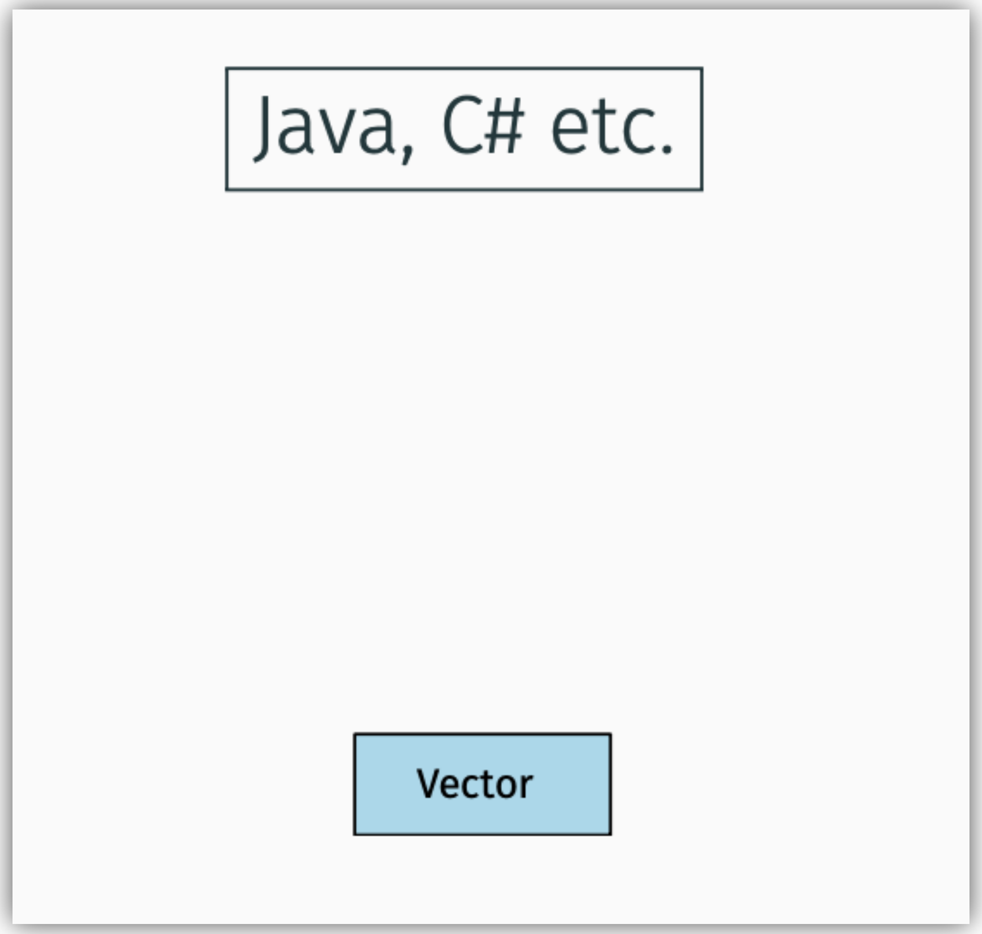
2.4 Traditional Safe Languages



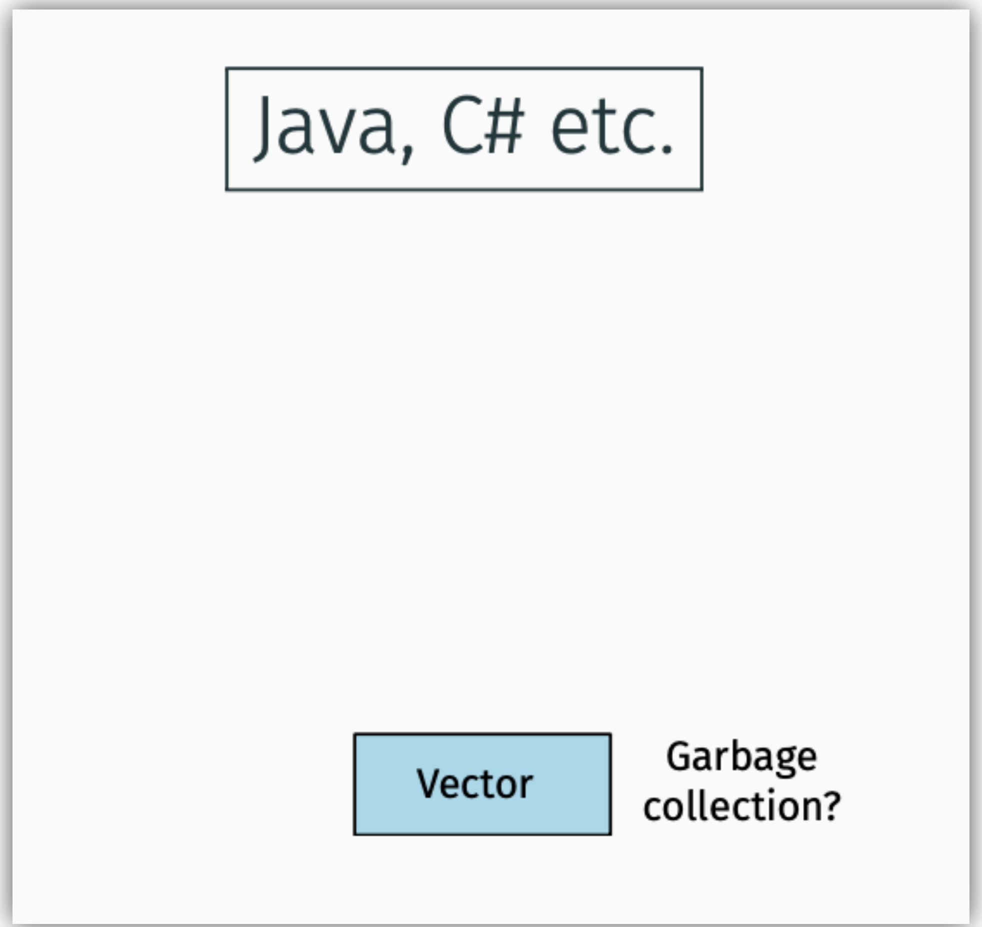
2.4 Traditional Safe Languages



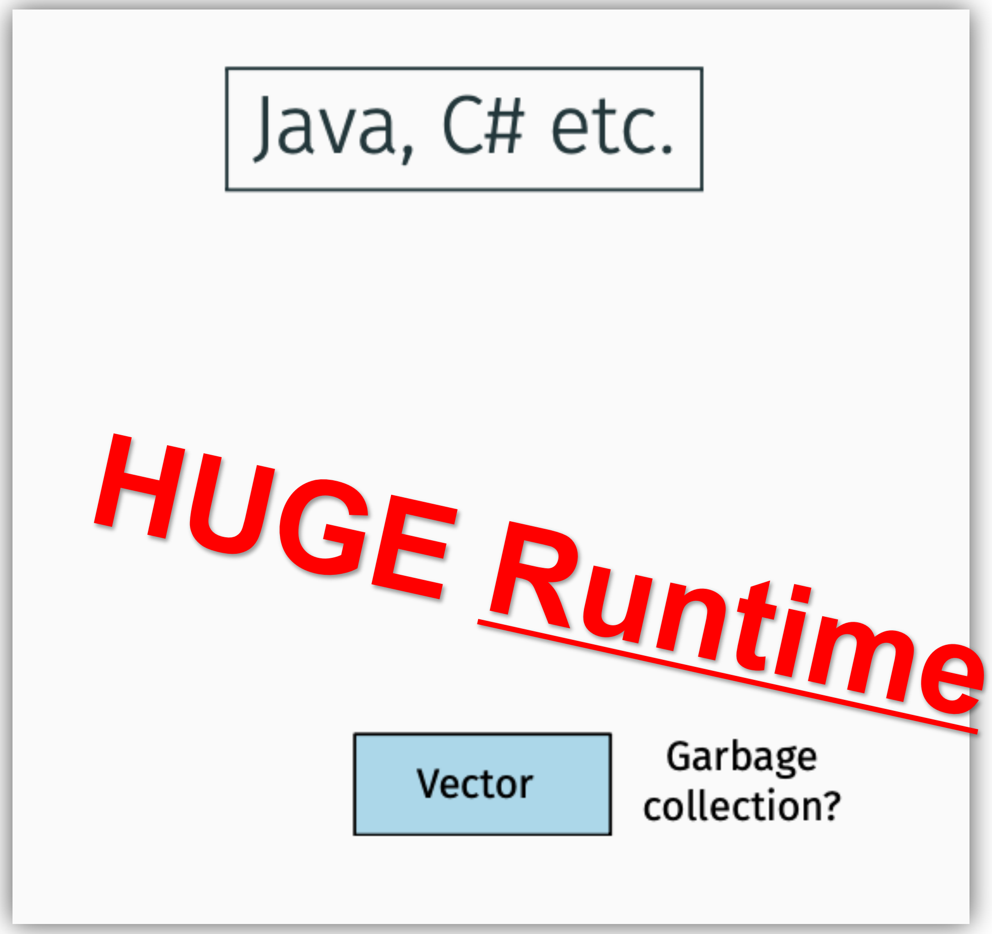
2.4 Traditional Safe Languages



2.4 Traditional Safe Languages

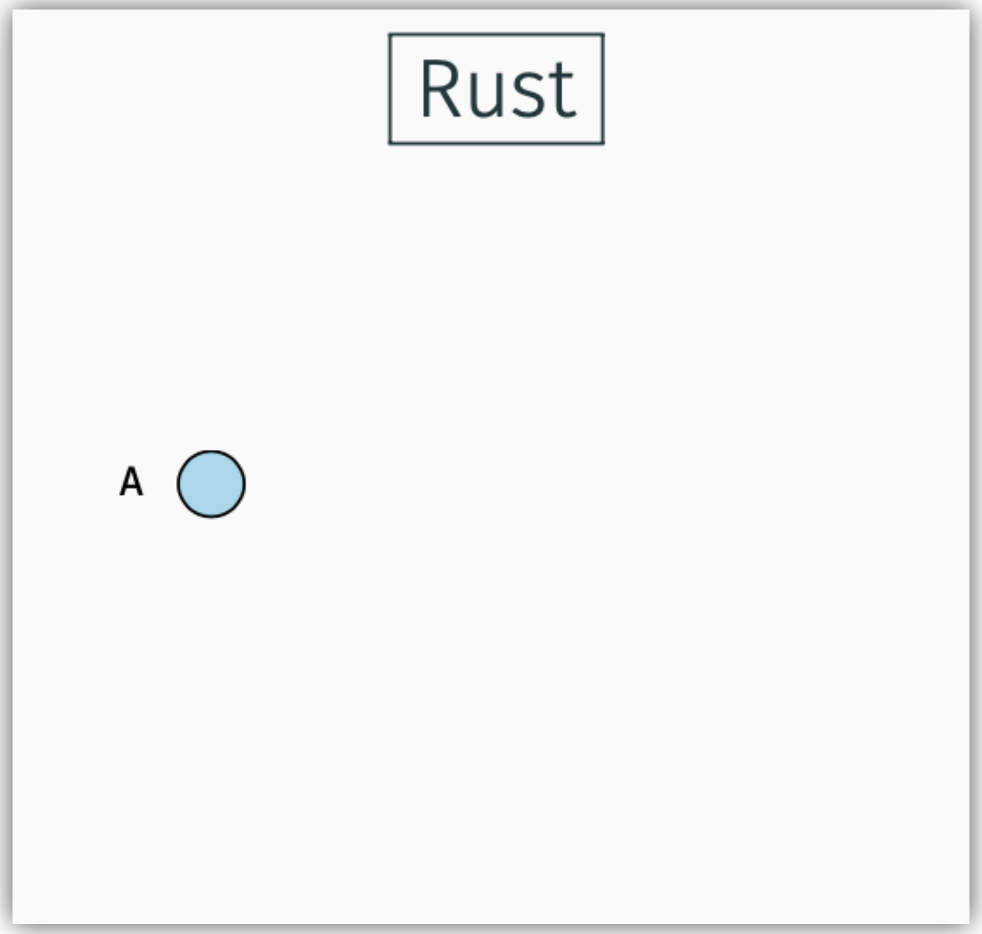


2.4 Traditional Safe Languages

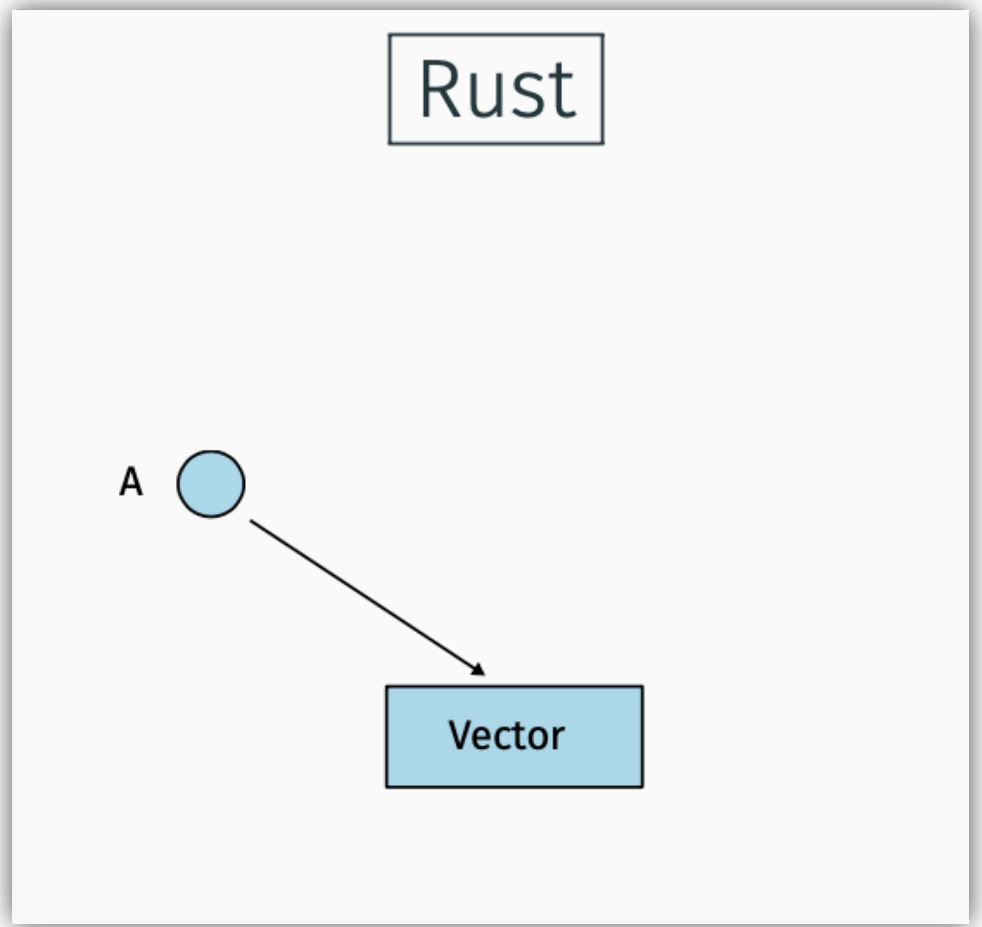


HUGE Runtime Overhead

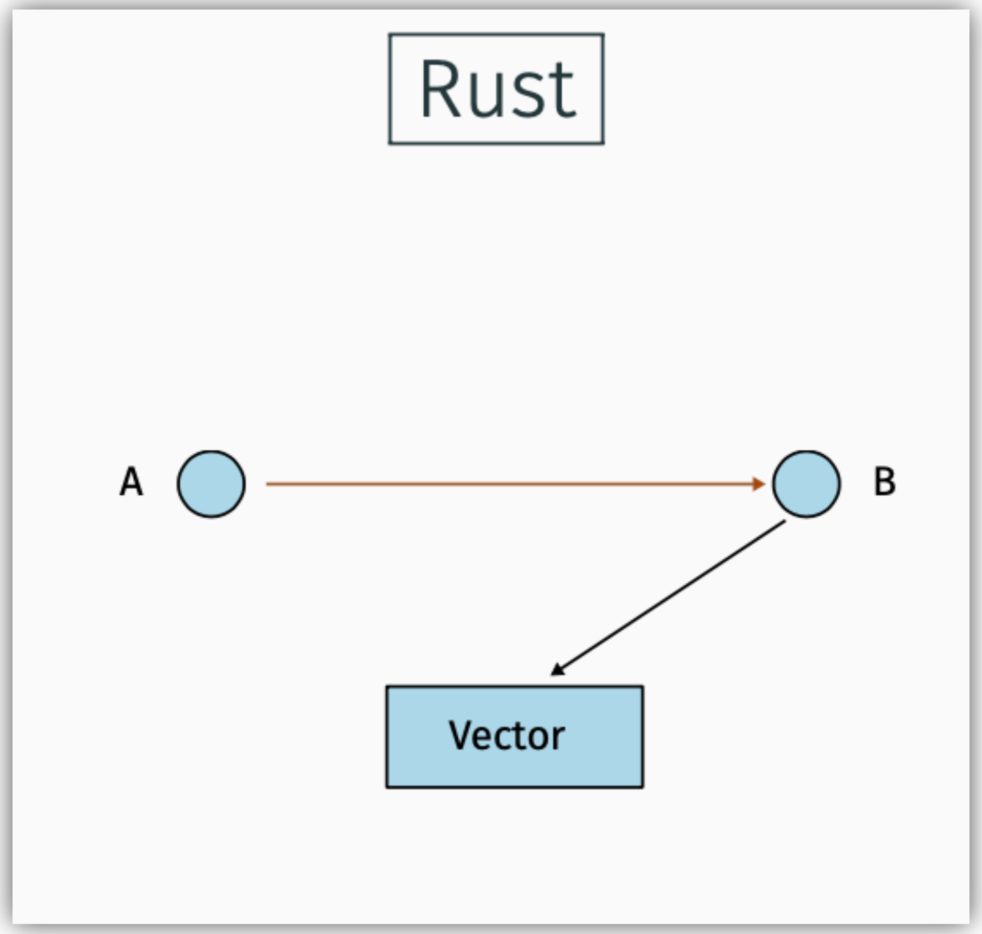
2.5 Rust



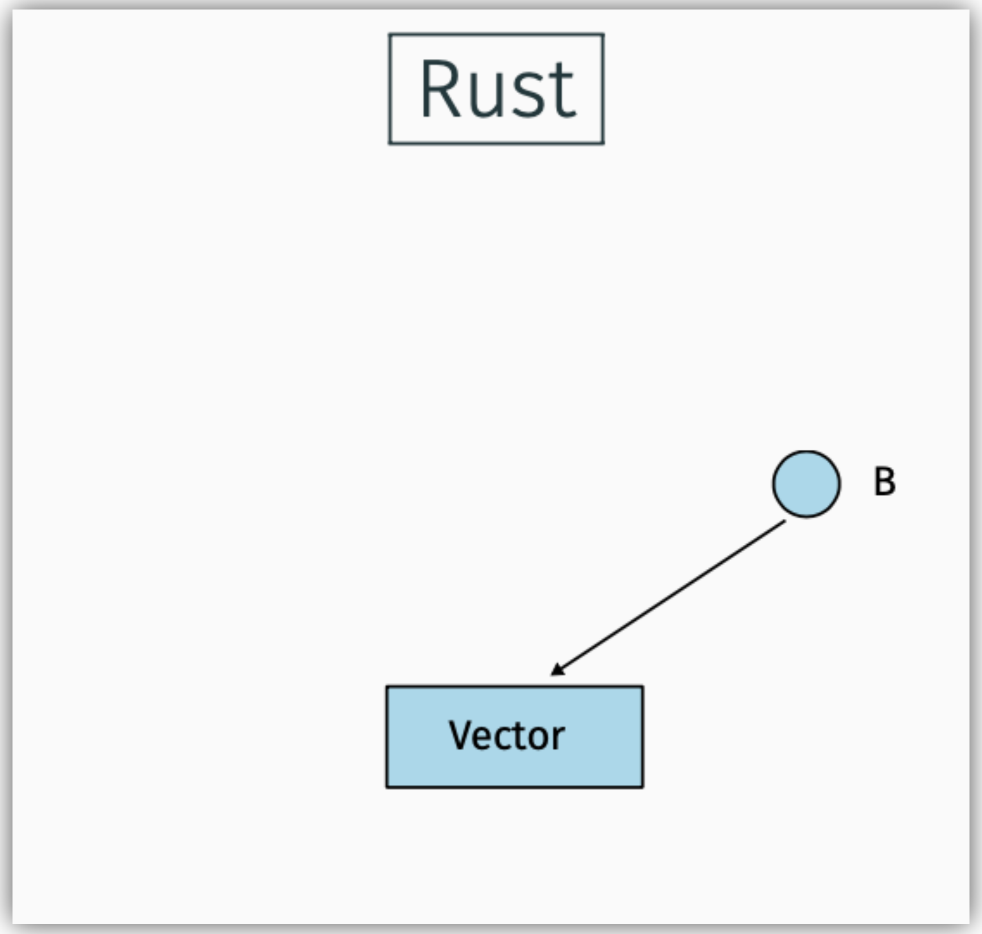
2.5 Rust



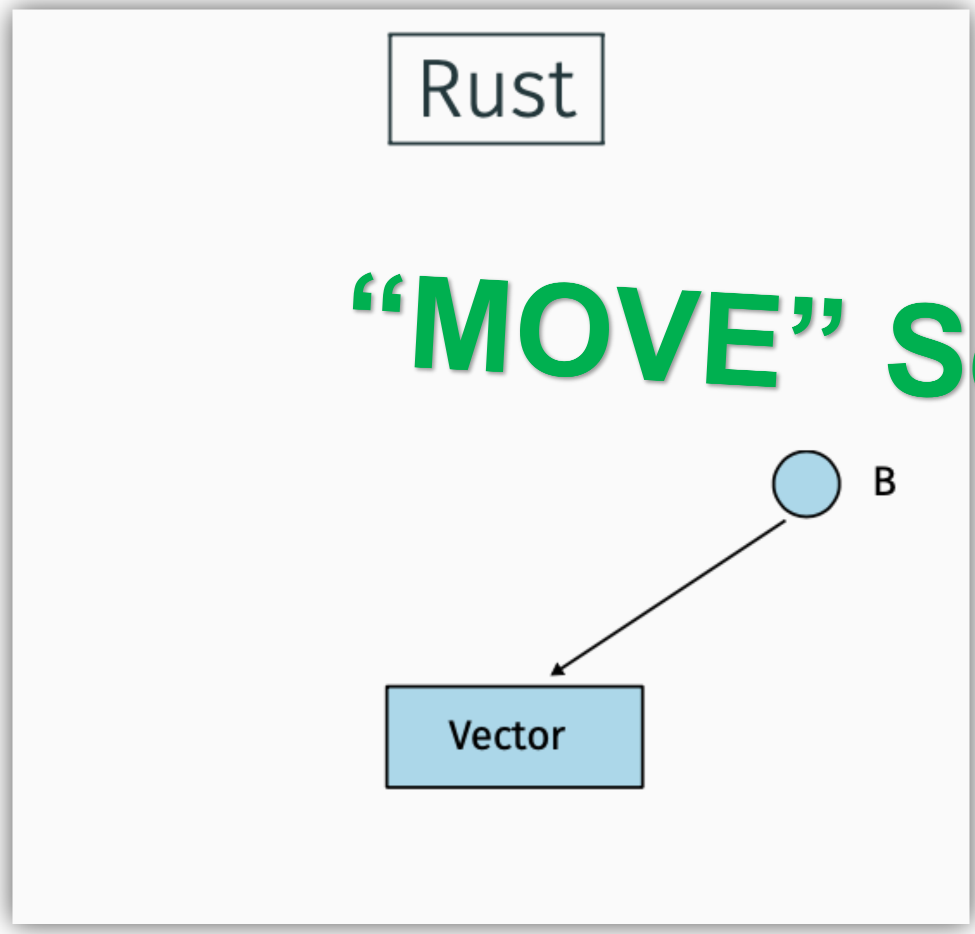
2.5 Rust



2.5 Rust

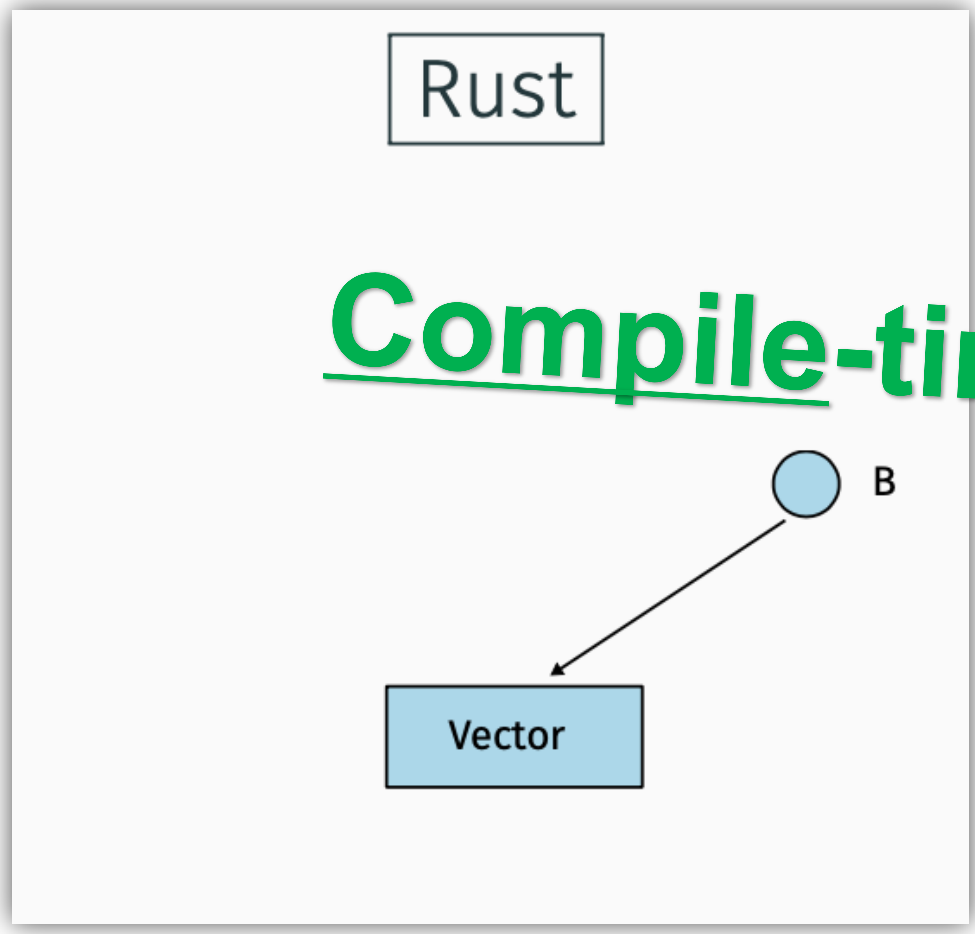


2.5 Rust



“MOVE” Semantic

2.5 Rust



Compile-time GC √

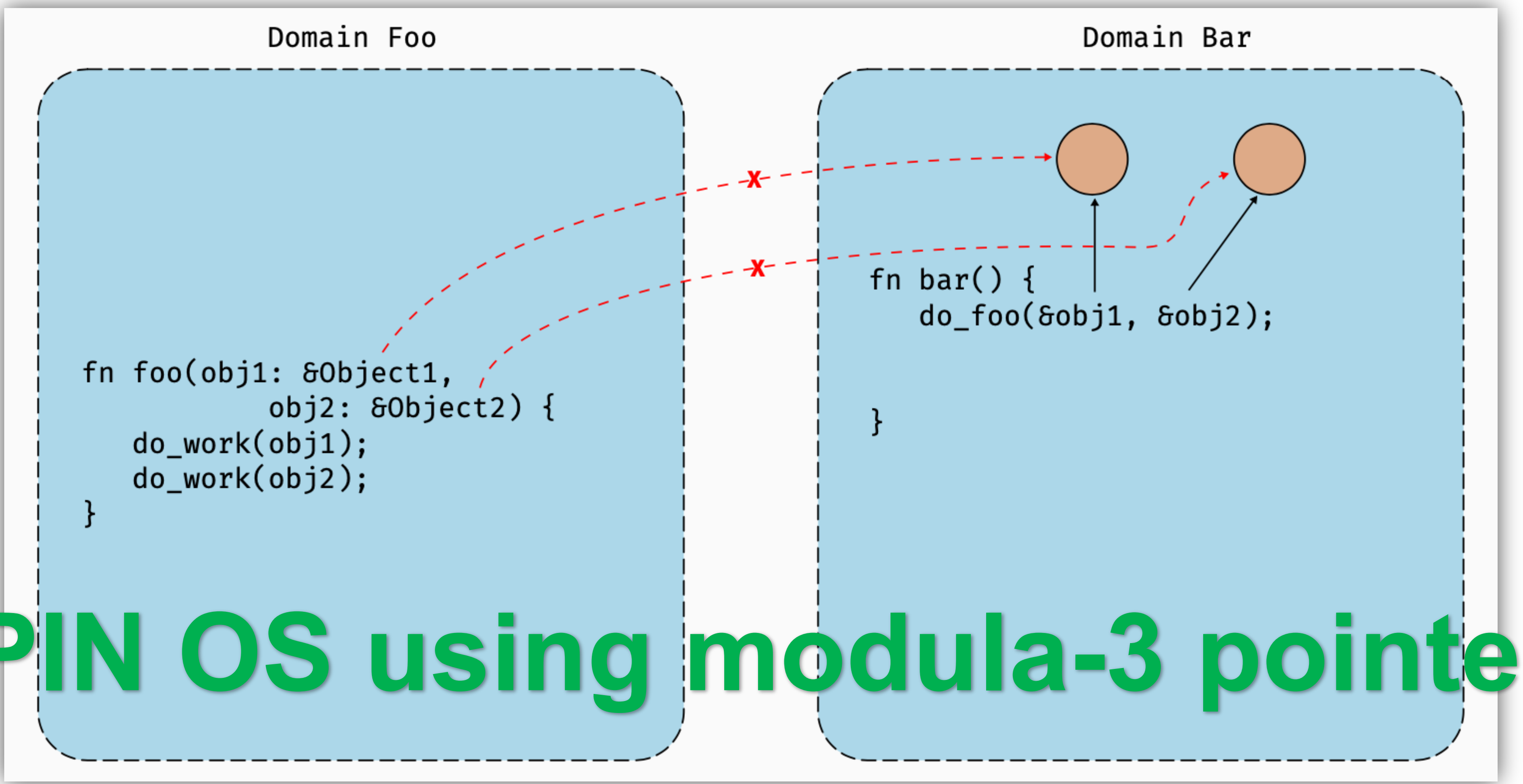
2.5 Rust

- Mostly use Rust as a drop-in replacement for C
- Numerous possibilities
  - Fault Isolation
  - Transparent device-driver recovery
  - Safe Kernel extensions
  - Fine-grained capability-based access control etc.

2.6 Fault Isolation in Language-based Systems

- Fault Isolation as is mentioned before:
  - Domains can be **cleanly** terminated
  - The **faults and crashes** in one domain do not affect other domains

2.6 Fault Isolation in Language-based Systems

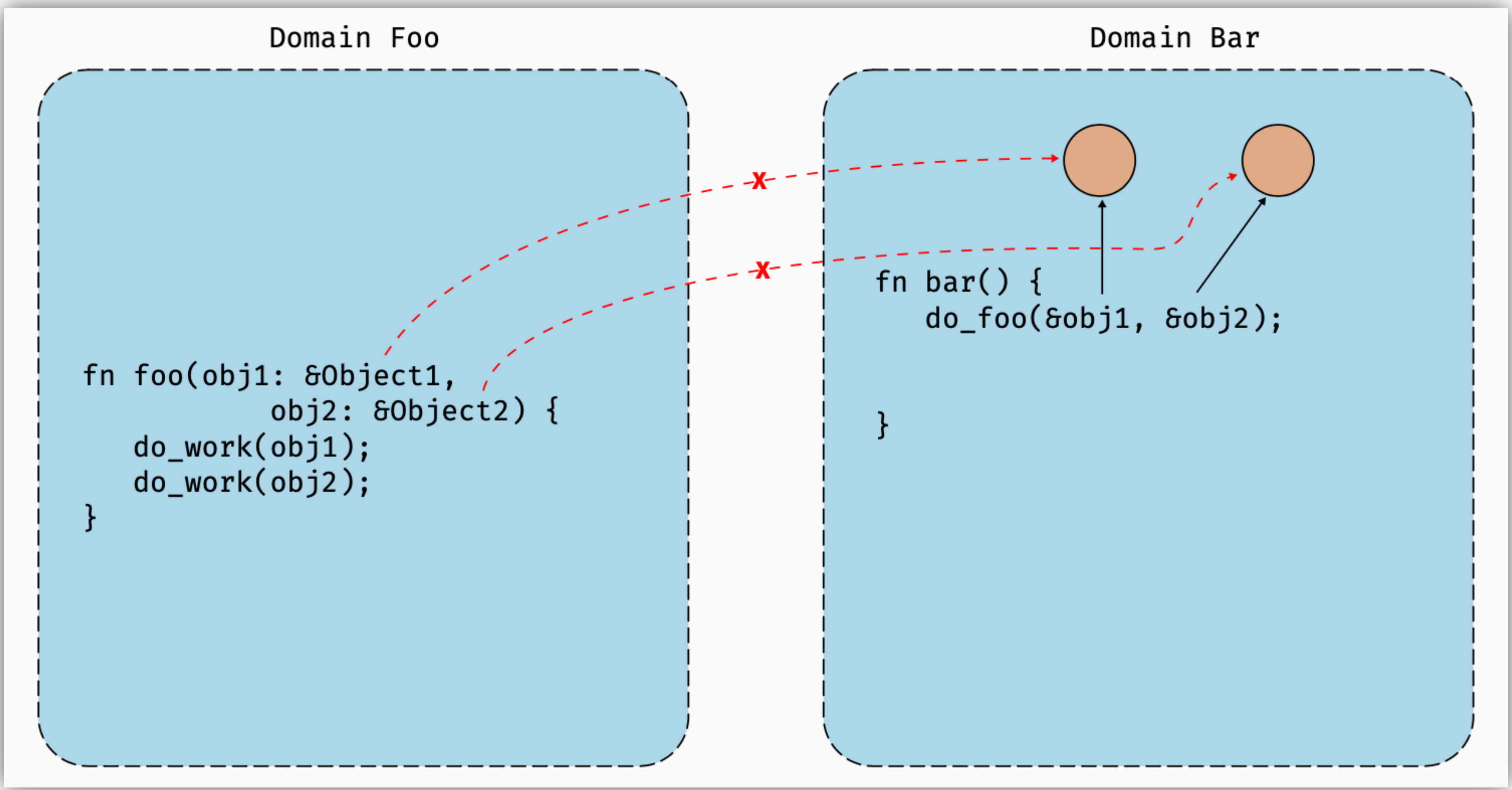


```
fn foo(obj1: &Object1,
           obj2: &Object2) {
   do_work(obj1);
   do_work(obj2);
}
```

```
fn bar() {
   do_foo(&obj1, &obj2);

}
```

SPIN OS using modula-3 pointers

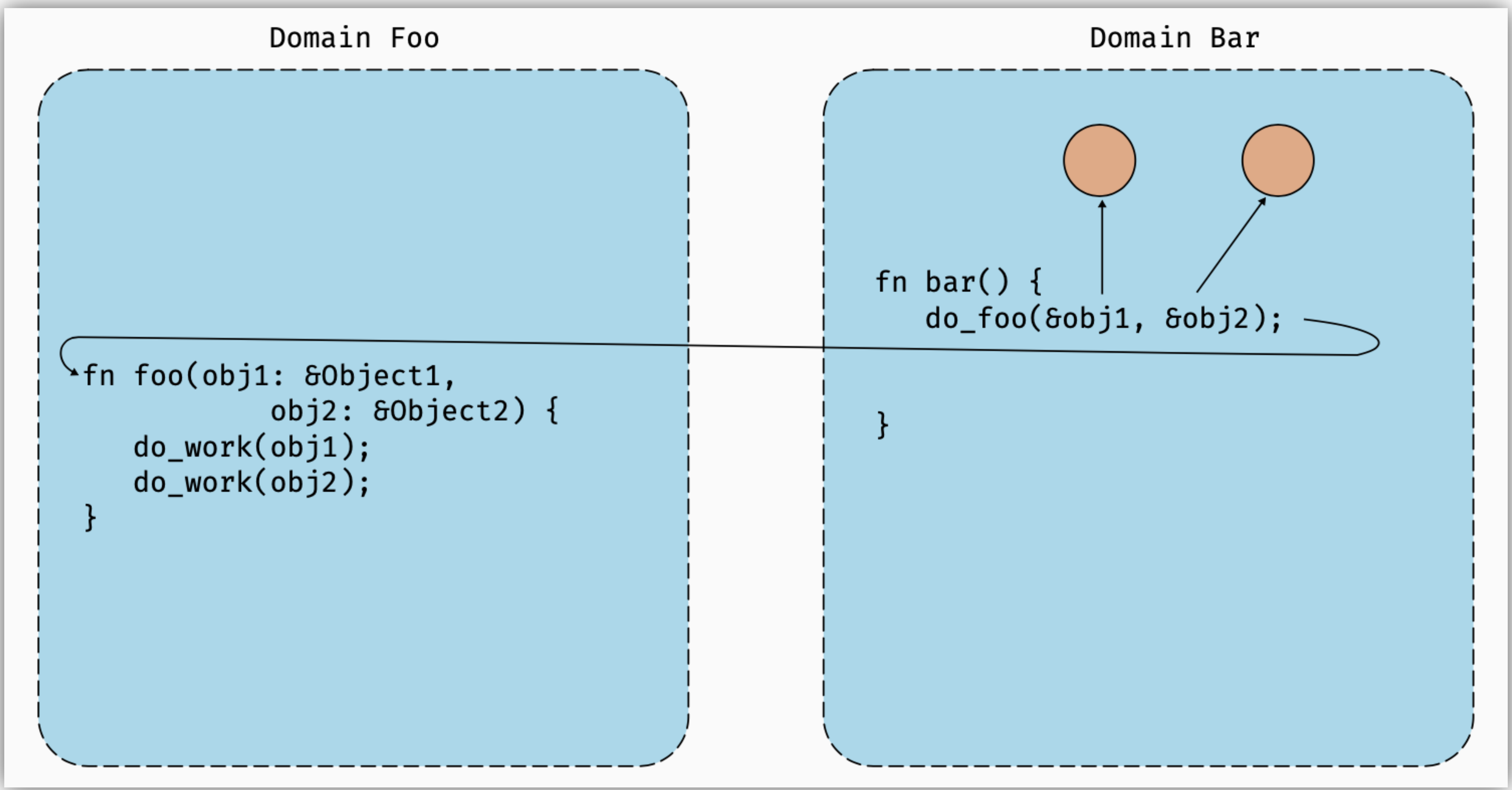
2.6 Fault Isolation in Language-based Systems



```
fn foo(obj1: &Object1,
          obj2: &Object2) {
   do_work(obj1);
   do_work(obj2);
}
```

```
fn bar() {
   do_foo(&obj1, &obj2);

}
```

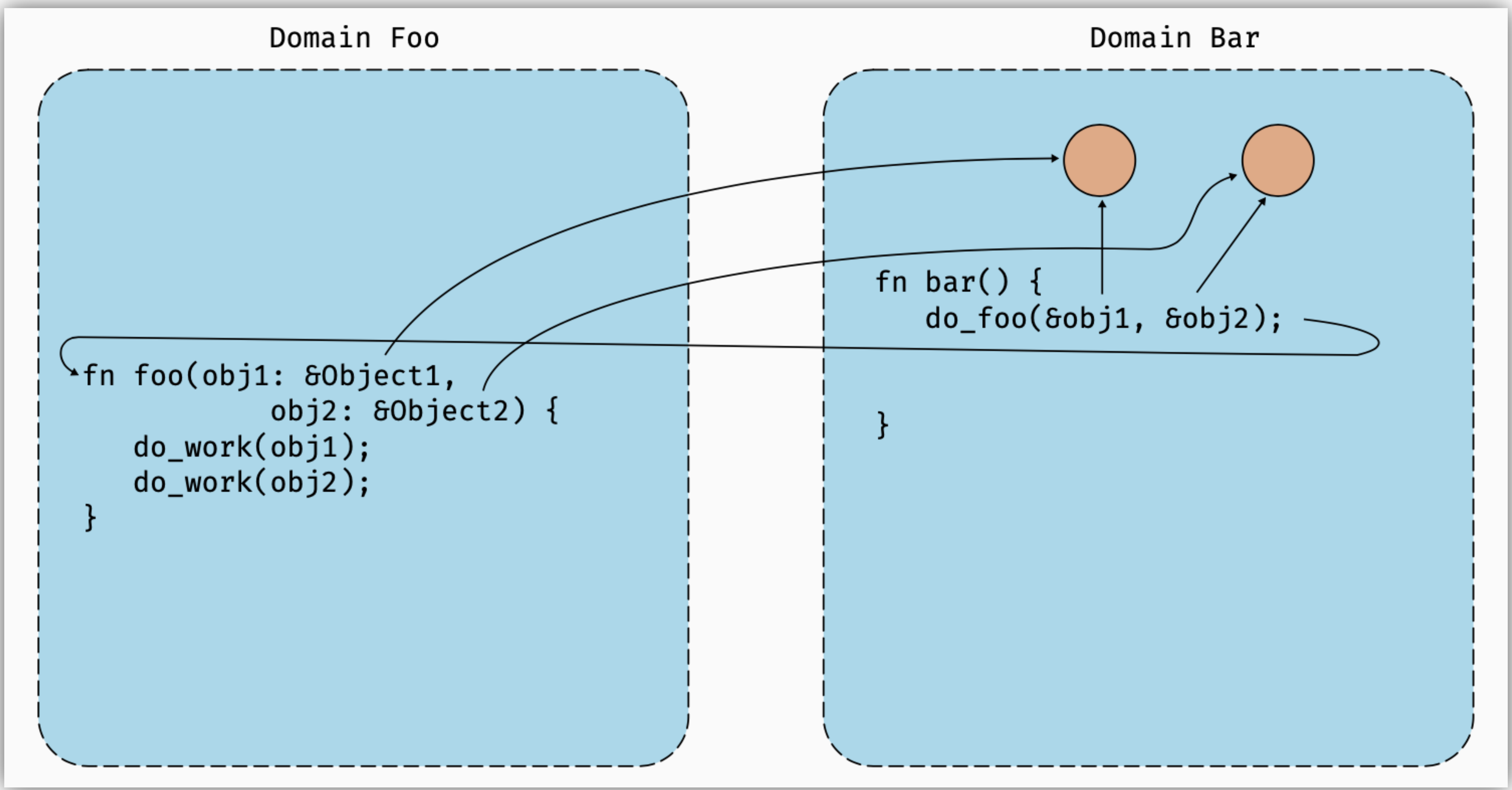
2.6 Fault Isolation in Language-based Systems



```
fn foo(obj1: &Object1,
           obj2: &Object2) {
   do_work(obj1);
   do_work(obj2);
}
```

```
fn bar() {
   do_foo(&obj1, &obj2);

}
```

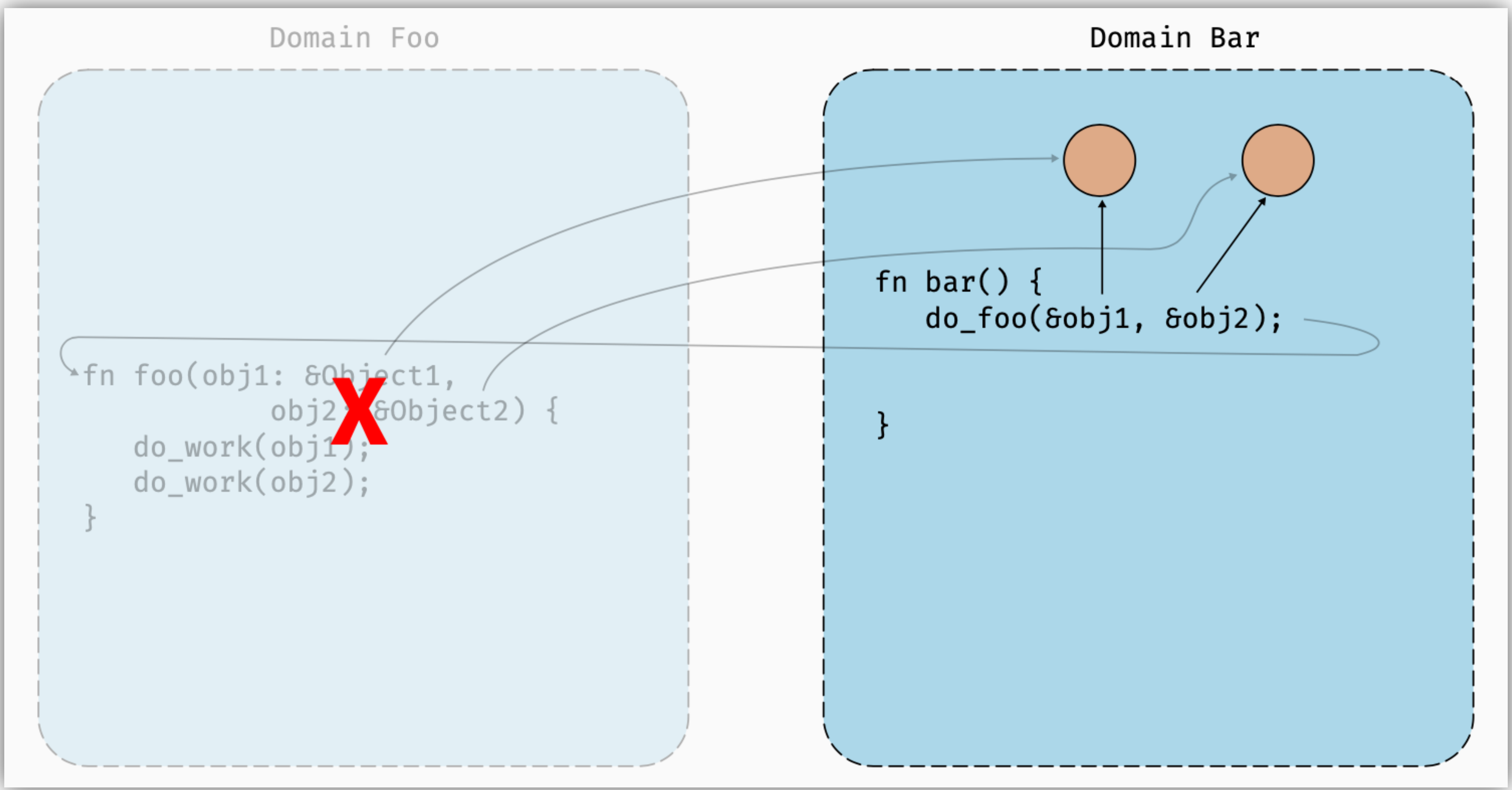
2.6 Fault Isolation in Language-based Systems



```
fn foo(obj1: &Object1,
           obj2: &Object2) {
   do_work(obj1);
   do_work(obj2);
}
```

```
fn bar() {
   do_foo(&obj1, &obj2);

}
```

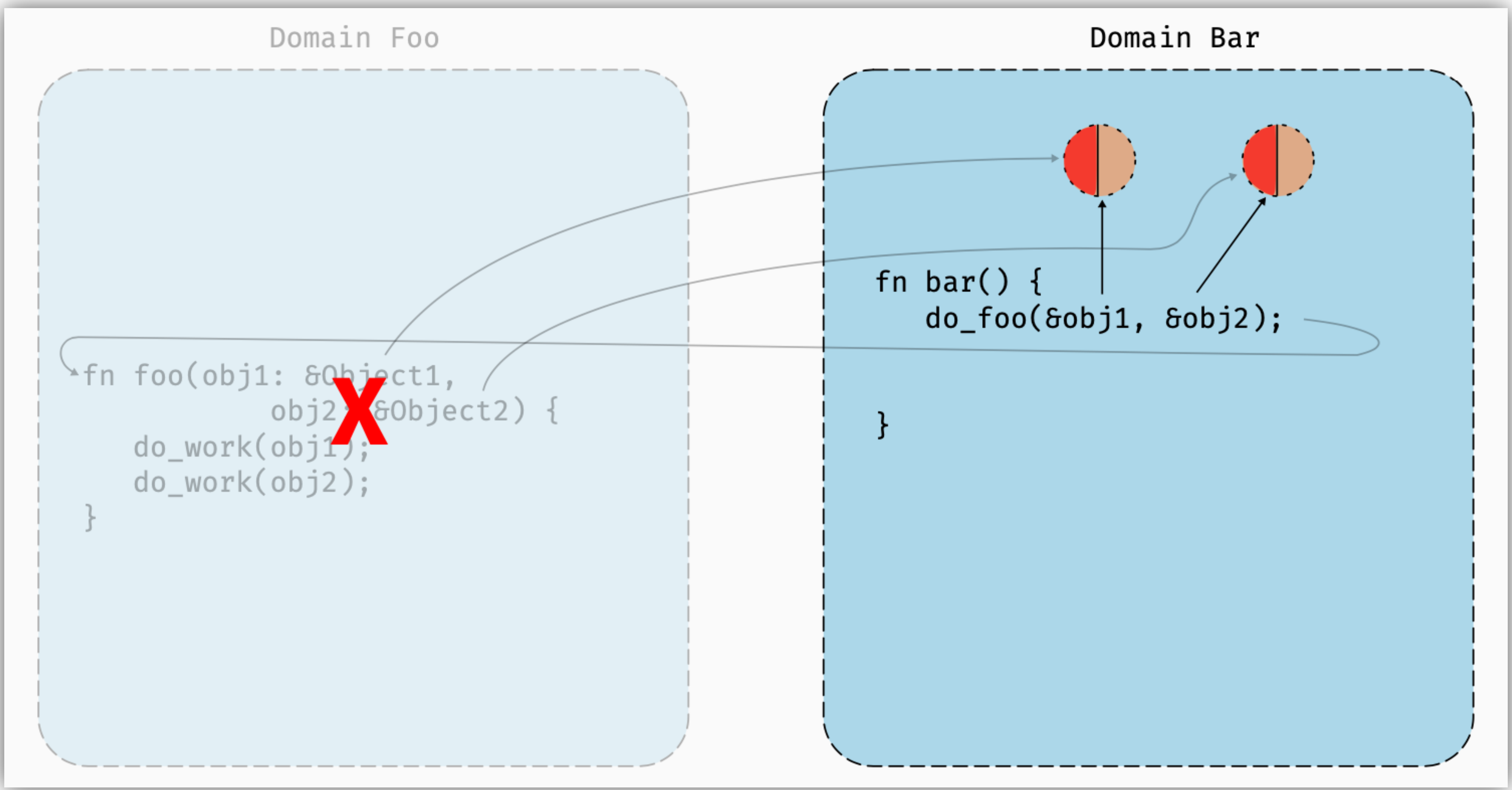
2.6 Fault Isolation in Language-based Systems



```
fn foo(obj1: &Object1,
           obj2: &Object2) {
    do_work(obj1);
    do_work(obj2);
}
```

```
fn bar() {
    do_foo(&obj1, &obj2);

}
```

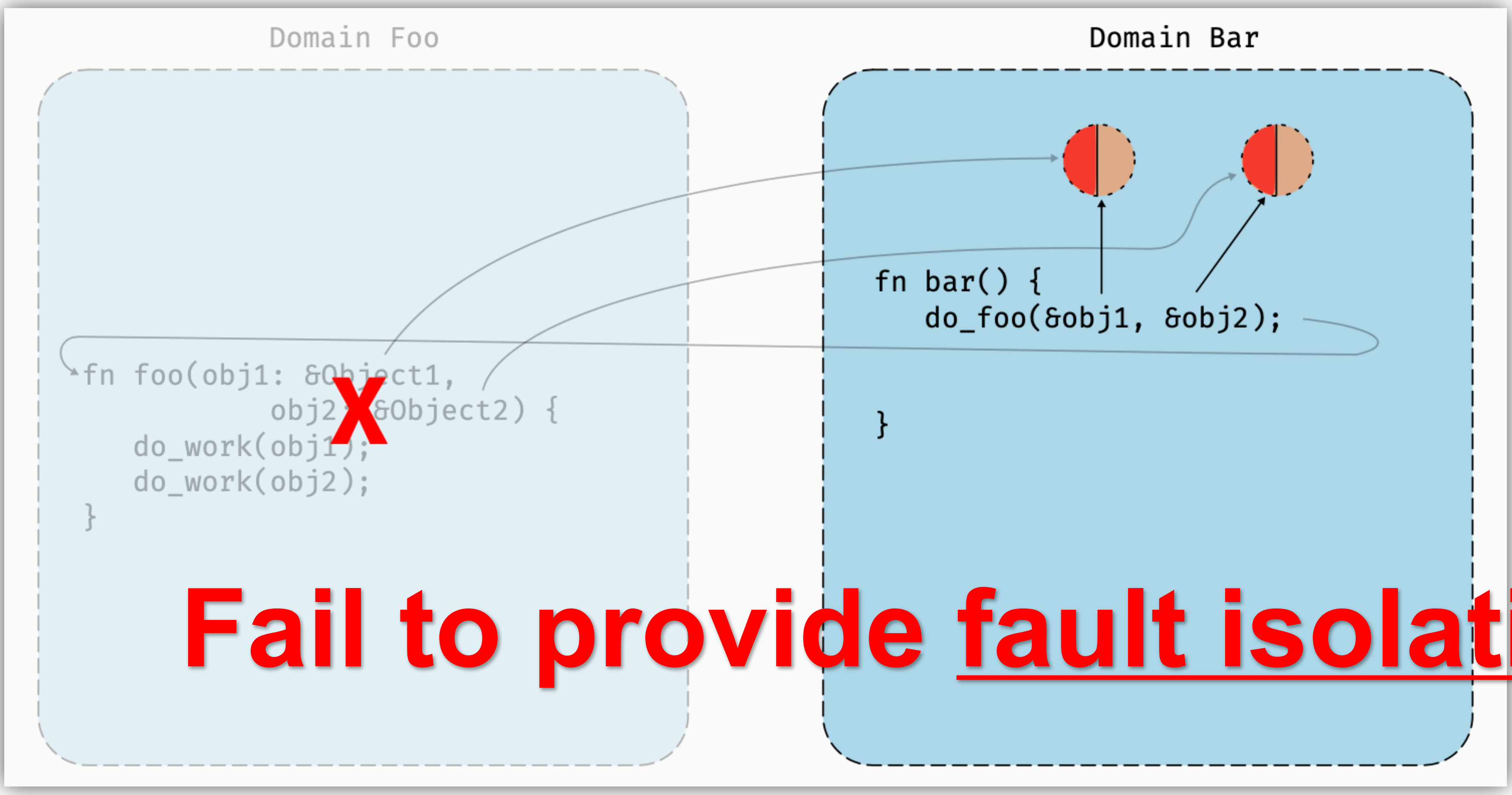
2.6 Fault Isolation in Language-based Systems



```
fn foo(obj1: &Object1,
           obj2: &Object2) {
    do_work(obj1);
    do_work(obj2);
}
```

```
fn bar() {
    do_foo(&obj1, &obj2);

}
```

2.6 Fault Isolation in Language-based Systems

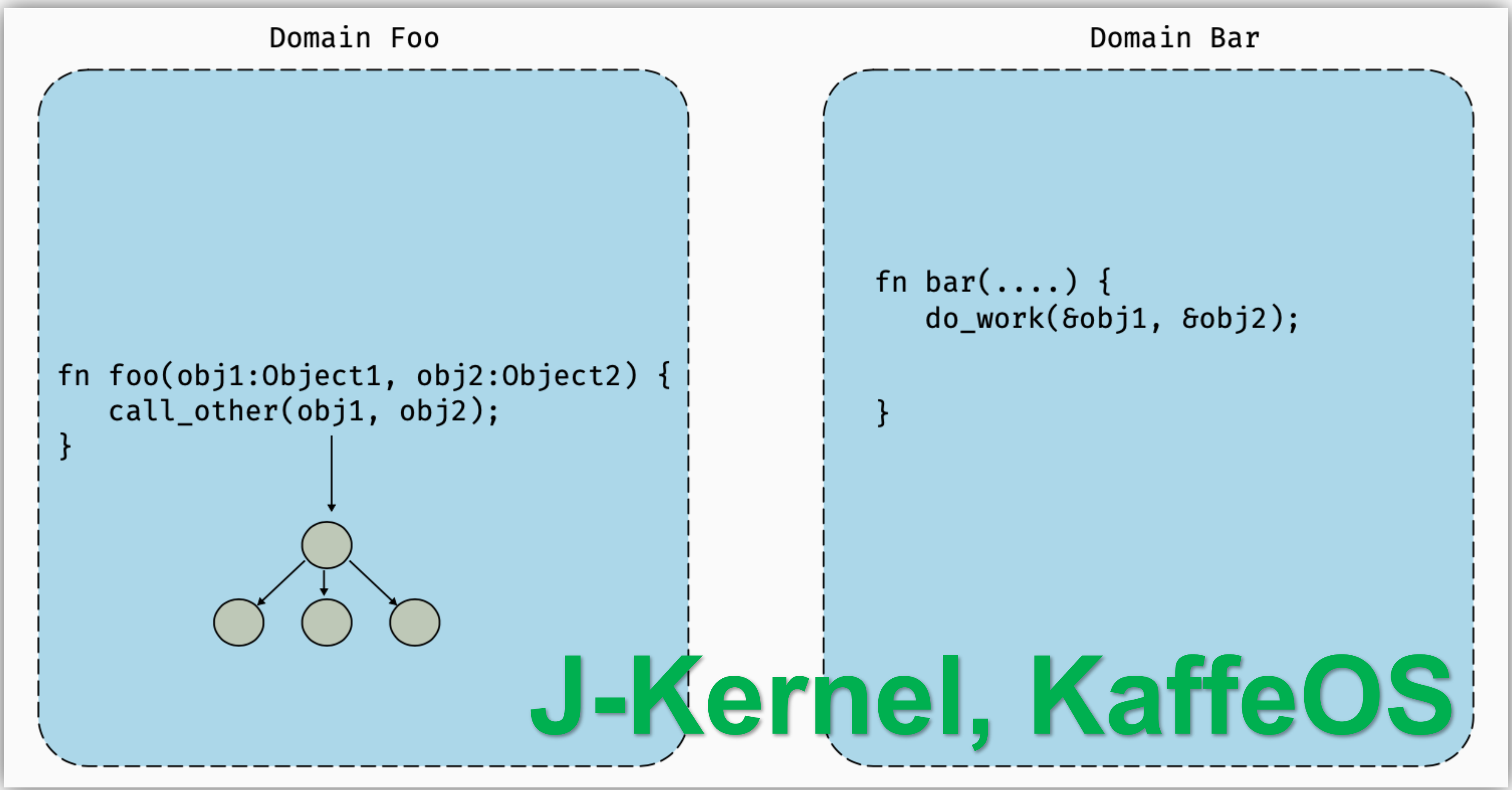


```
fn foo(obj1: &Object1,
           obj2: &Object2) {
    do_work(obj1);
    do_work(obj2);
}
```

```
fn bar() {
    do_foo(&obj1, &obj2);

}
```

Fail to provide <u>fault isolation</u>

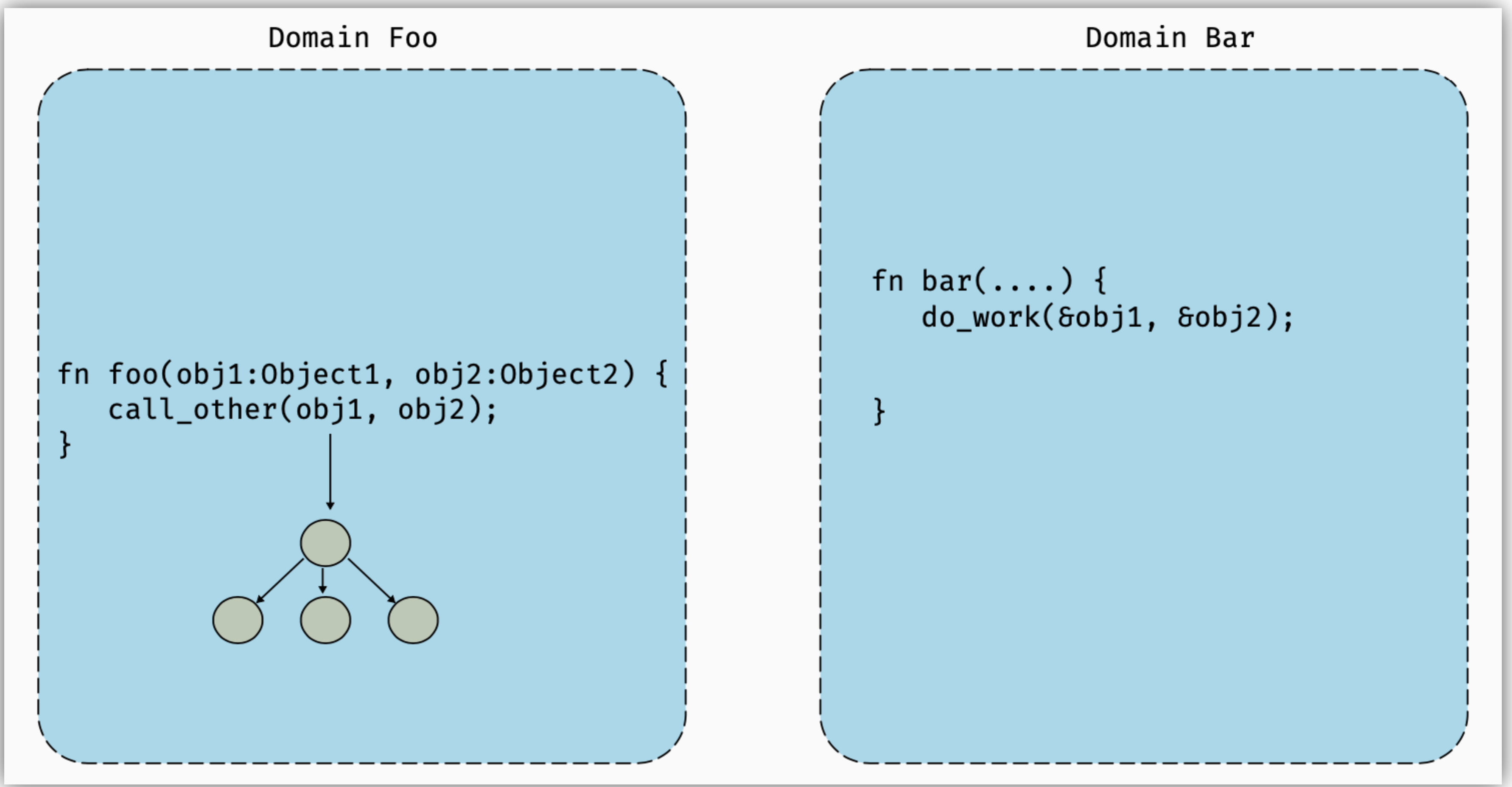
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

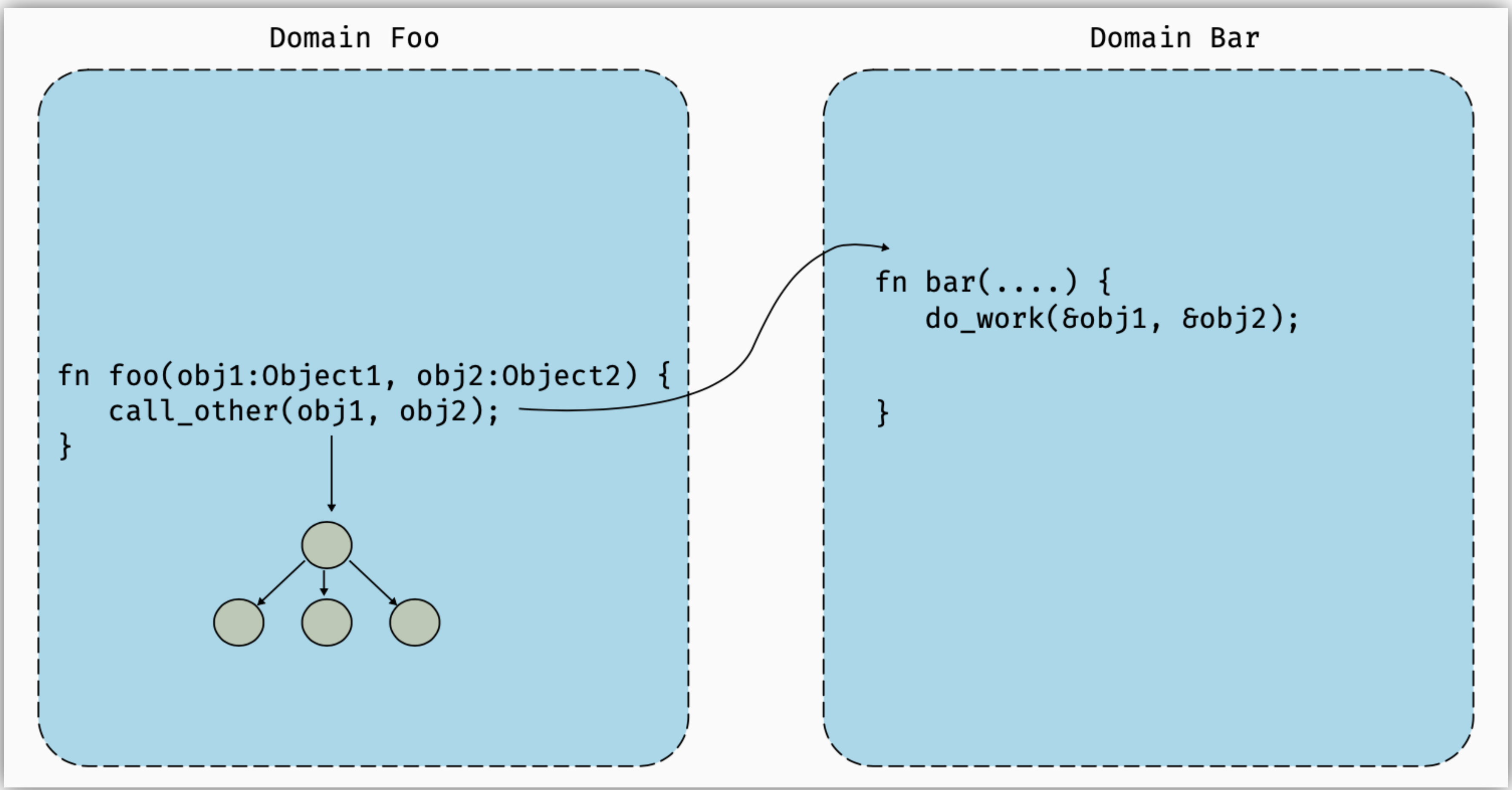
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

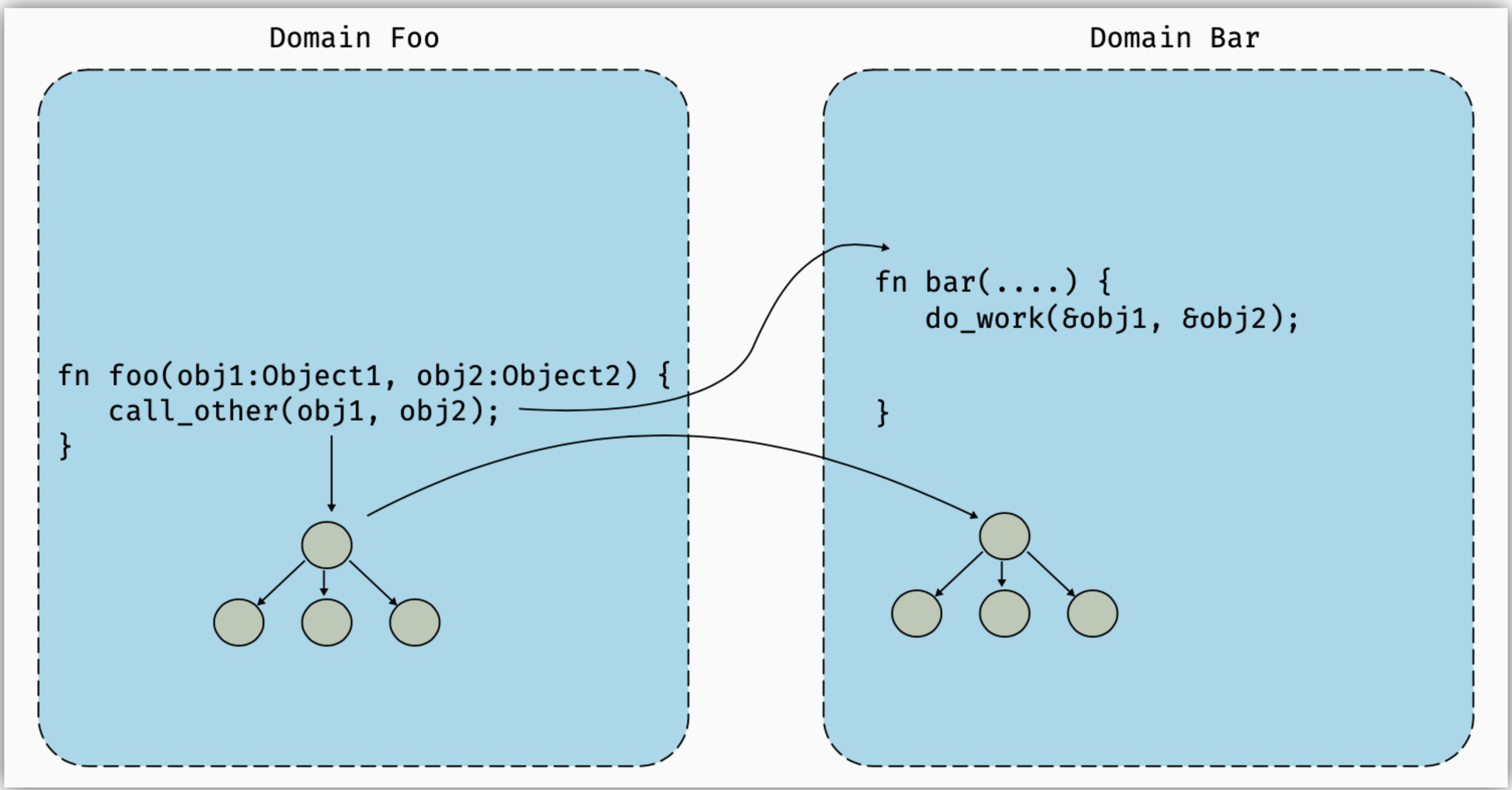
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

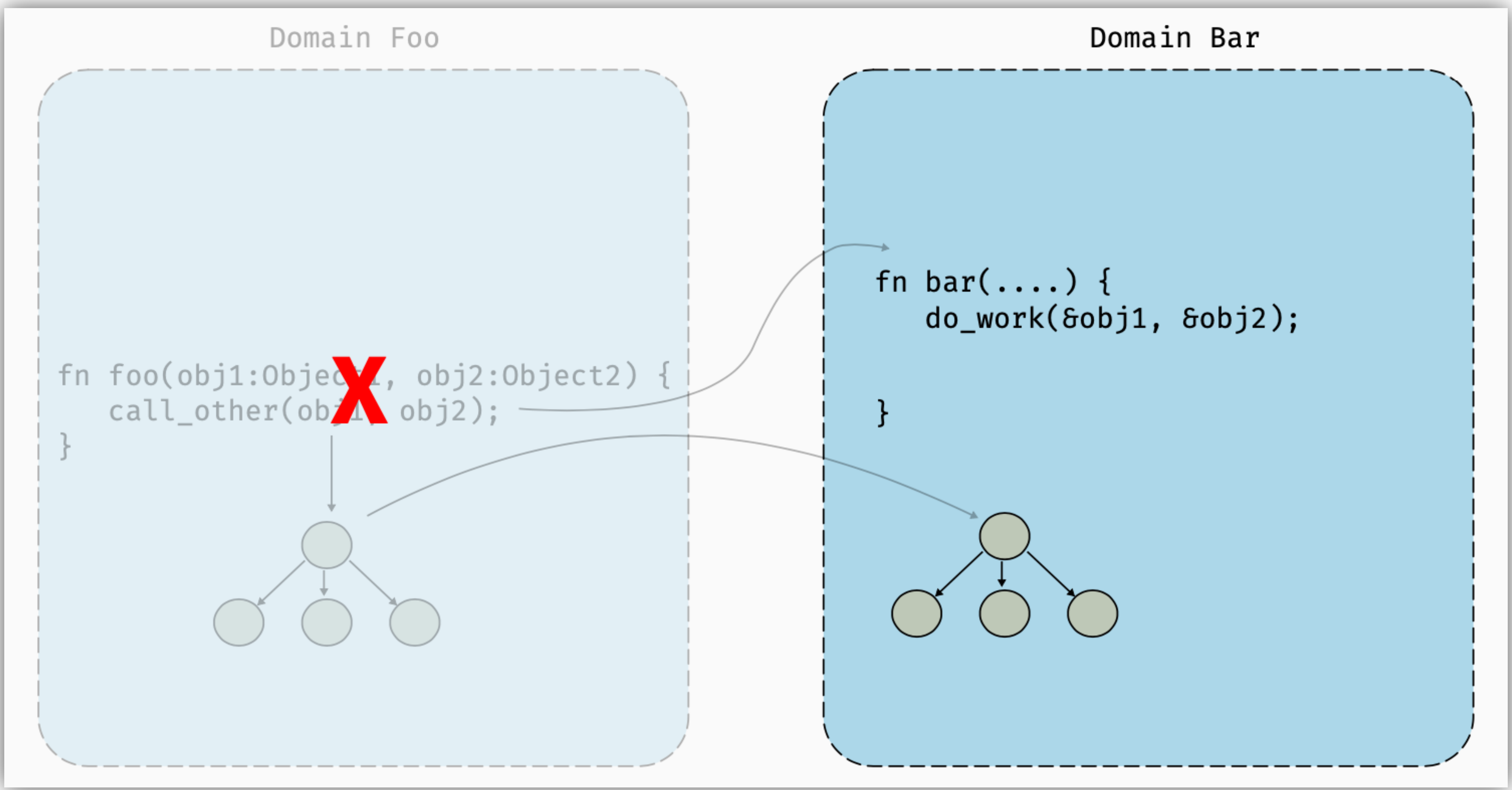
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

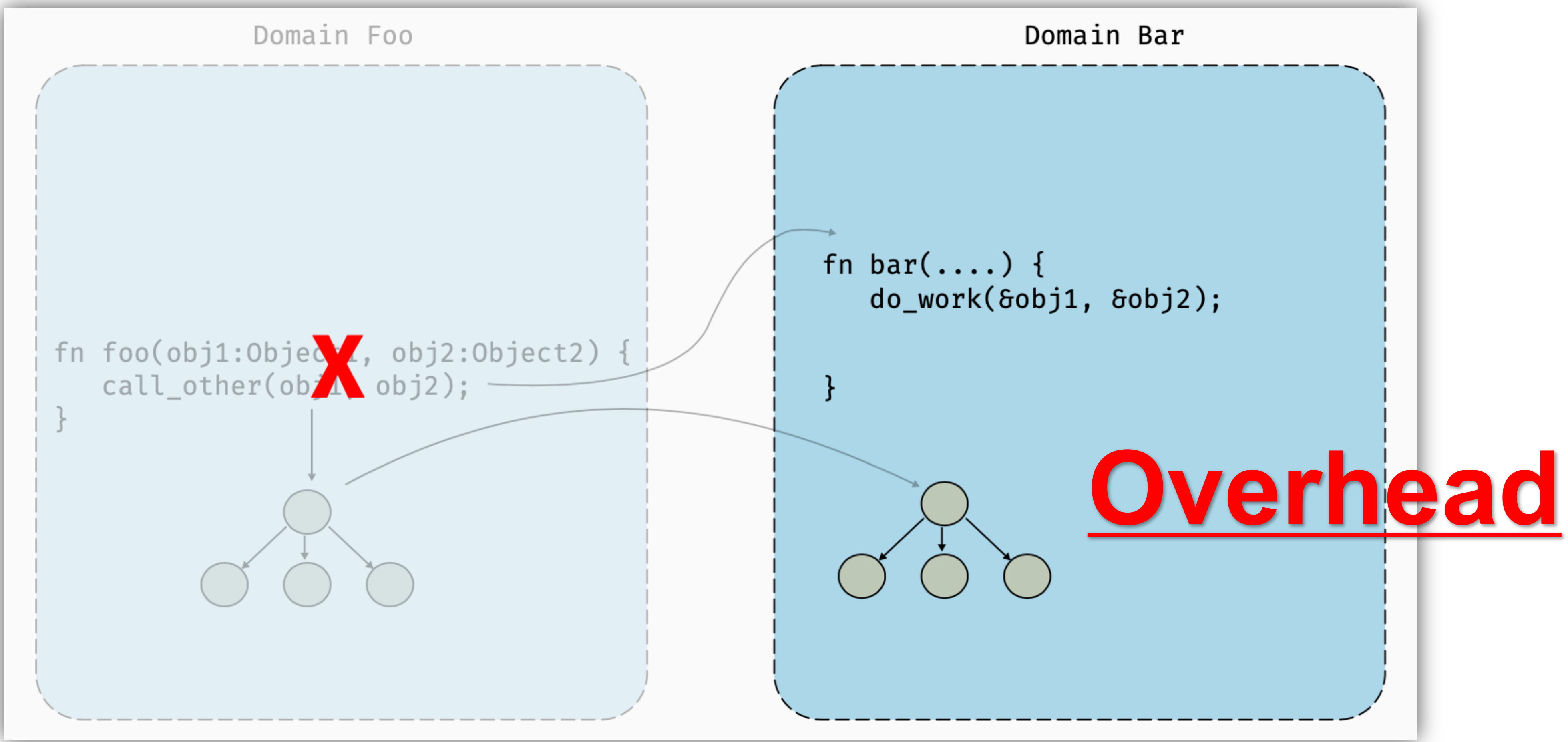
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

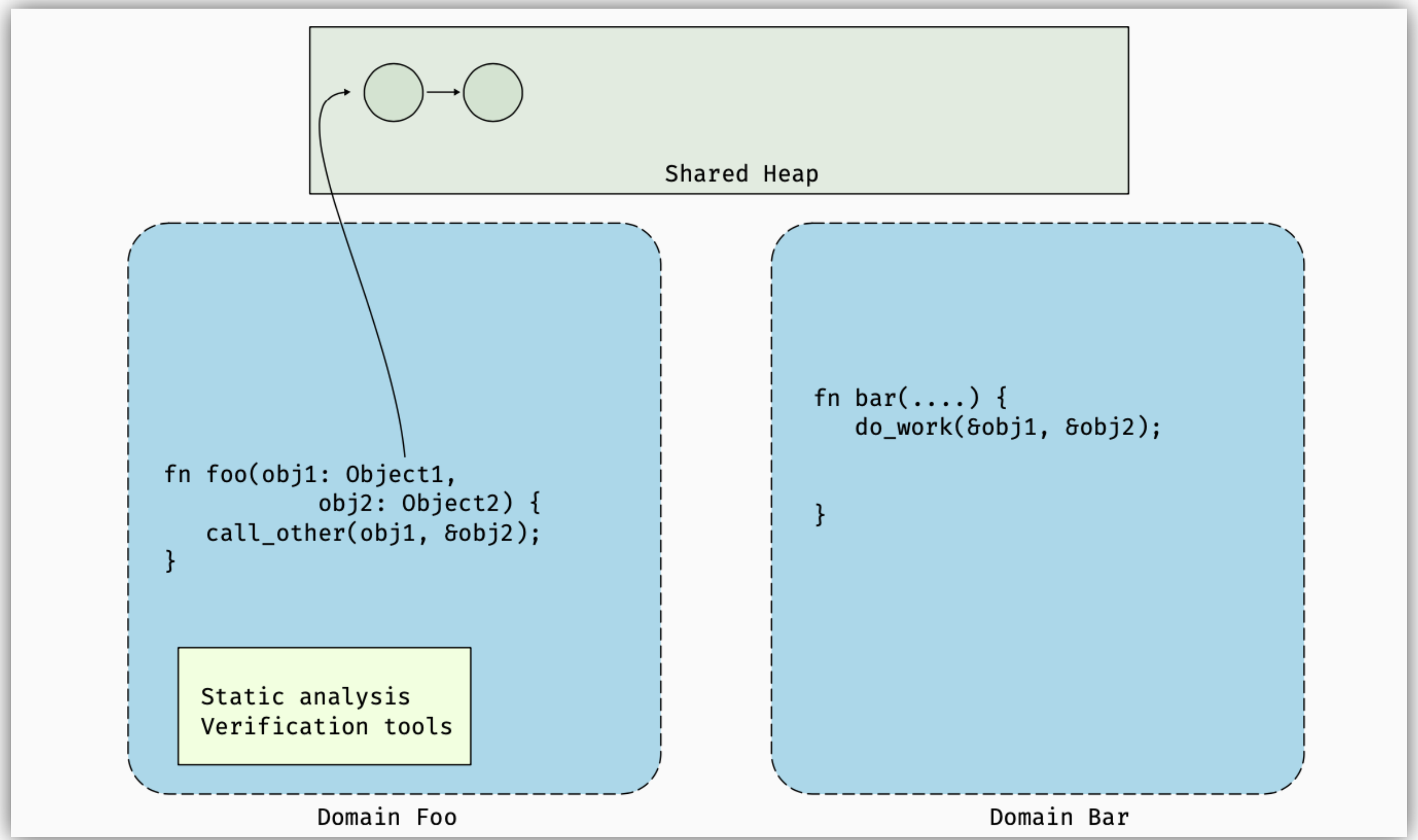
2.6 Fault Isolation in Language-based Systems: Deep Copy



```
fn foo(obj1:Object1, obj2:Object2) {
    call_other(obj1, obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

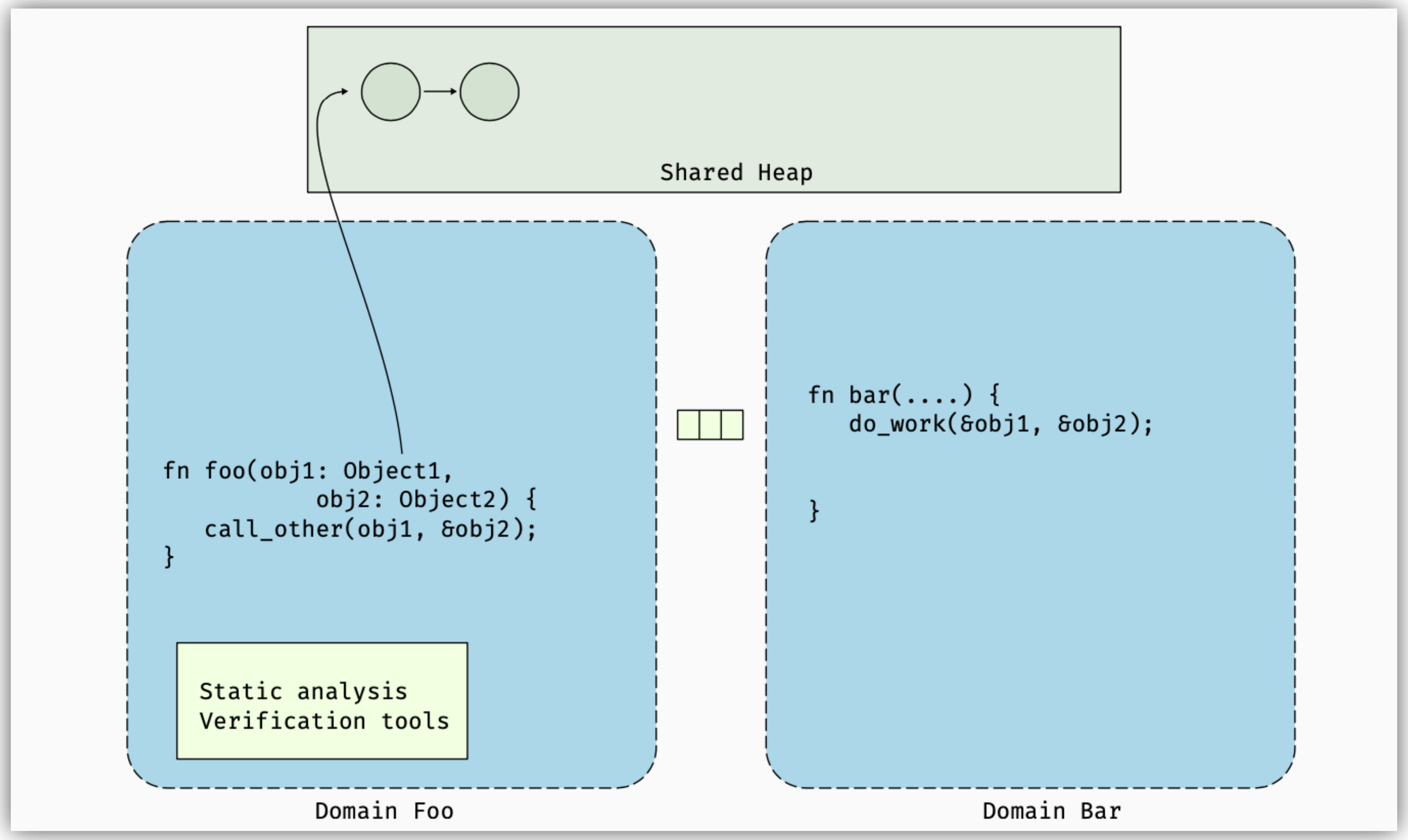
2.6 Fault Isolation in Language-based Systems: Singularity



```
fn foo(obj1: Object1,
           obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);


}
```

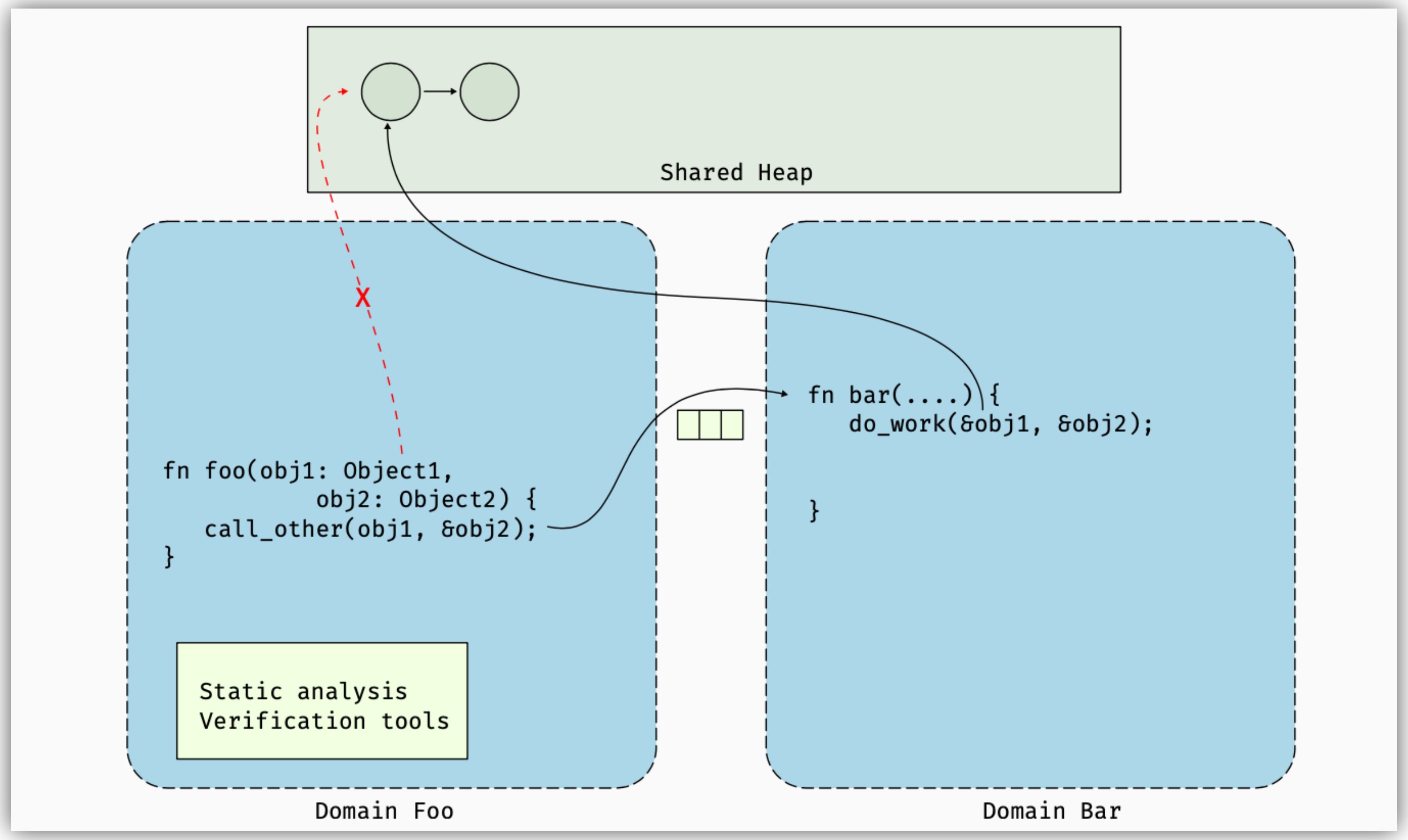
2.6 Fault Isolation in Language-based Systems: Singularity



```
fn foo(obj1: Object1,
            obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);


}
```

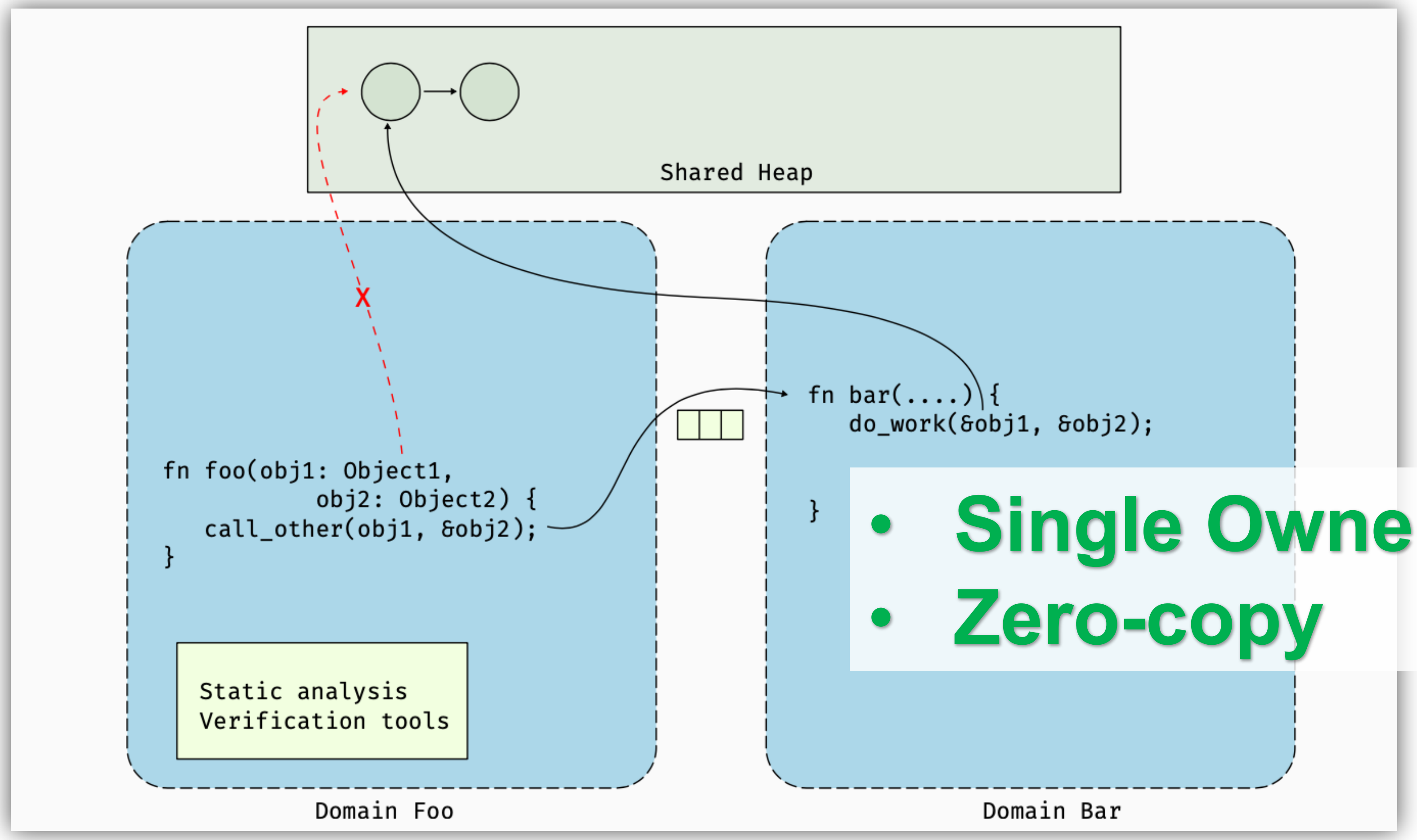
2.6 Fault Isolation in Language-based Systems: Singularity



```
fn foo(obj1: Object1,
             obj2: Object2) {
    call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

2.6 Fault Isolation in Language-based Systems: Singularity



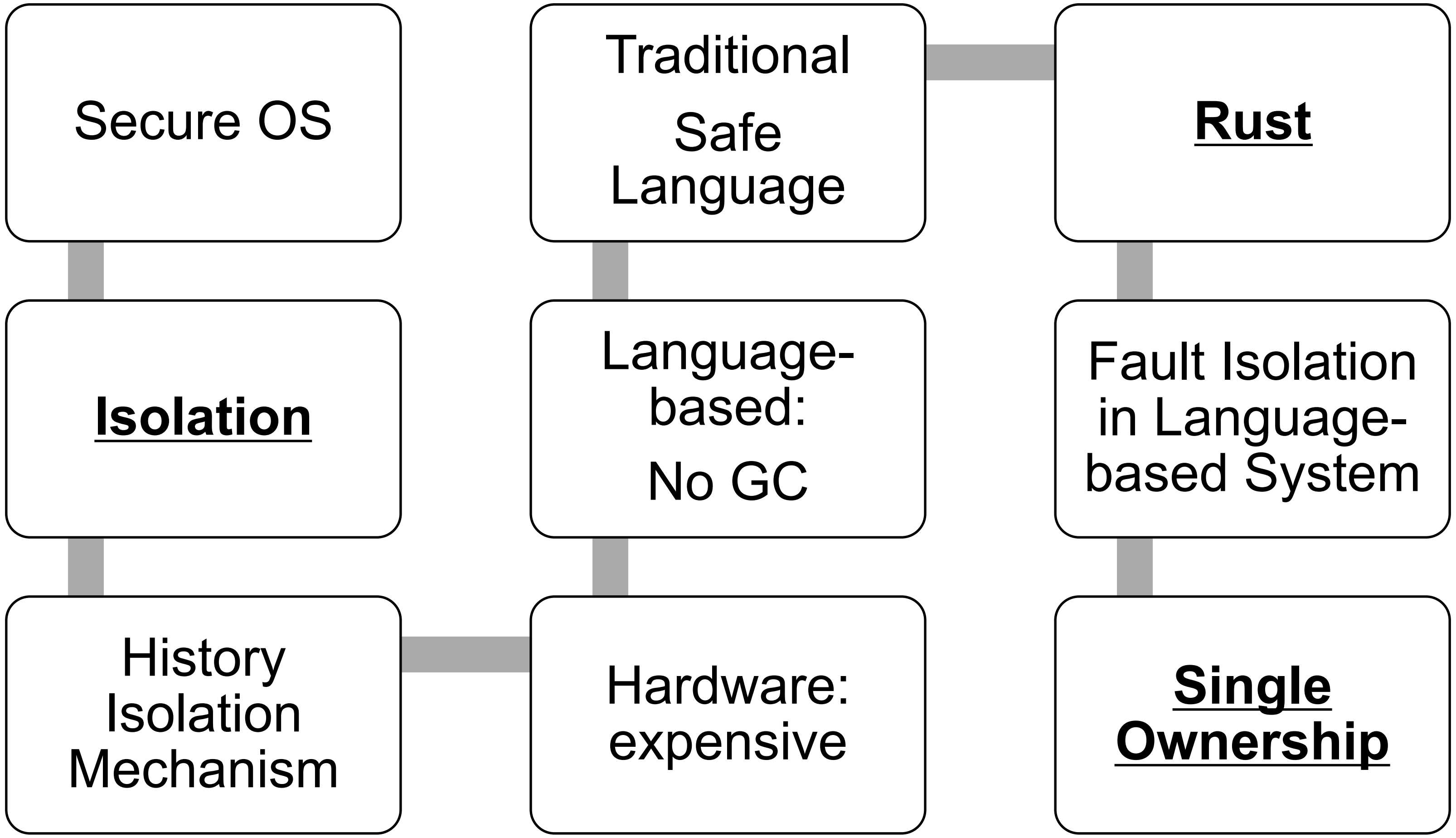
```
fn foo(obj1: Object1,
            obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

```
Static analysis
Verification tools
```

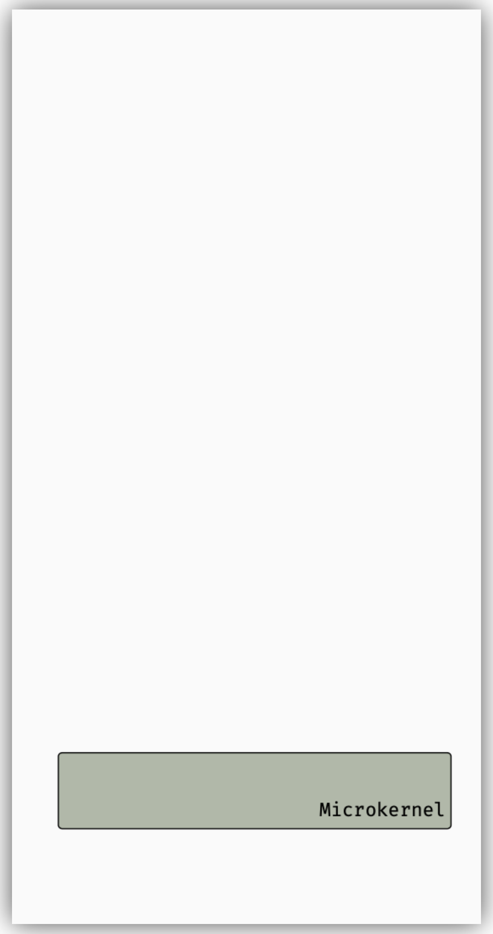
- **Single Ownership**
- **Zero-copy**

2.7 Summary of Background

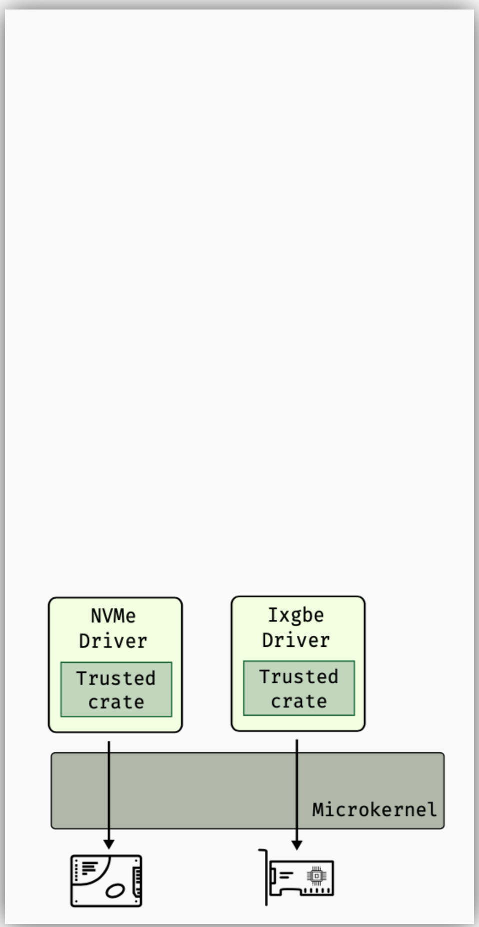

Secure OS
Isolation
History Isolation Mechanism
Hardware: expensive
Language-based: No GC
Traditional Safe Language
Rust
Fault Isolation in Language-based System
Single Ownership

3. RedLeaf

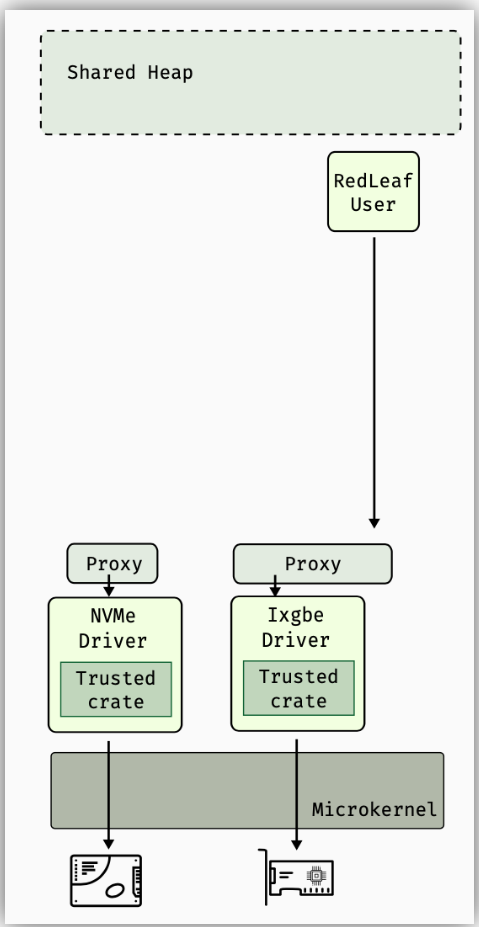
3.1 Architecture



3.1 Architecture

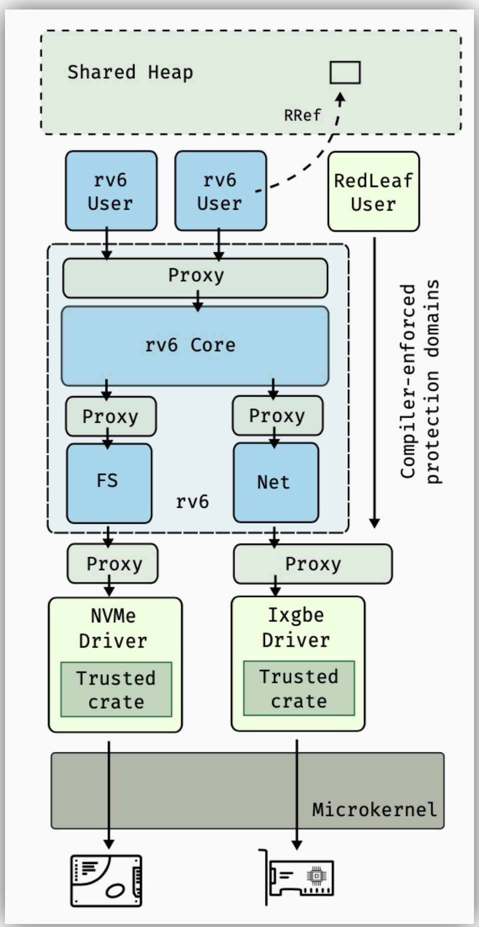


3.1 Architecture



RedLeaf: Architecture

3.1 Architecture



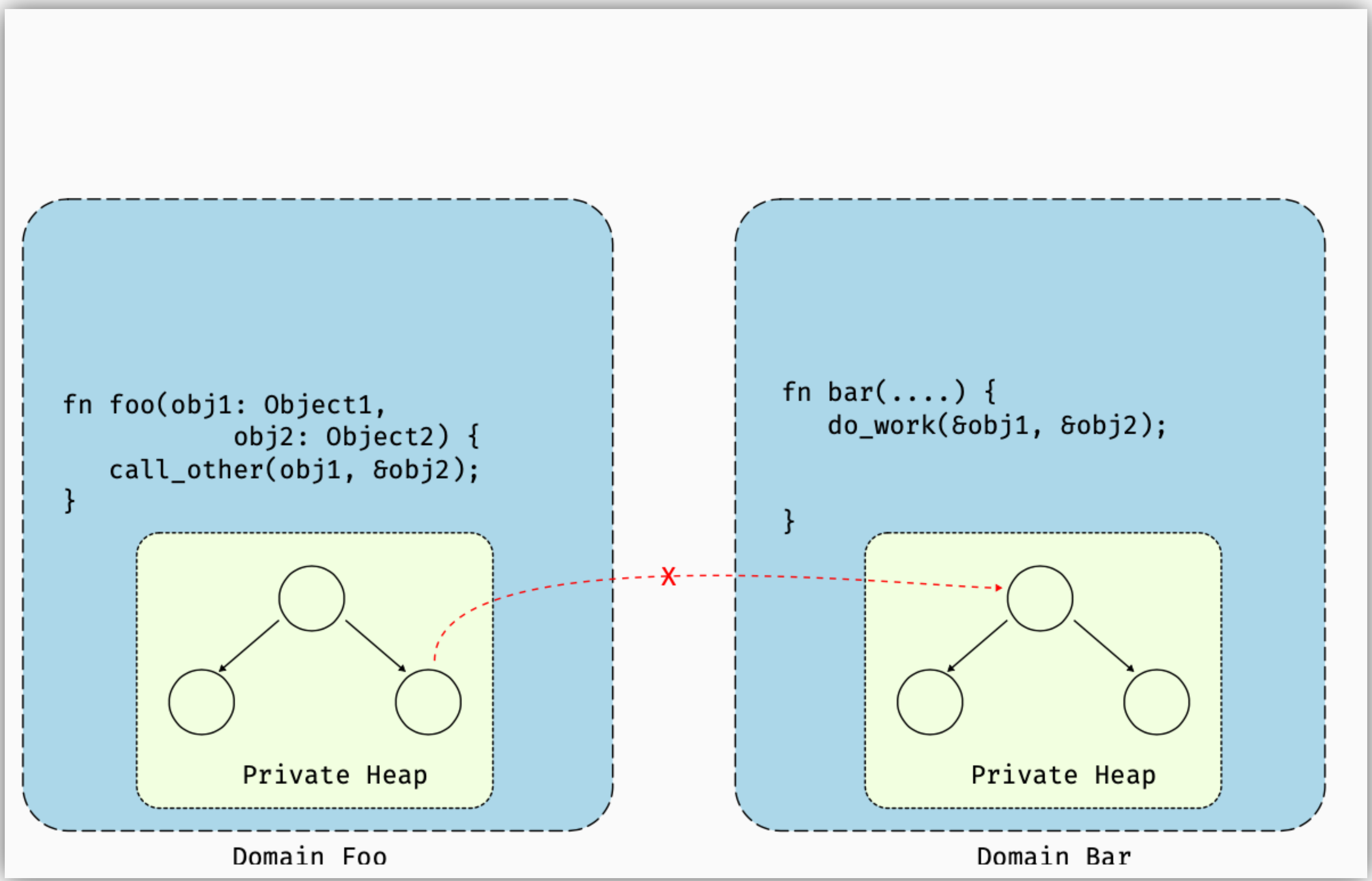
3.2 * Trust Base

- Rust **compiler**
- Rust core **libraries** (crates)
- Non-malicious devices (can be spared by IOMMU)

3.3 Fault Isolation

- After a domain crash
  - Unwind all threads running inside
  - Subsequent invocations return error
  - All resources are deallocated
  - Other threads continue execution

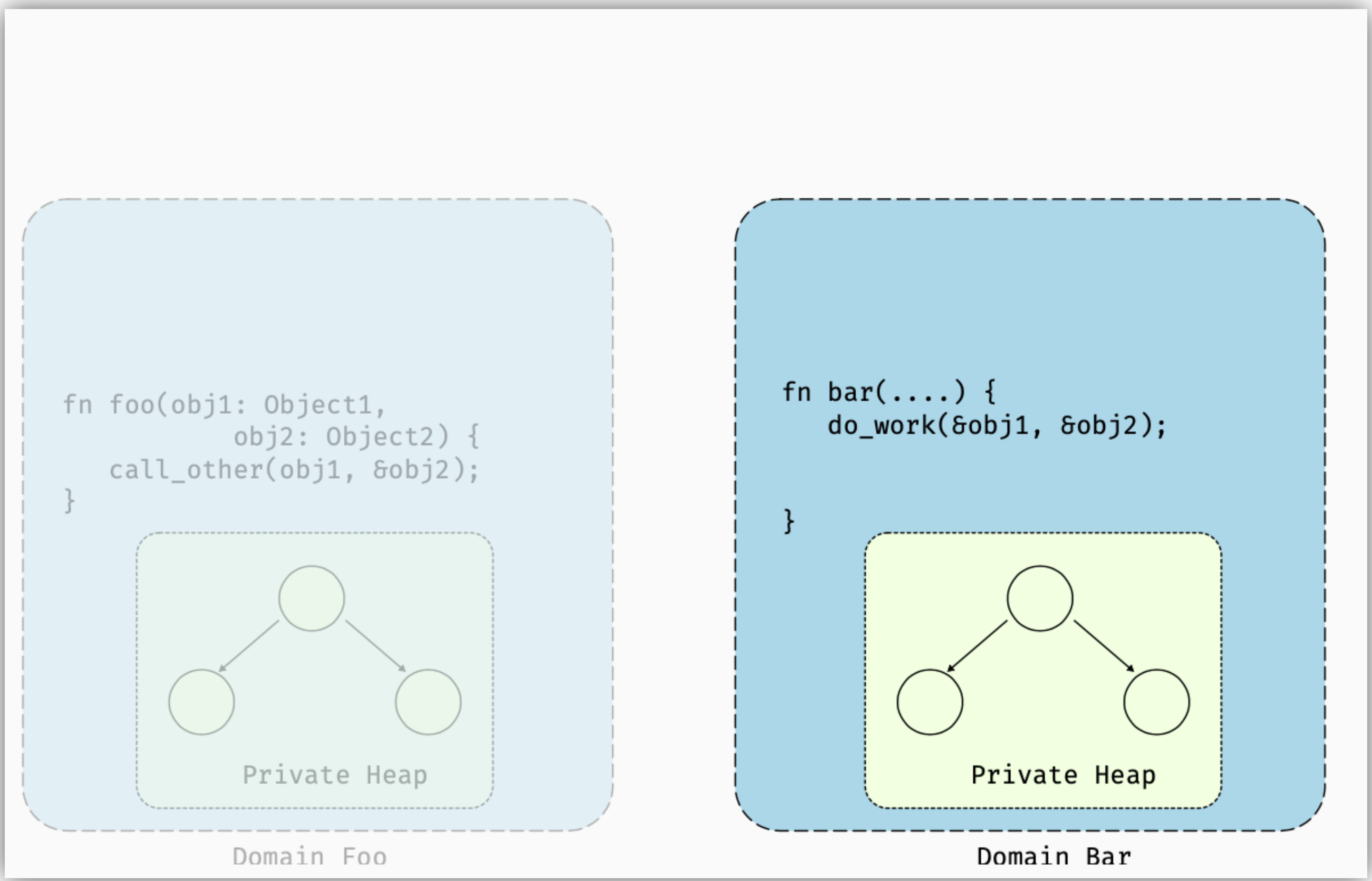
3.4 Heap Isolation



```
fn foo(obj1: Object1,
           obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.4 Heap Isolation

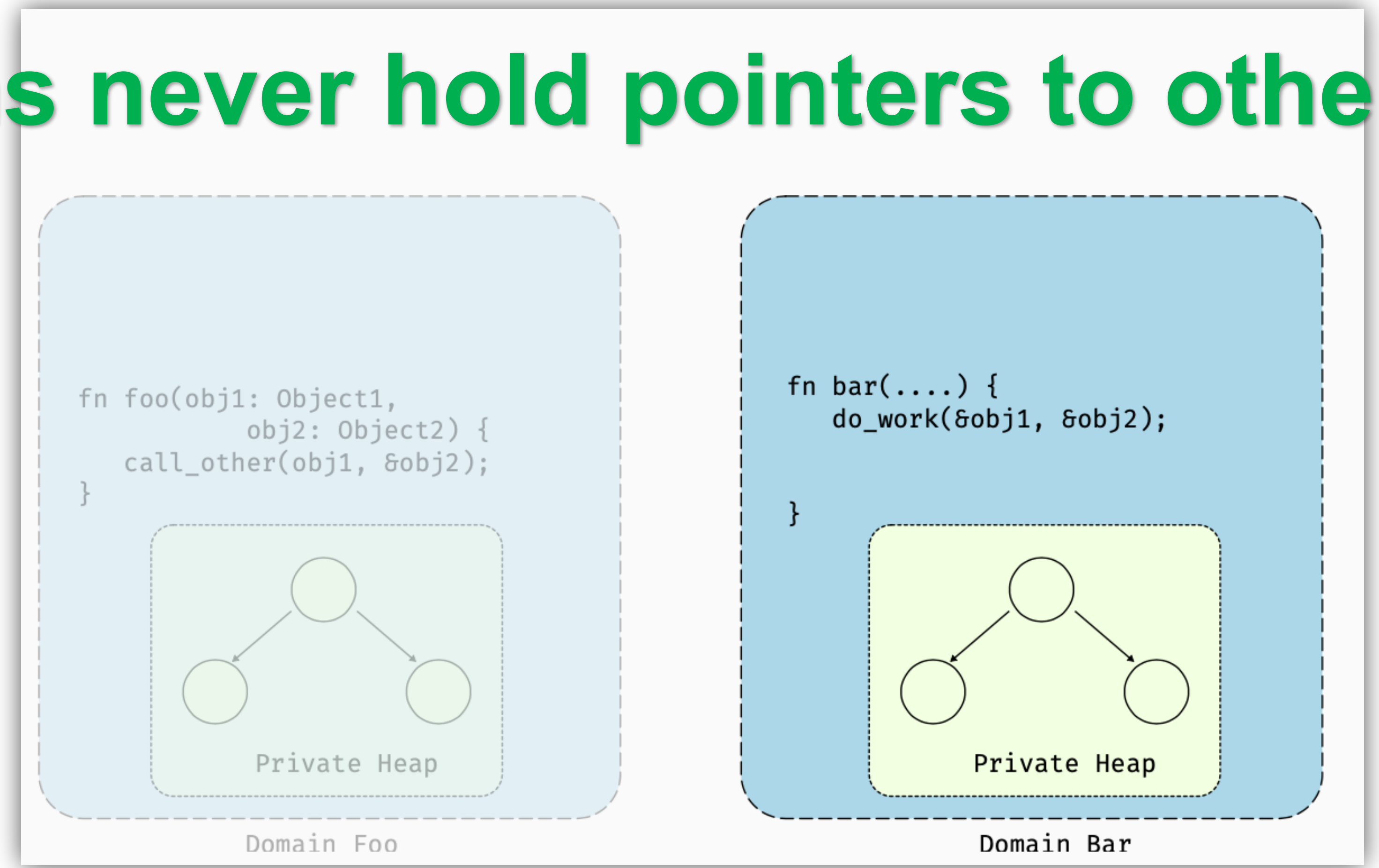


```
fn foo(obj1: Object1,
       obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);


}
```

3.4 Heap Isolation

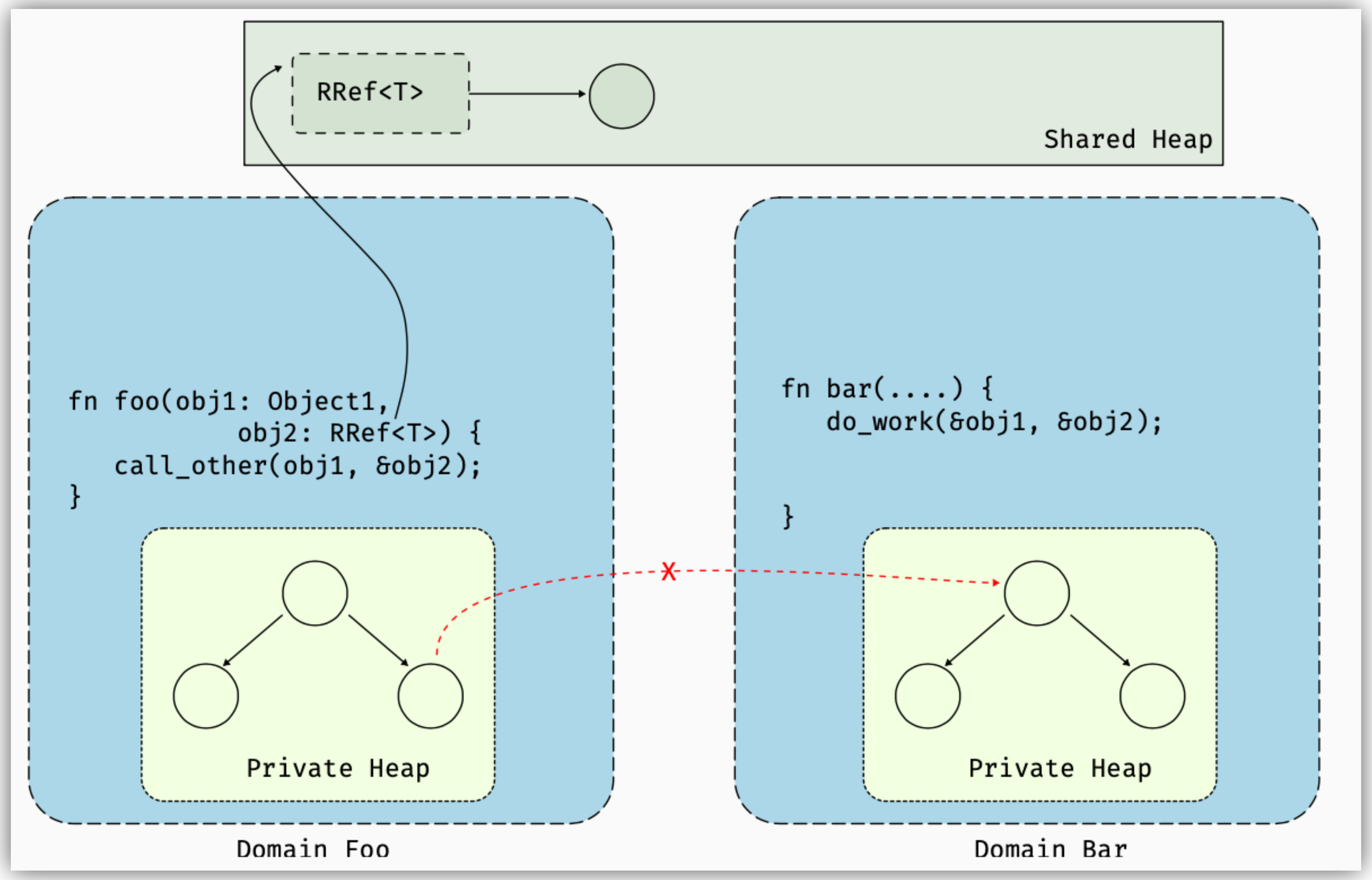
Domains never hold pointers to other domains



```
fn foo(obj1: Object1,
       obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

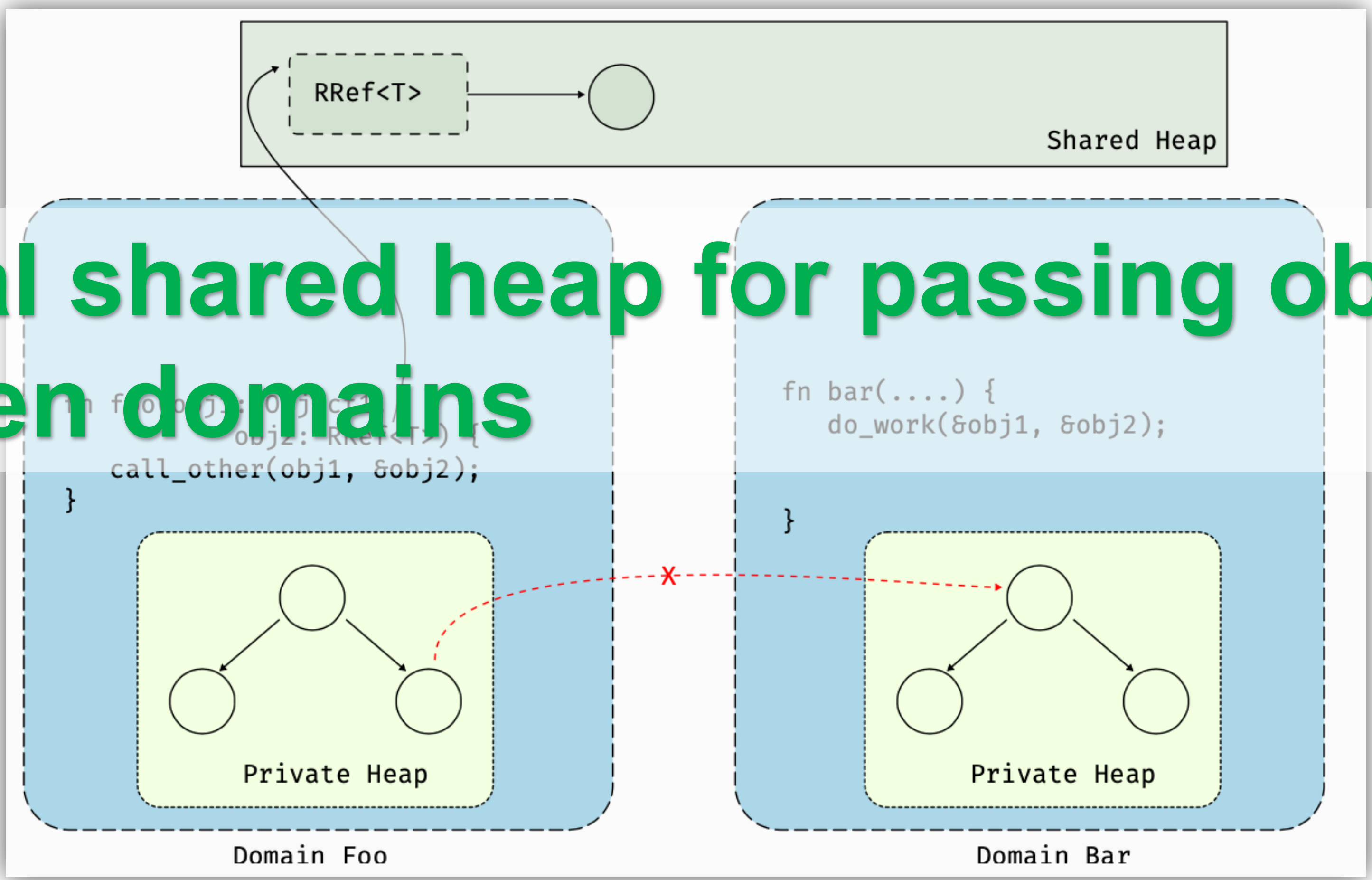
3.4 Heap Isolation



```
fn foo(obj1: Object1,
          obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.4 Heap Isolation

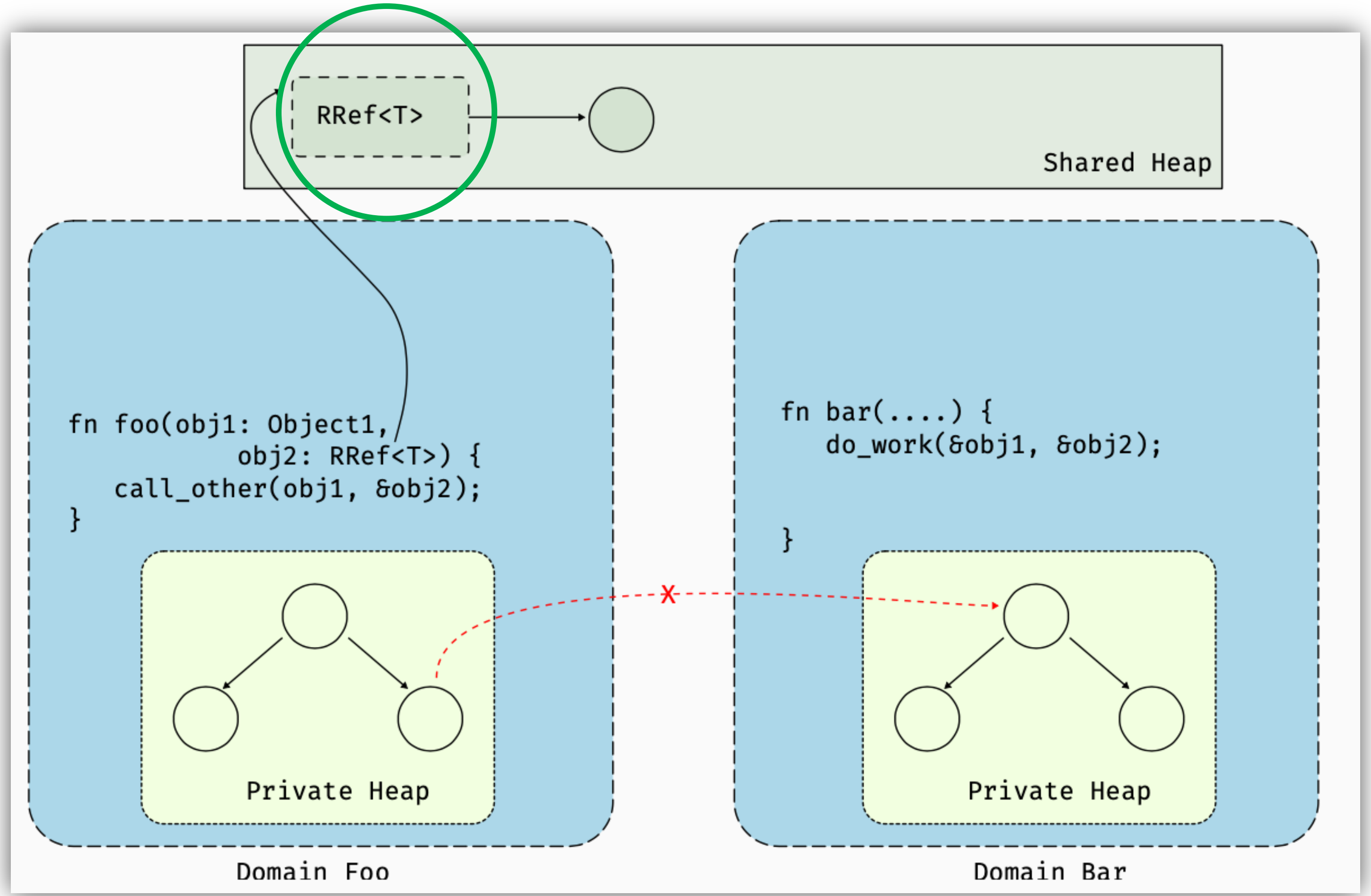


```
fn foo(obj1: Object1,
        obj2: RRef<T>) {
    call_other(obj1, &obj2);
}
```

```
fn bar(....) {
    do_work(&obj1, &obj2);

}
```

Special shared heap for passing objects between domains

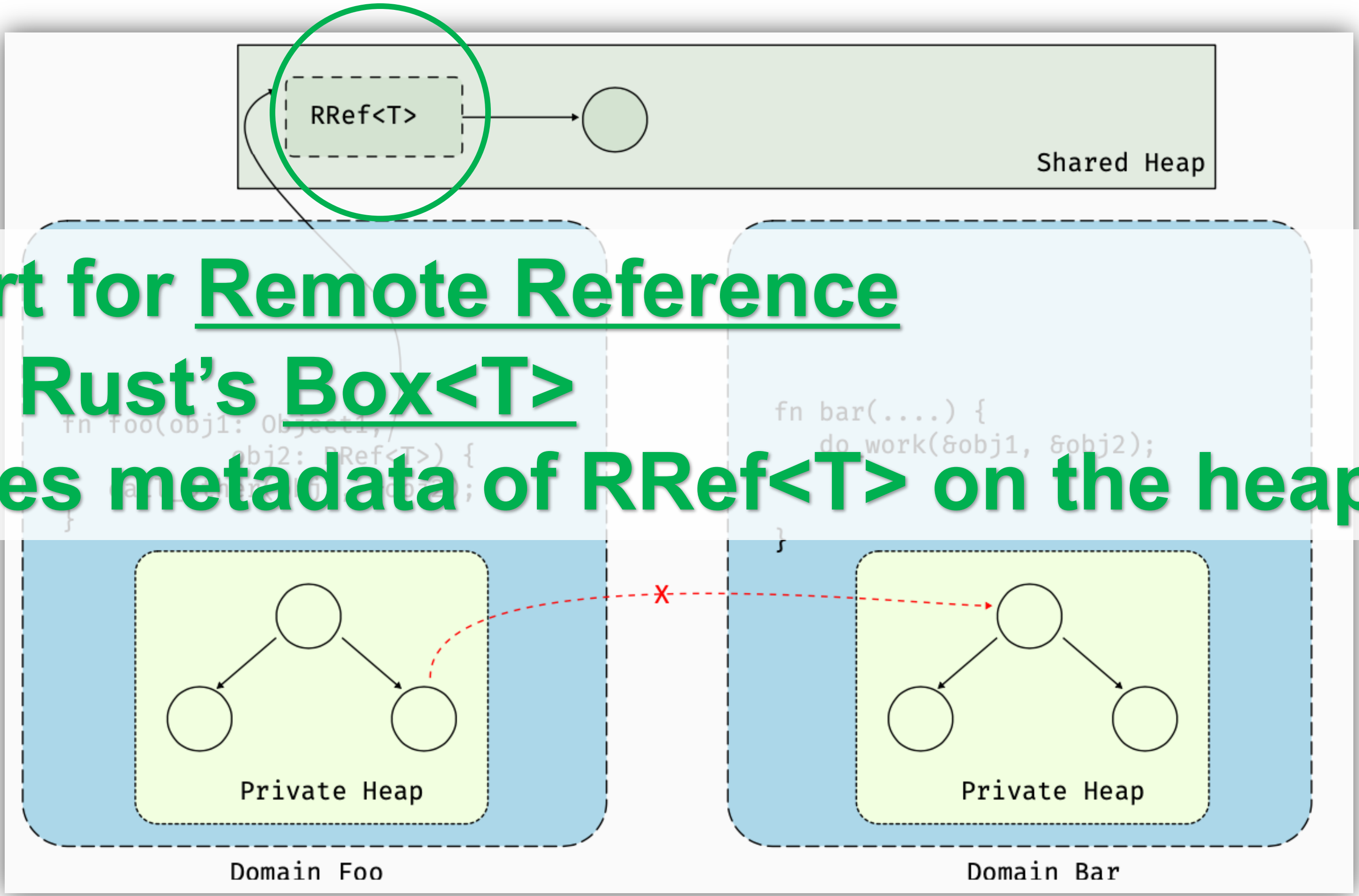
3.4 Heap Isolation



```
fn foo(obj1: Object1,
           obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.4 Heap Isolation

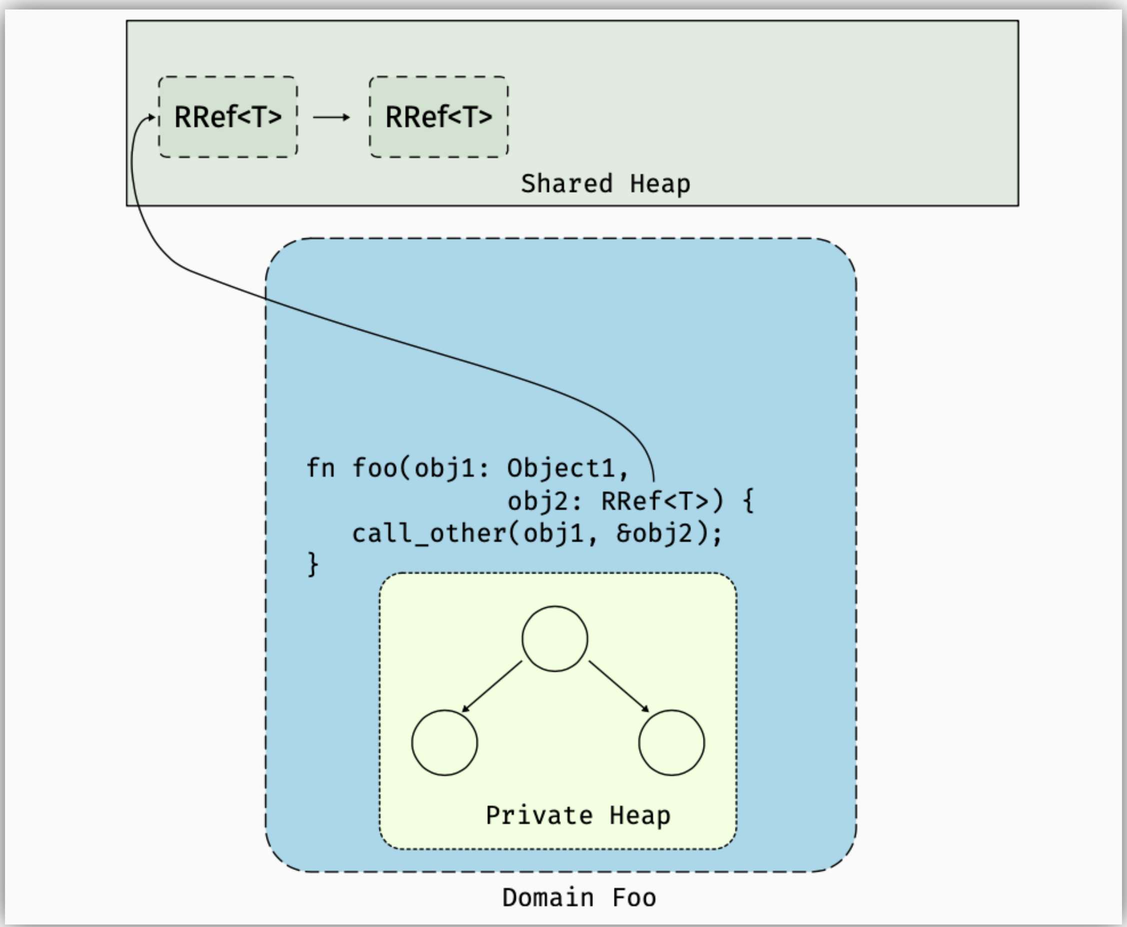


1. Short for Remote Reference
2. Like Rust's Box<T>
3. Stores metadata of RRef<T> on the heap

3.5 Exchangeable Types

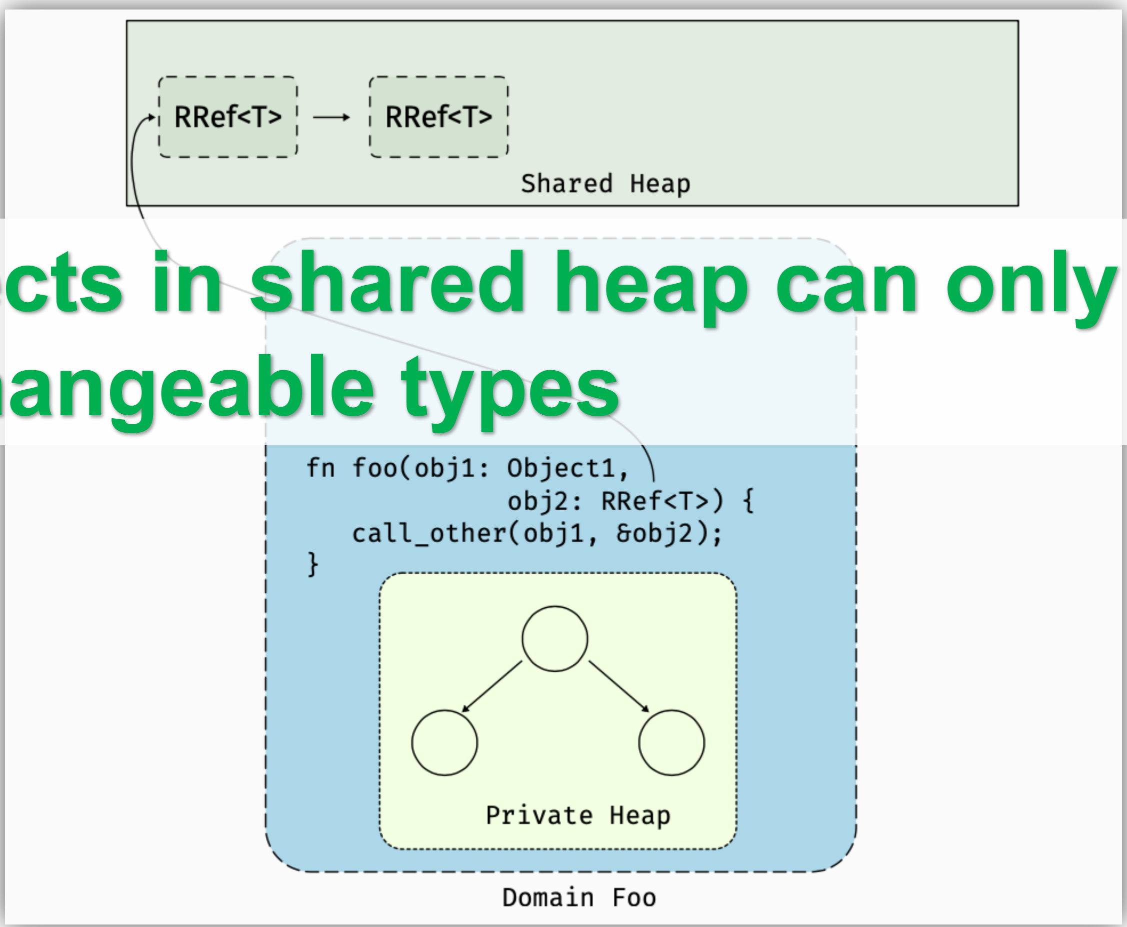
- Exchangeable Types can be:
  - **RRef<T>** itself
  - A subset of Rust primitive **Copy** types (not references or pointers)
  - **Composite types** constructed out of exchangeable types
  - References to traits **with methods that receive exchangeable types**

3.5 Exchangeable Types



```
fn foo(obj1: Object1,
           obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

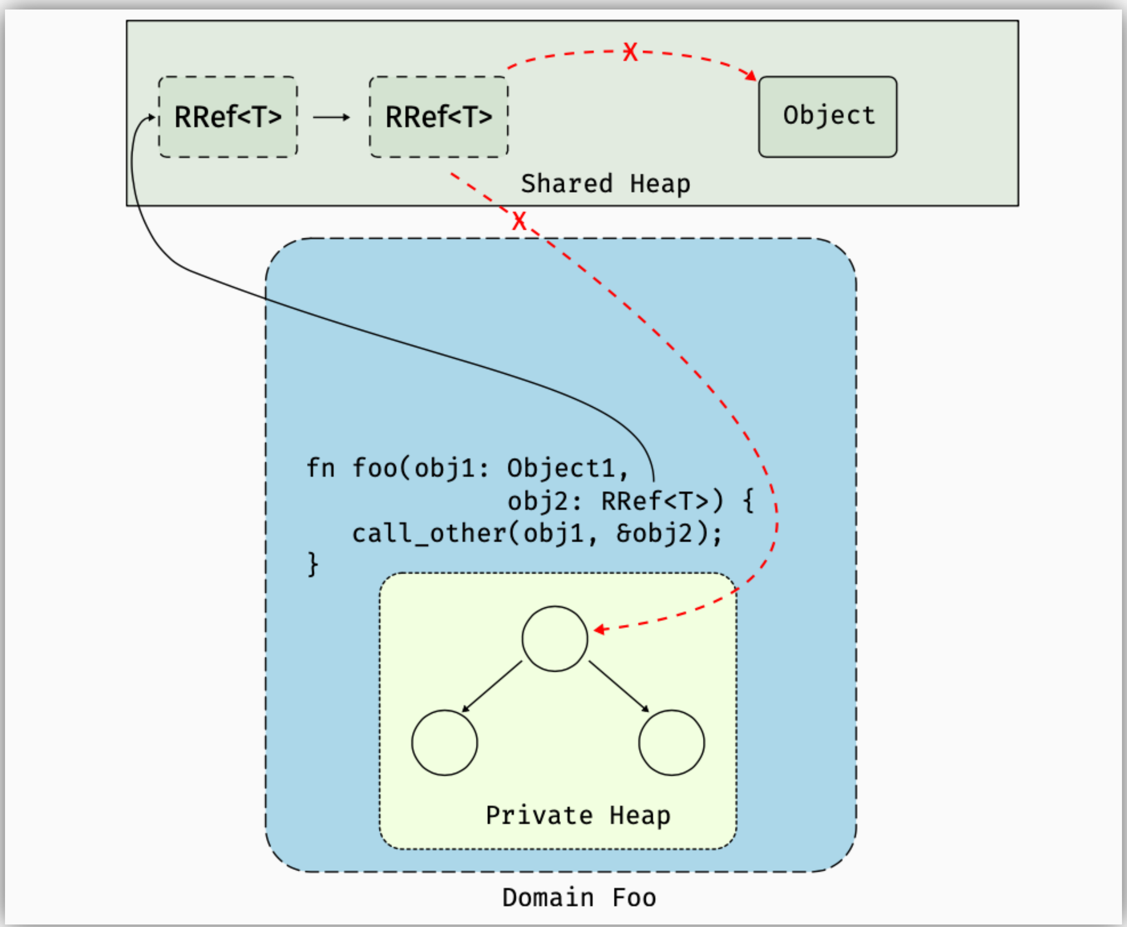
3.5 Exchangeable Types



```
fn foo(obj1: Object1,
           obj2: RRef<T>) {
    call_other(obj1, &obj2);
}
```

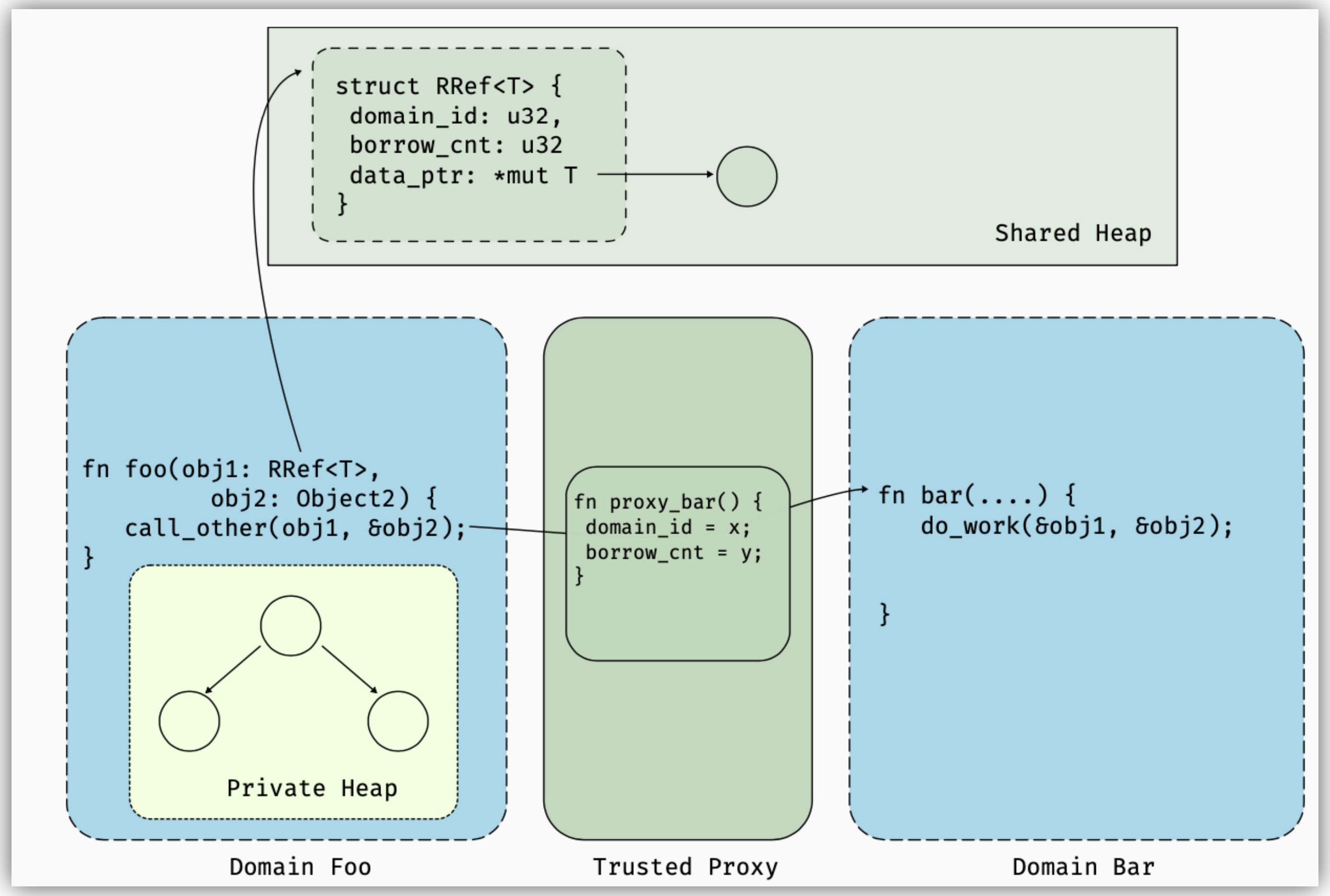
Objects in shared heap can only be exchangeable types

3.5 Exchangeable Types



```
fn foo(obj1: Object1,
          obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

3.6 Ownership Tracking



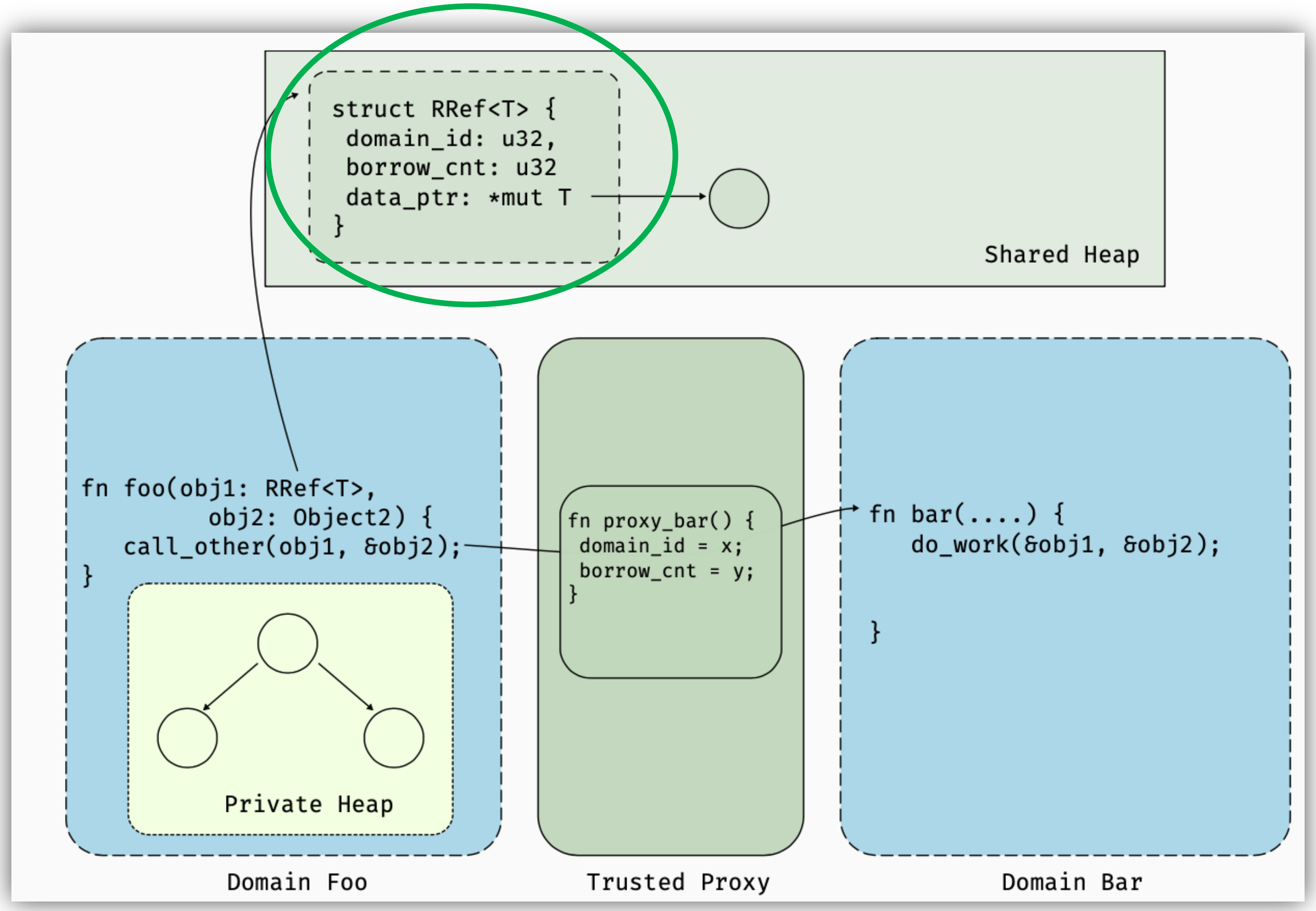
```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

```
fn foo(obj1: RRef<T>,
         obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.6 Ownership Tracking



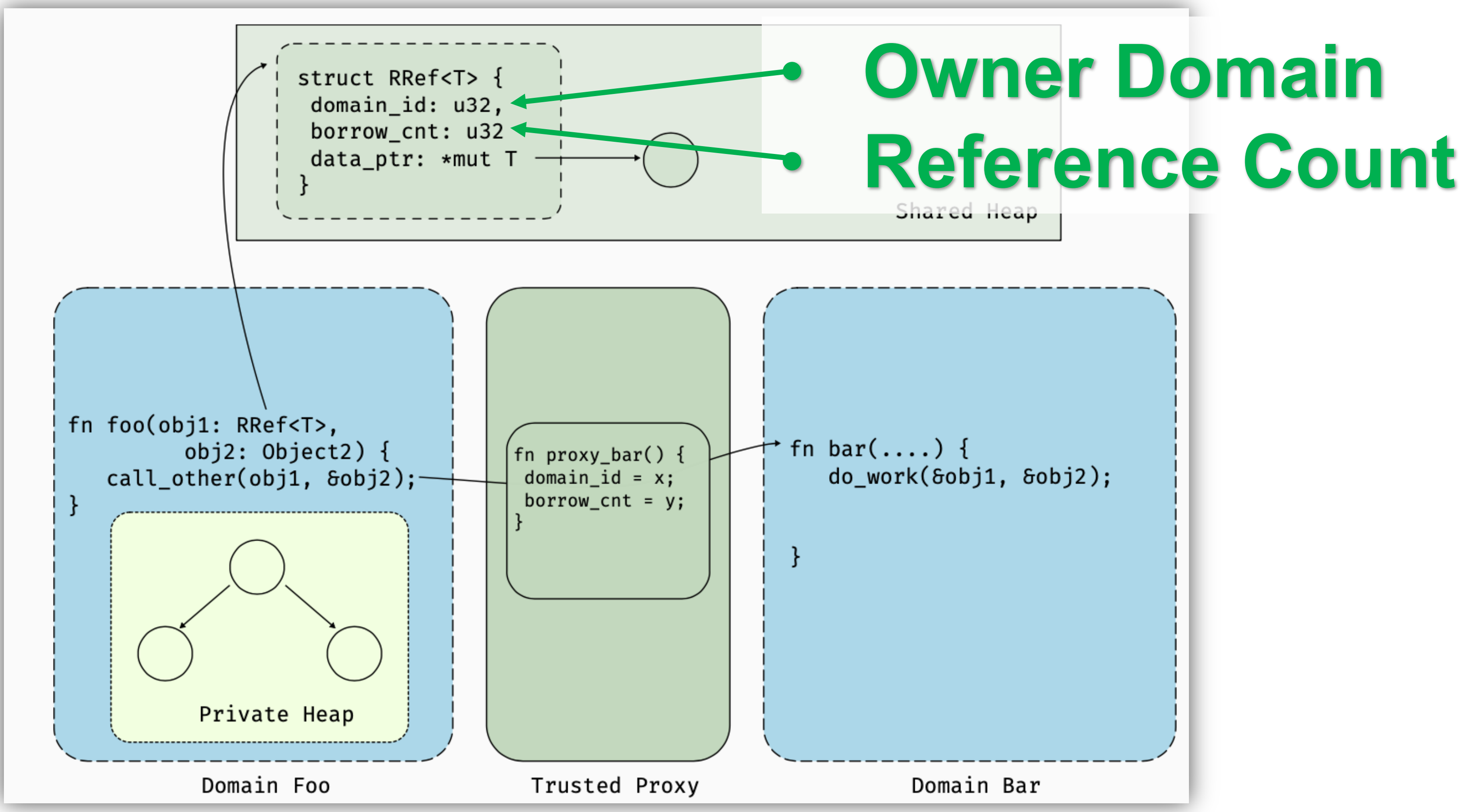
```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

```
fn foo(obj1: RRef<T>,
         obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.6 Ownership Tracking



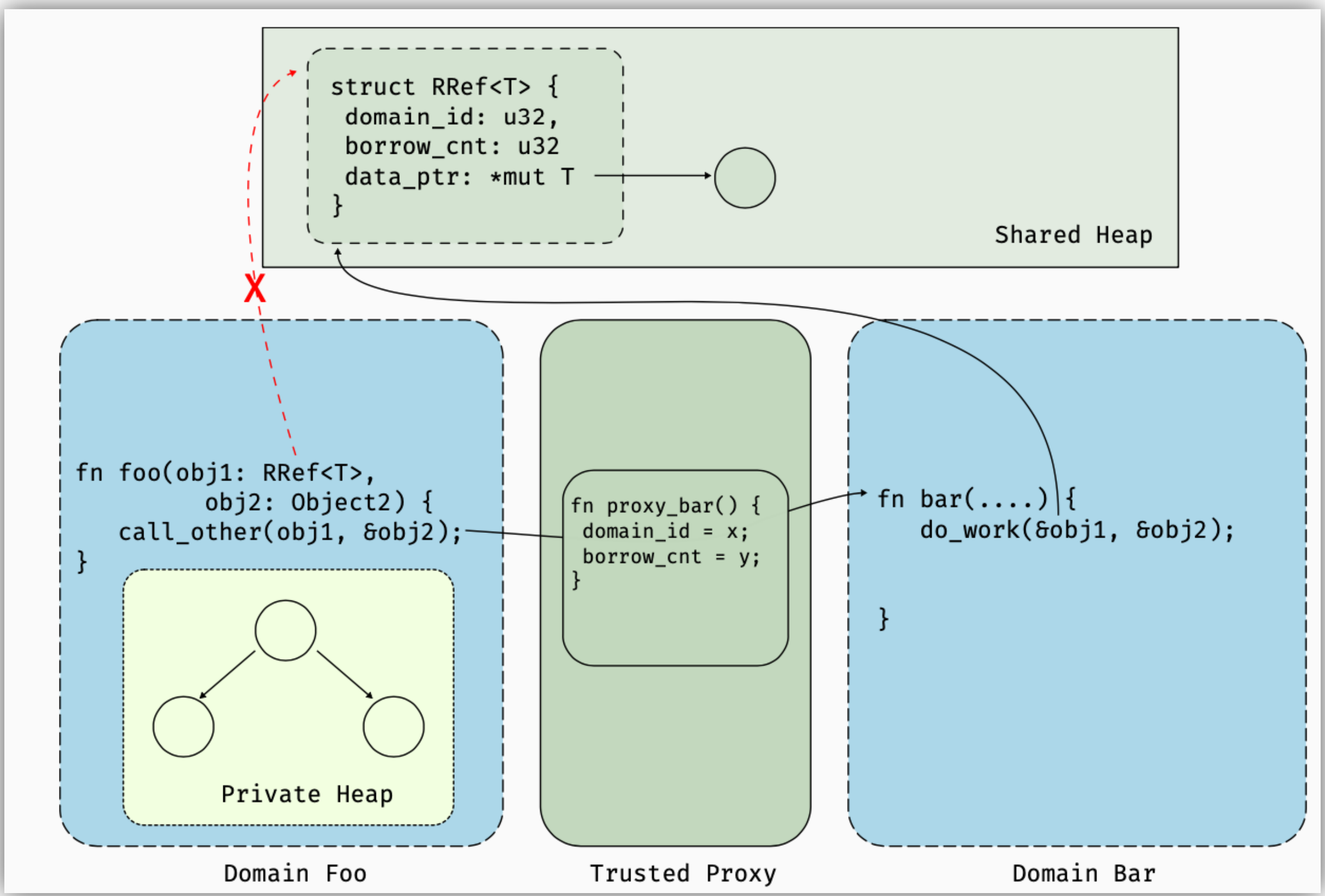
```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

```
fn foo(obj1: RRef<T>,
       obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.6 Ownership Tracking

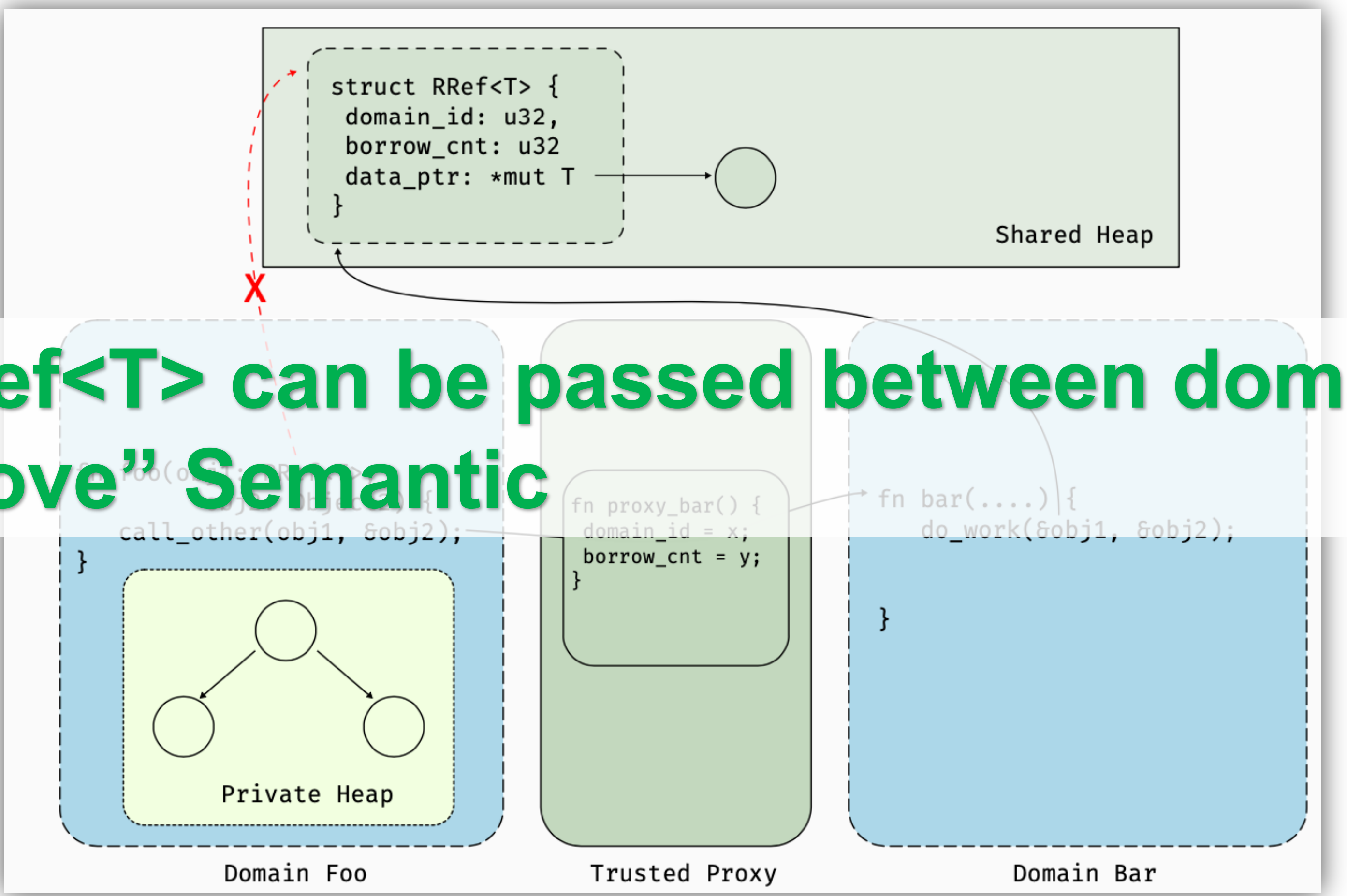

```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

```
fn foo(obj1: RRef<T>,
         obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.6 Ownership Tracking



```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

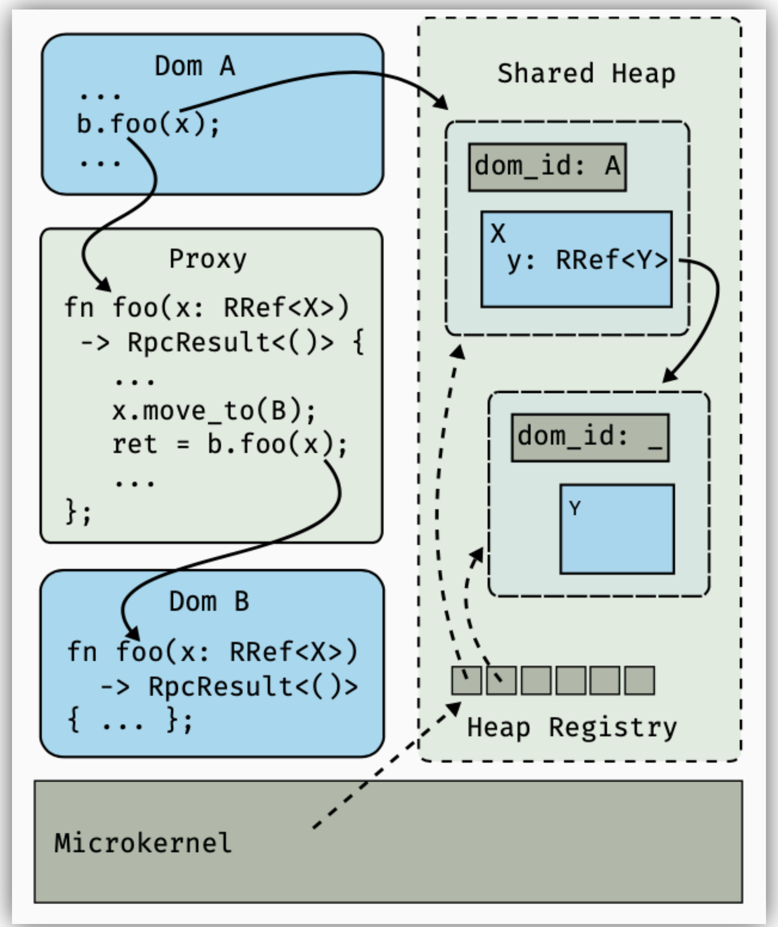
```
fn foo(o[illegible]: RR[illegible]
    [illegible] Object2) {
  call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

- RRef<T> can be passed between domains
- "Move" Semantic

3.6 Ownership Tracking

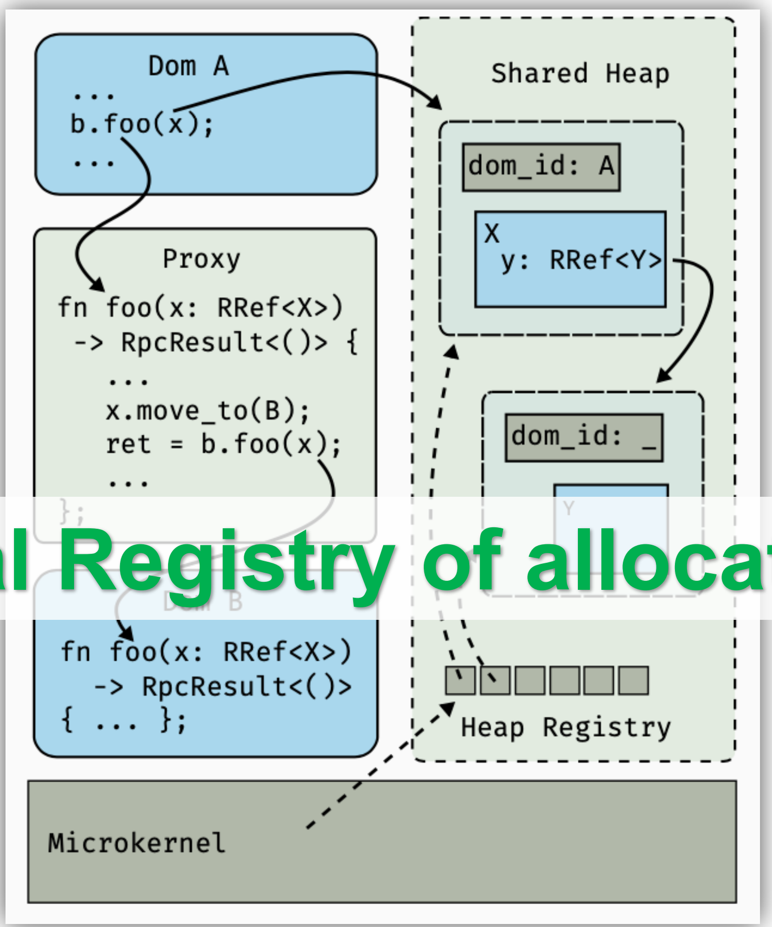


```
...
b.foo(x);
...
```

```
fn foo(x: RRef<X>)
 -> RpcResult<()> {
   ...
   x.move_to(B);
   ret = b.foo(x);
   ...
};
```

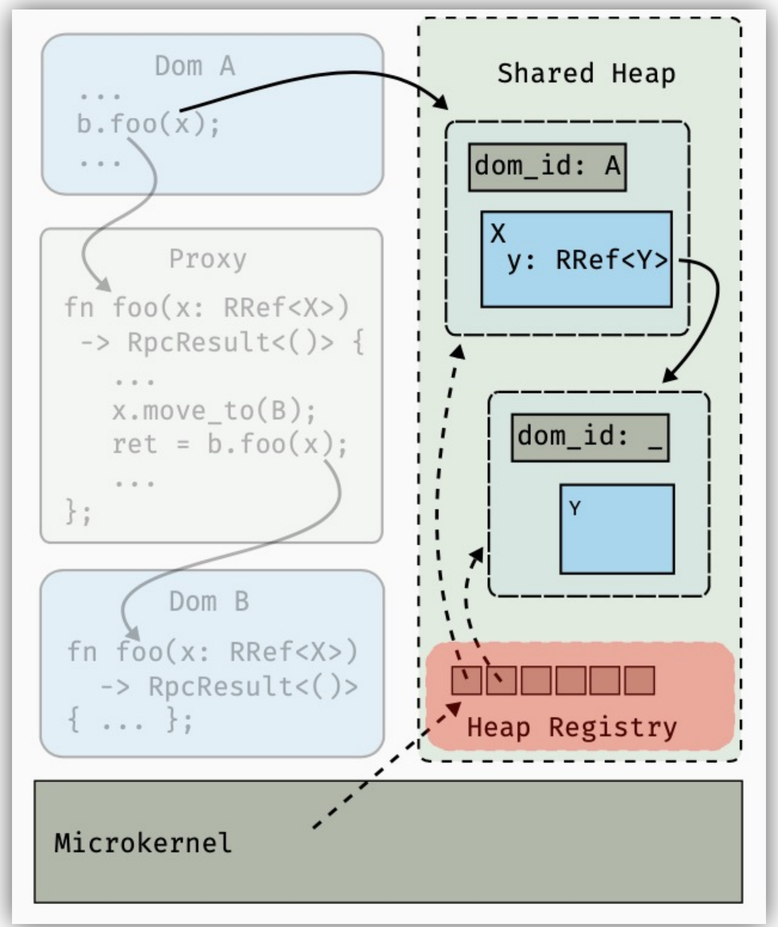
```
fn foo(x: RRef<X>)
  -> RpcResult<()>
{ ... };
```

3.6 Ownership Tracking



Global Registry of allocated objects

3.6 Ownership Tracking

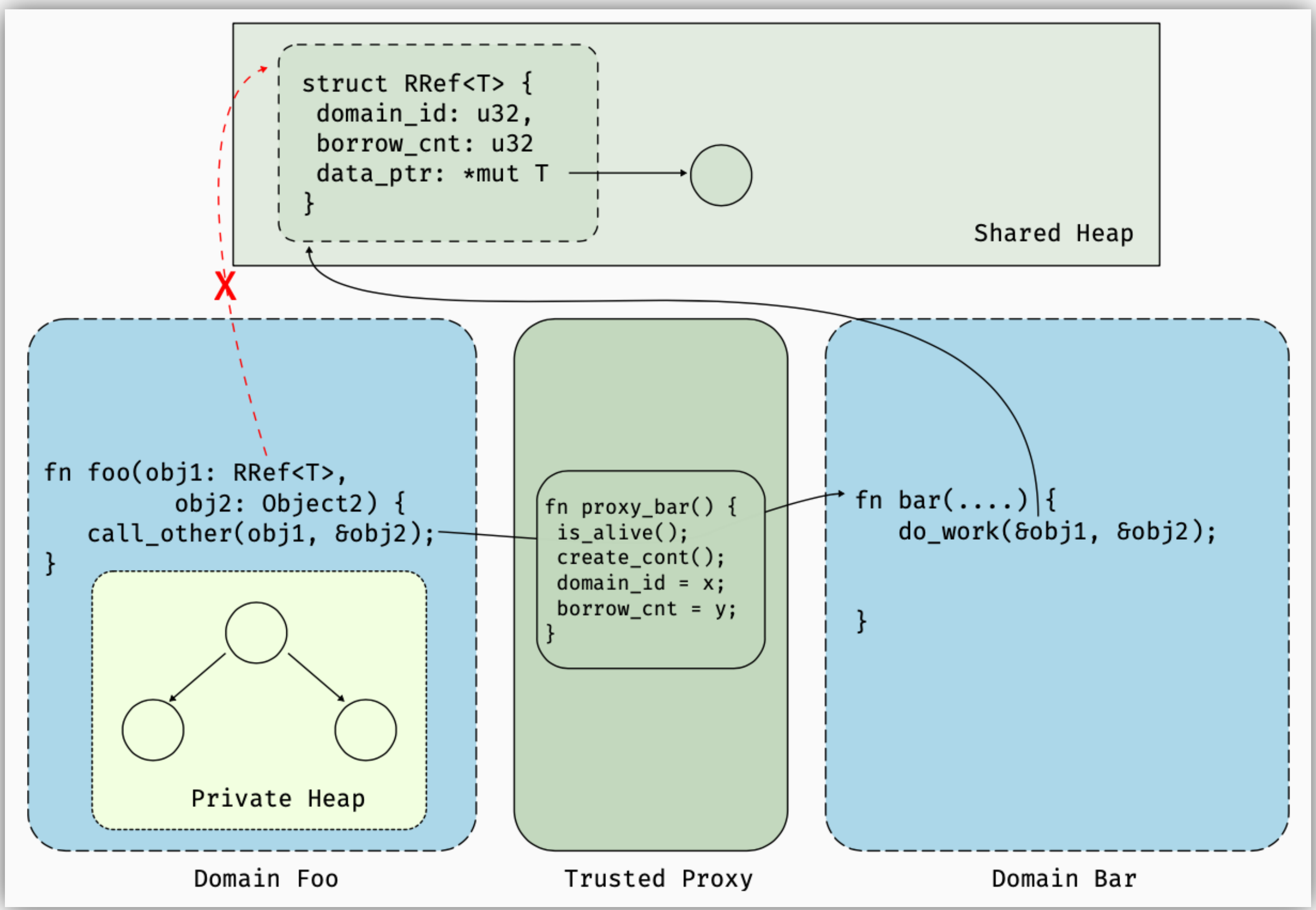


```
...
b.foo(x);
...
```

```
fn foo(x: RRef<X>)
 -> RpcResult<()> {
   ...
   x.move_to(B);
   ret = b.foo(x);
   ...
};
```

```
fn foo(x: RRef<X>)
  -> RpcResult<()>
{ ... };
```

3.7 Cross-domain Call Proxying



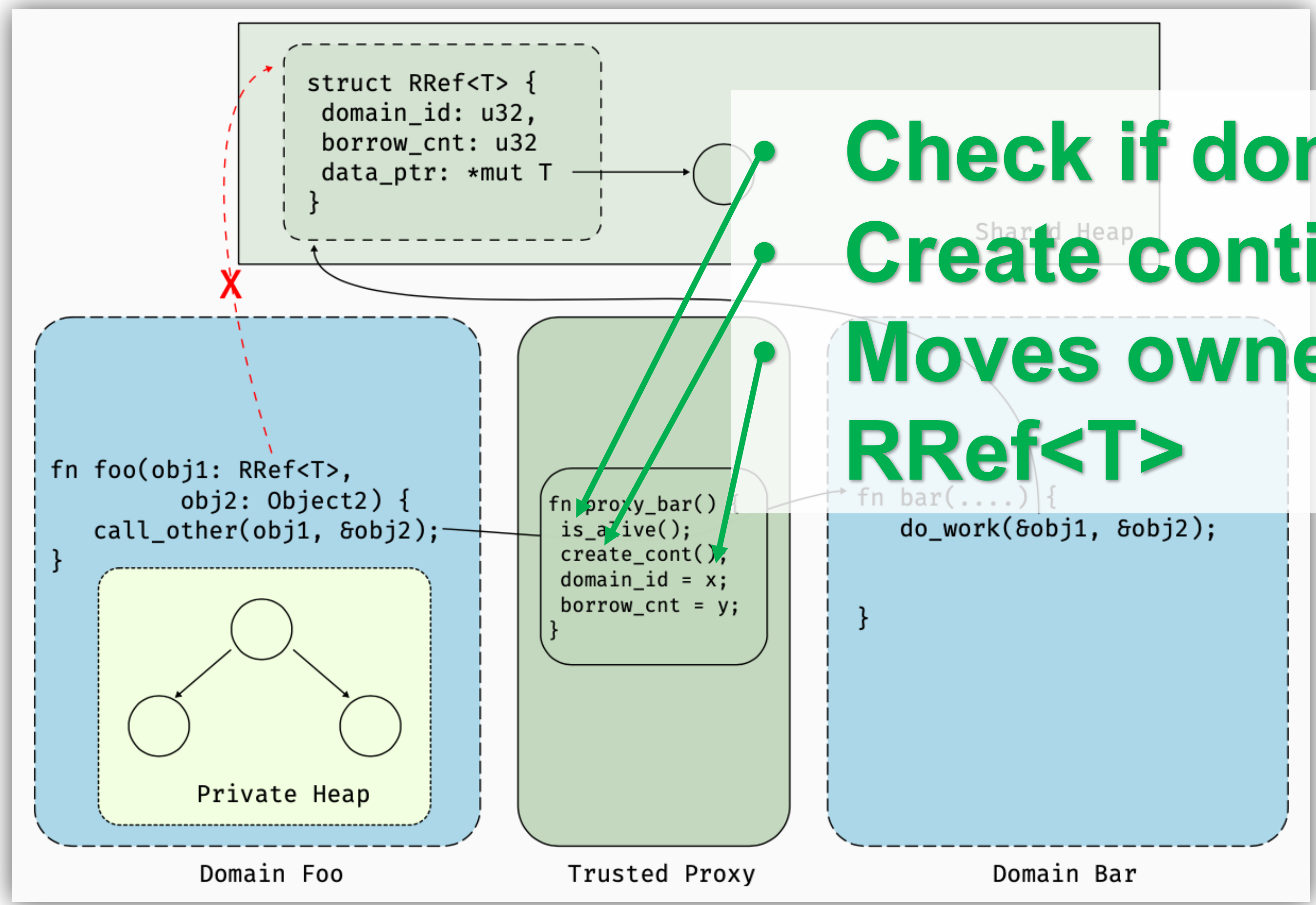
```
struct RRef<T> {
  domain_id: u32,
  borrow_cnt: u32
  data_ptr: *mut T
}
```

```
fn foo(obj1: RRef<T>,
         obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
 is_alive();
 create_cont();
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

3.7 Cross-domain Call Proxying



```
struct RRef<T> {
 domain_id: u32,
 borrow_cnt: u32
 data_ptr: *mut T
}
```

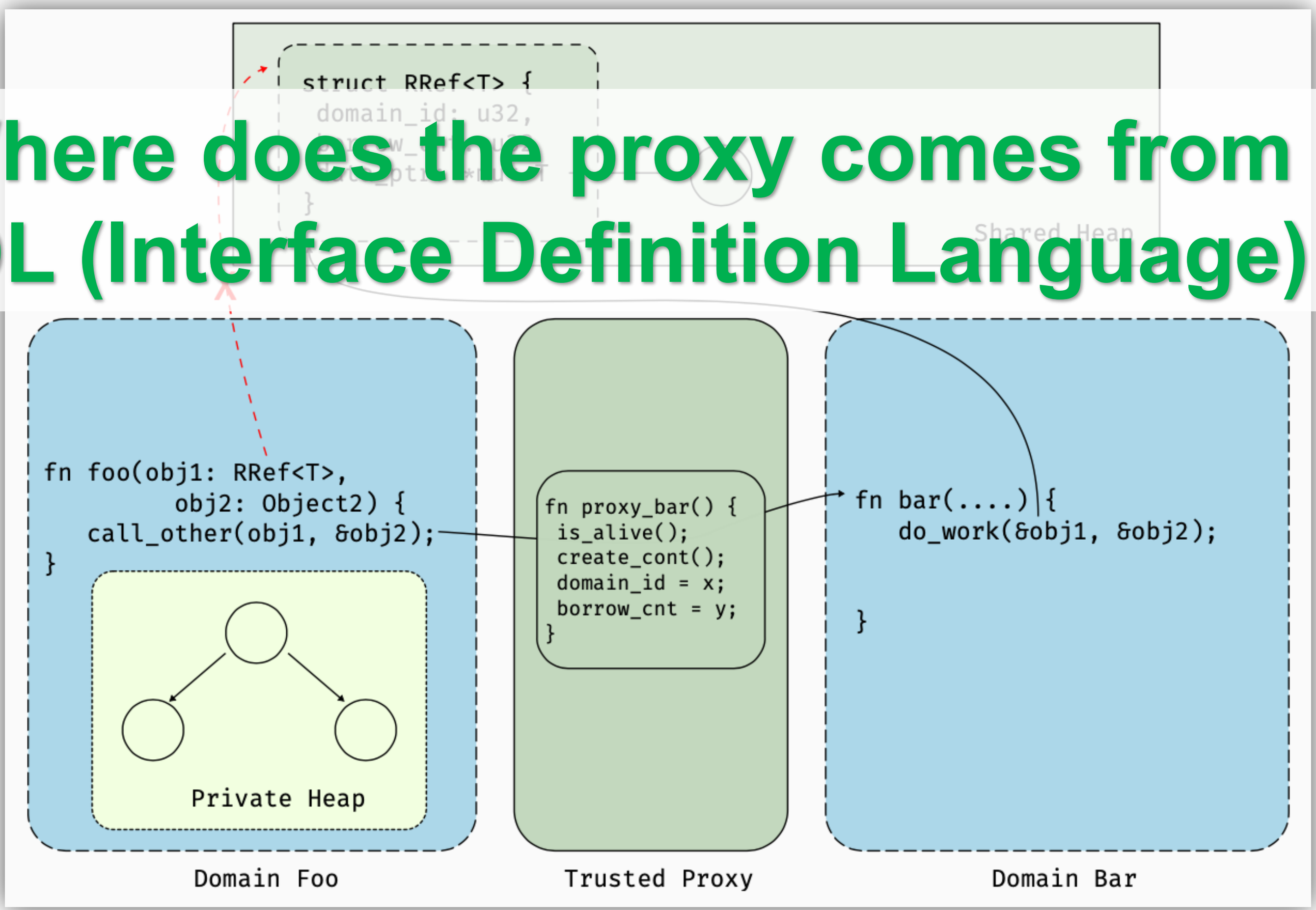
```
fn foo(obj1: RRef<T>,
         obj2: Object2) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar()
 is_alive();
 create_cont();
 domain_id = x;
 borrow_cnt = y;
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);

}
```

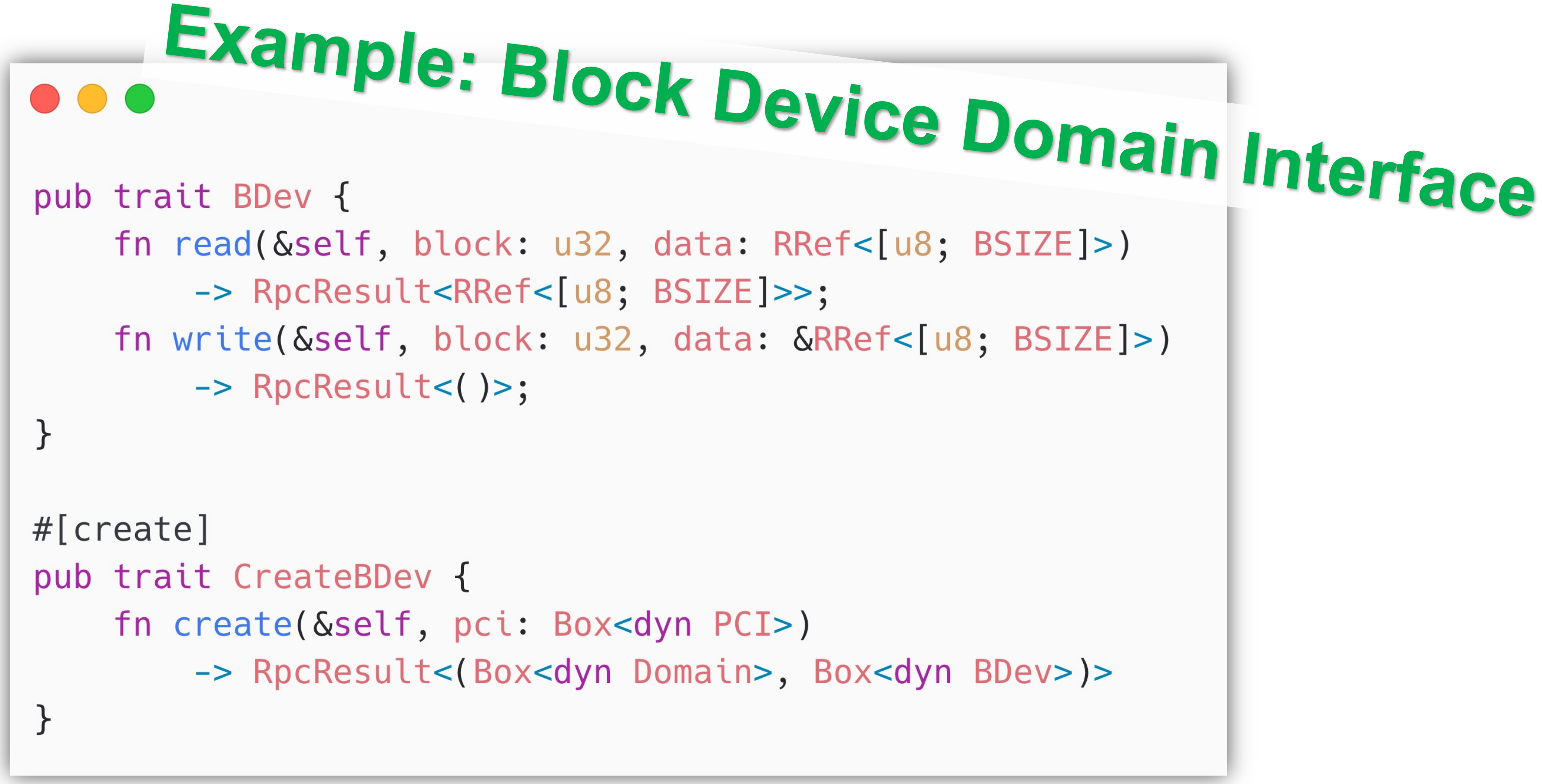
- Check if domain is alive
- Create continuation
- Moves ownership of RRef<T>

3.7 Cross-domain Call Proxying



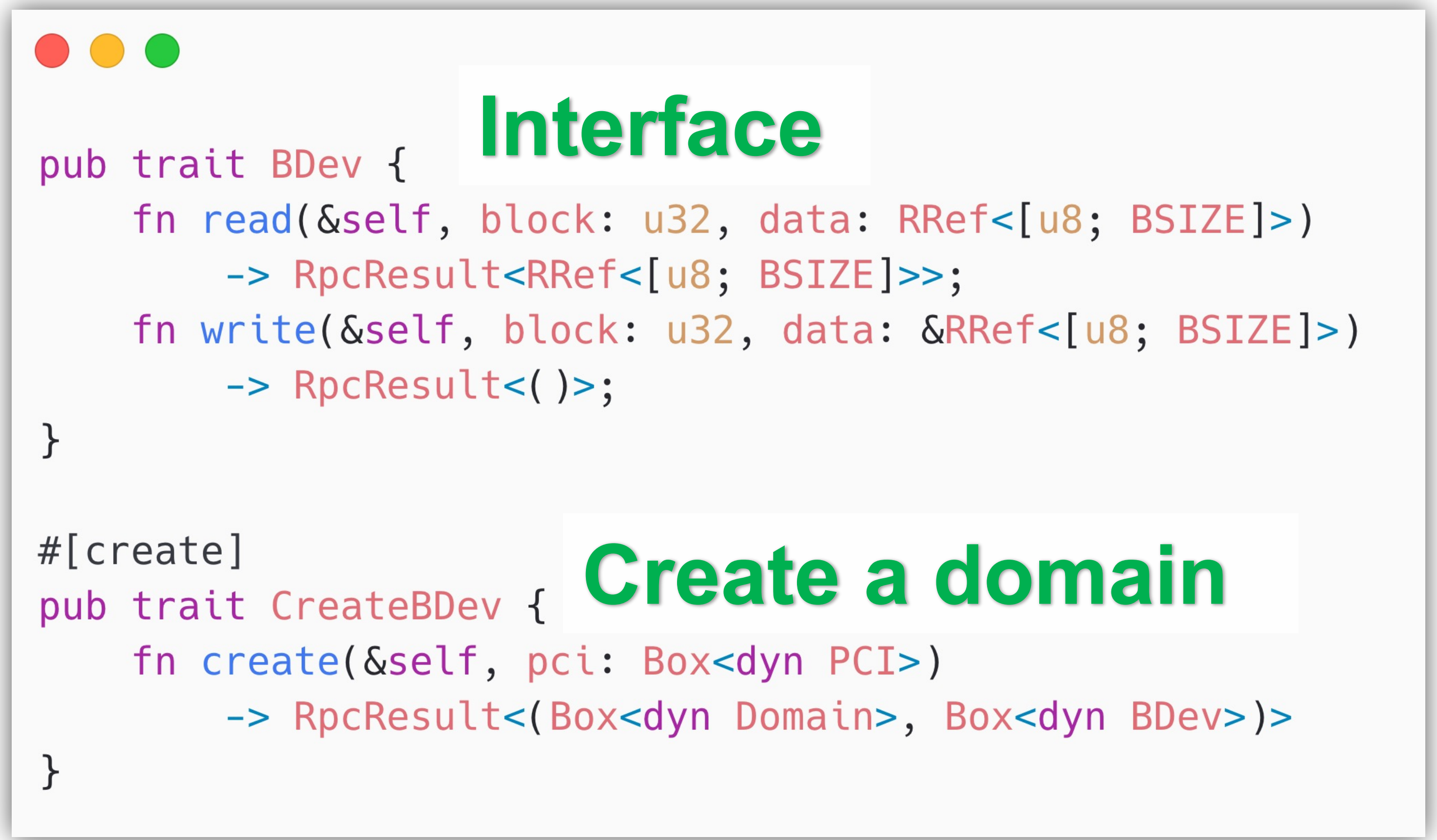
- **Where does the proxy comes from ?**
- **IDL (Interface Definition Language)**

3.8 IDL & IDL Compiler


Example: Block Device Domain Interface

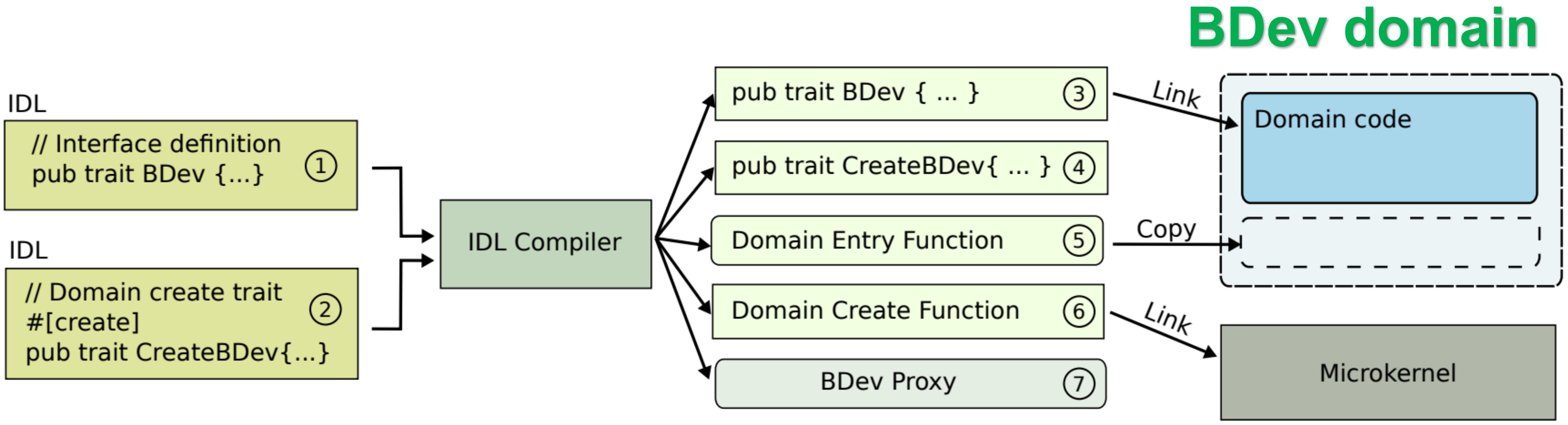
```
pub trait BDev {
    fn read(&self, block: u32, data: RRef<[u8; BSIZE]>)
        -> RpcResult<RRef<[u8; BSIZE]>>;
    fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>)
        -> RpcResult<()>;
}

#[create]
pub trait CreateBDev {
    fn create(&self, pci: Box<dyn PCI>)
        -> RpcResult<(Box<dyn Domain>, Box<dyn BDev>)>
}
```

3.8 IDL & IDL Compiler



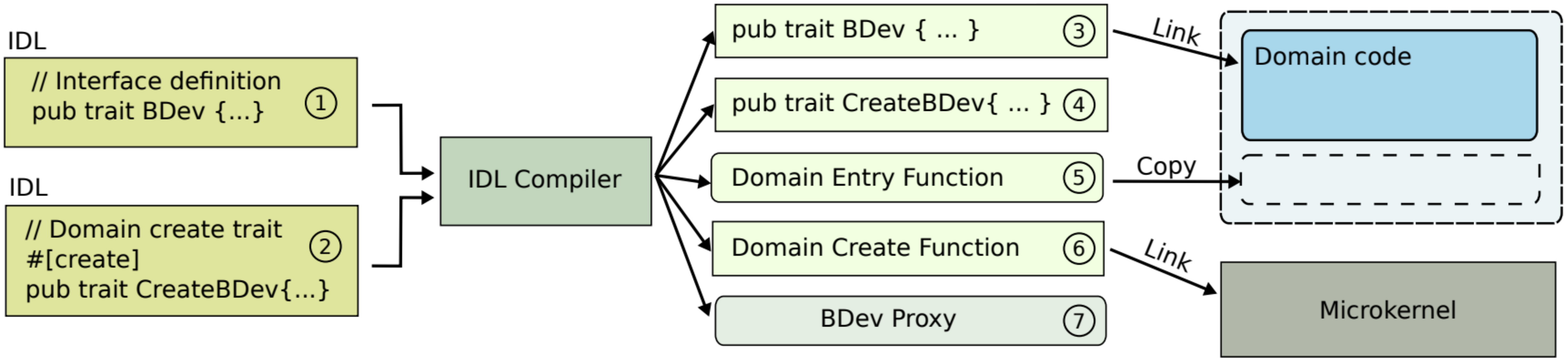
```
pub trait BDev {
    fn read(&self, block: u32, data: RRef<[u8; BSIZE]>)
        -> RpcResult<RRef<[u8; BSIZE]>>;
    fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>)
        -> RpcResult<()>;
}

#[create]
pub trait CreateBDev {
    fn create(&self, pci: Box<dyn PCI>)
        -> RpcResult<(Box<dyn Domain>, Box<dyn BDev>)>
}
```

3.8 IDL & IDL Compiler

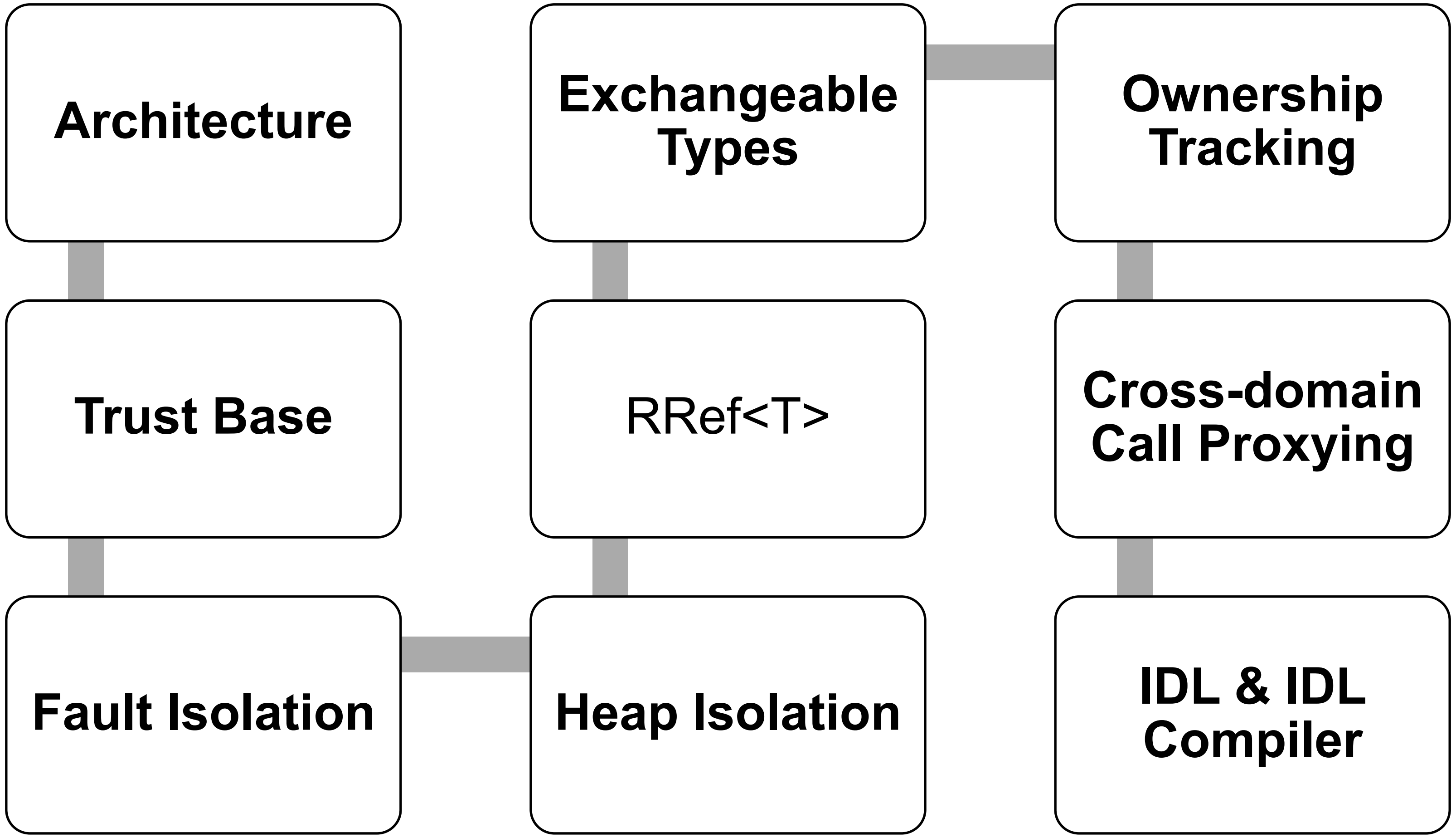


3.8 IDL & IDL Compiler

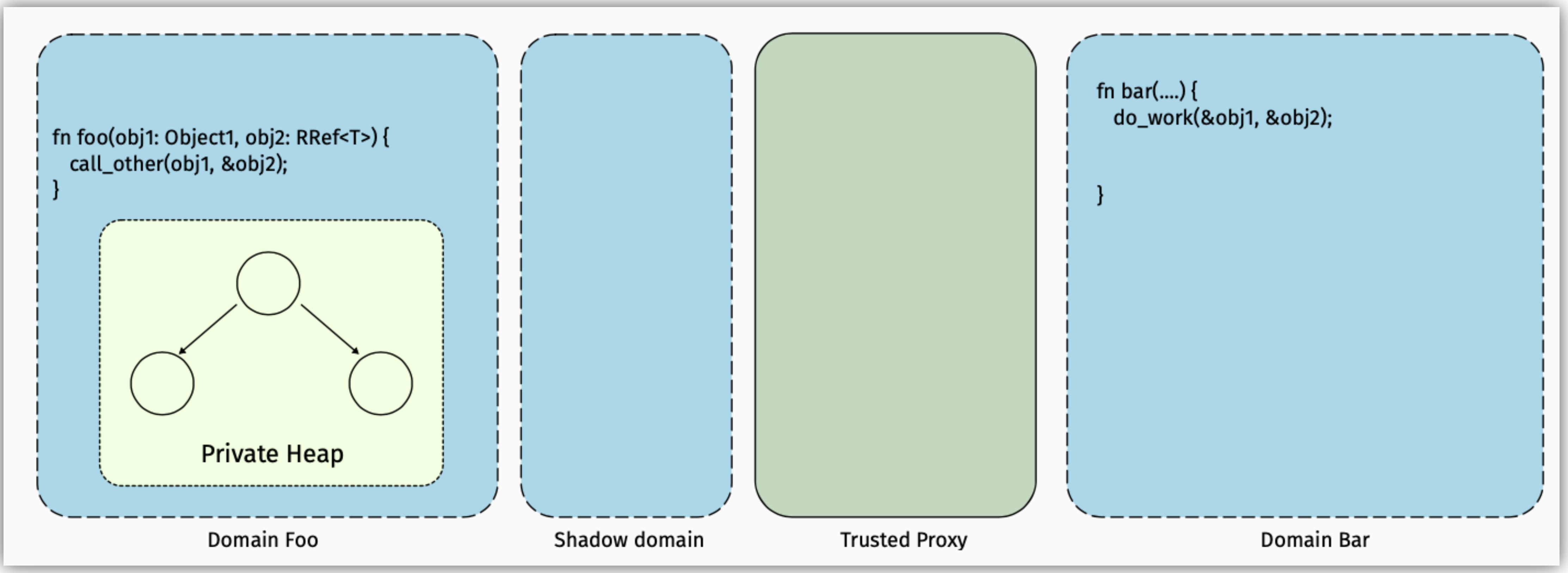
Static Analysis on AST
to extract interface definition



3.9 Summary of RedLeaf



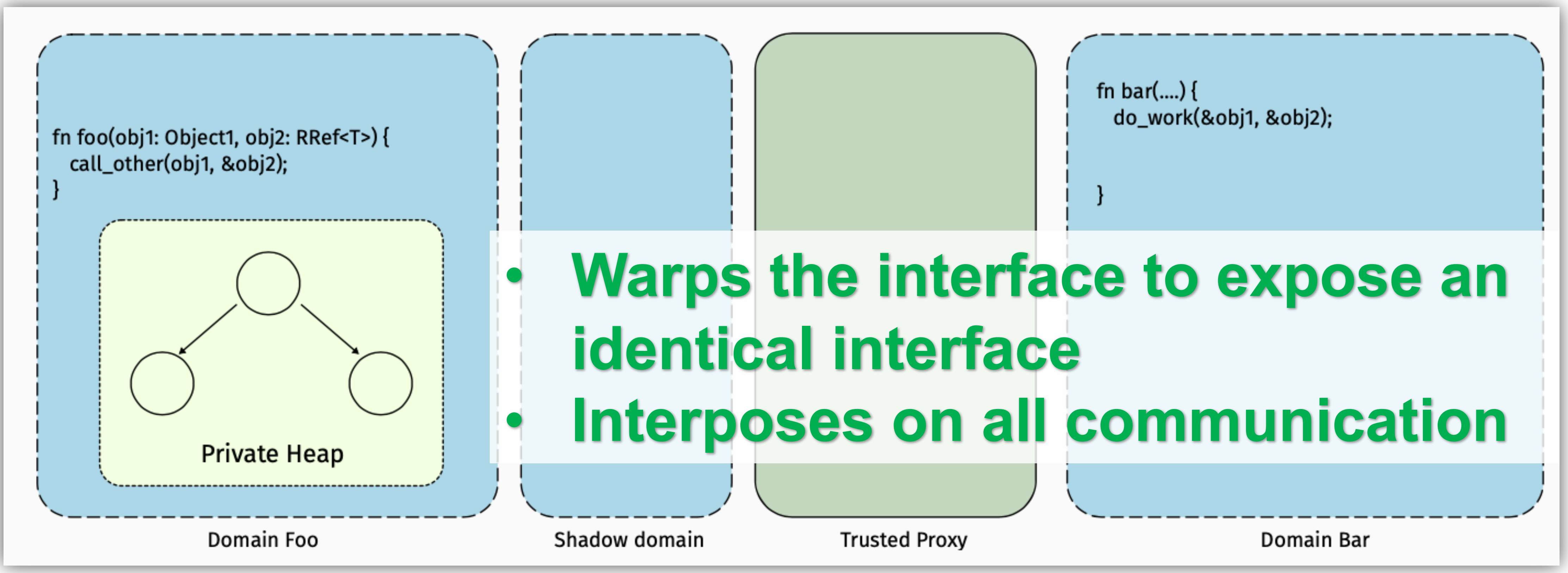
3.10 Device Driver Recovery



```
fn foo(obj1: Object1, obj2: RRef<T>) {
  call_other(obj1, &obj2);
}
```

```
fn bar(....) {
  do_work(&obj1, &obj2);

}
```

3.10 Device Driver Recovery

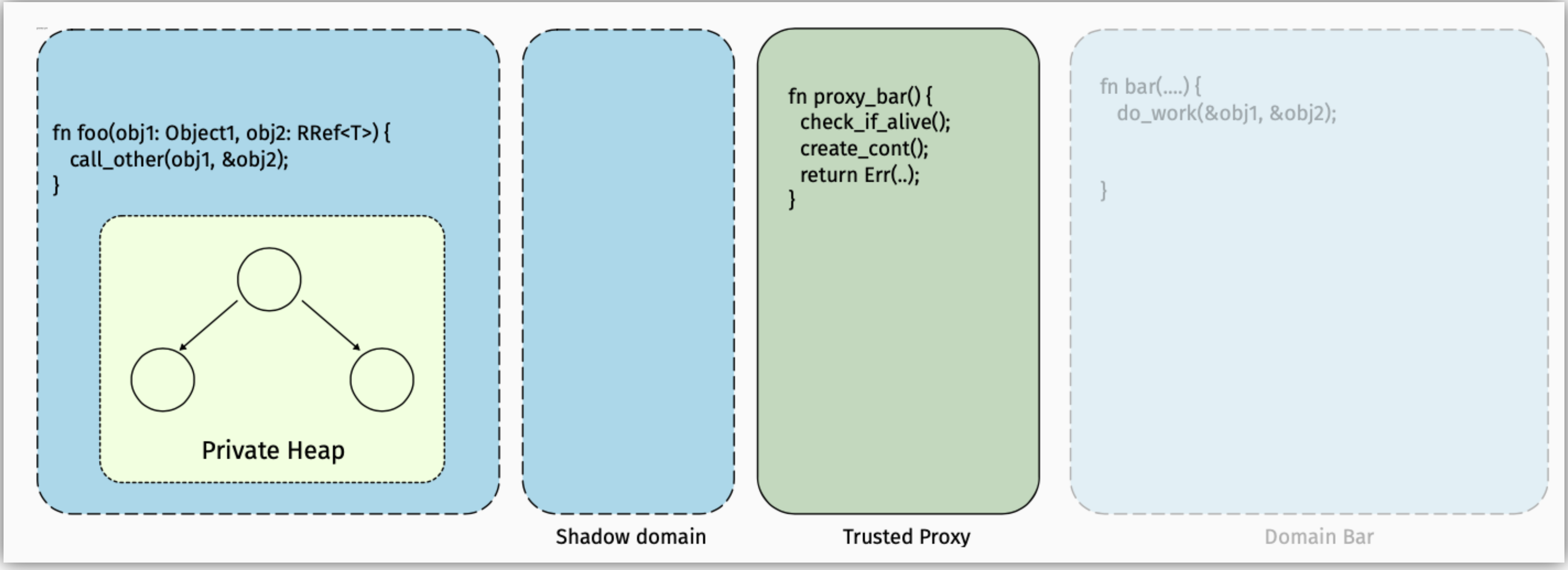


```
fn foo(obj1: Object1, obj2: RRef<T>) {
  call_other(obj1, &obj2);
}
```

```
fn bar(....) {
  do_work(&obj1, &obj2);

}
```

- Warps the interface to expose an identical interface
- Interposes on all communication

3.10 Device Driver Recovery

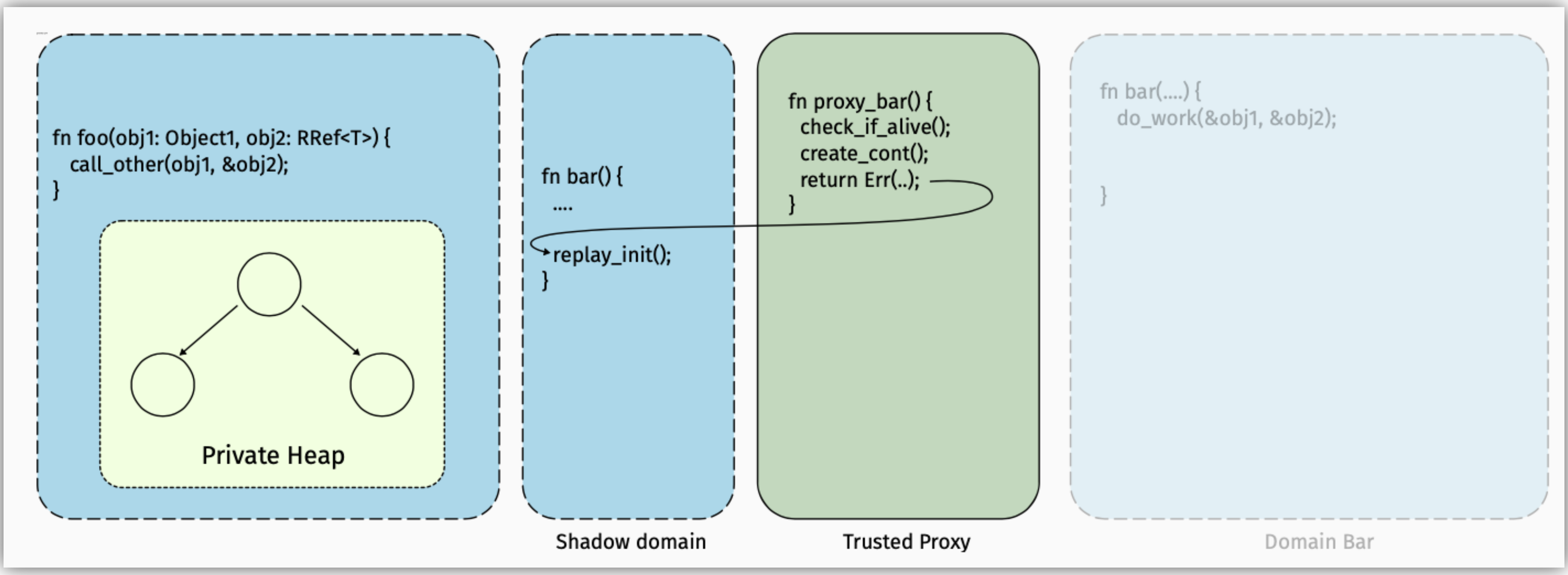


```
fn foo(obj1: Object1, obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

```
fn proxy_bar() {
  check_if_alive();
  create_cont();
  return Err(..);
}
```

```
fn bar(....) {
   do_work(&obj1, &obj2);


}
```

3.10 Device Driver Recovery



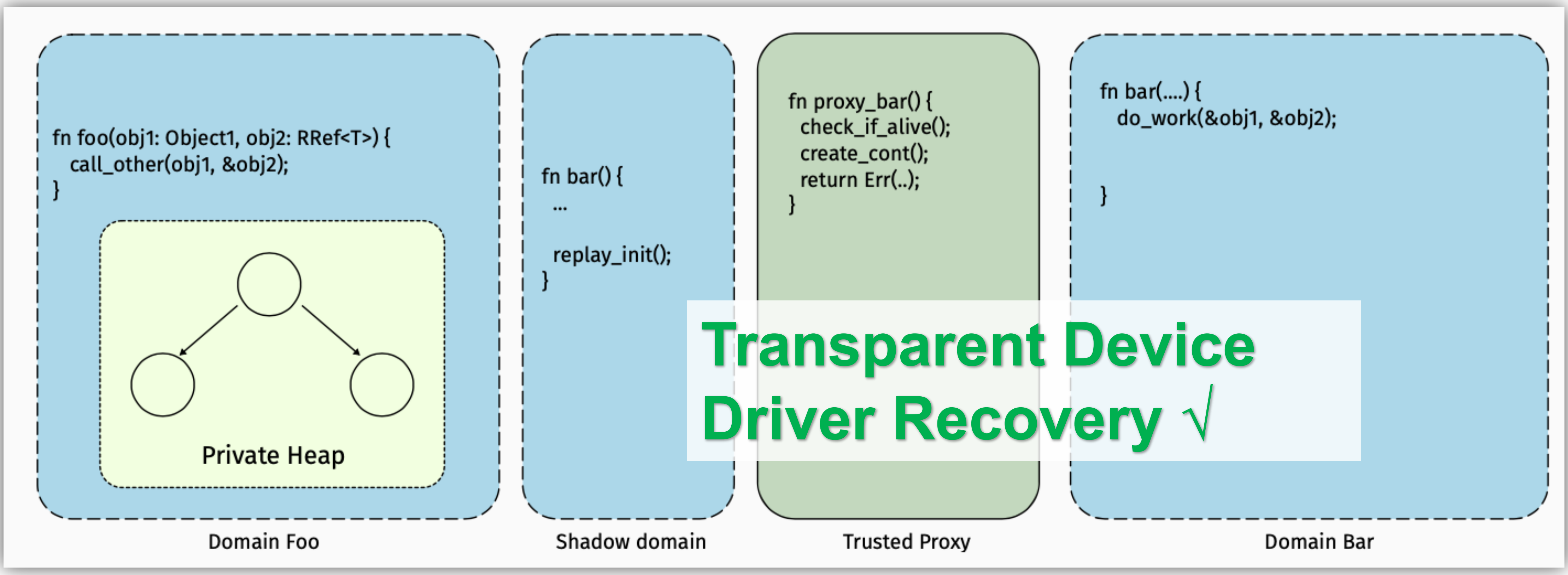
```
fn foo(obj1: Object1, obj2: RRef<T>) {
   call_other(obj1, &obj2);
}
```

```
fn bar() {
  ....

 replay_init();
}
```

```
fn proxy_bar() {
  check_if_alive();
  create_cont();
  return Err(..);
}
```

```
fn bar(....) {
  do_work(&obj1, &obj2);


}
```

3.10 Device Driver Recovery



```
fn foo(obj1: Object1, obj2: RRef<T>) {
  call_other(obj1, &obj2);
}
```

```
fn bar() {
  ...

  replay_init();
}
```

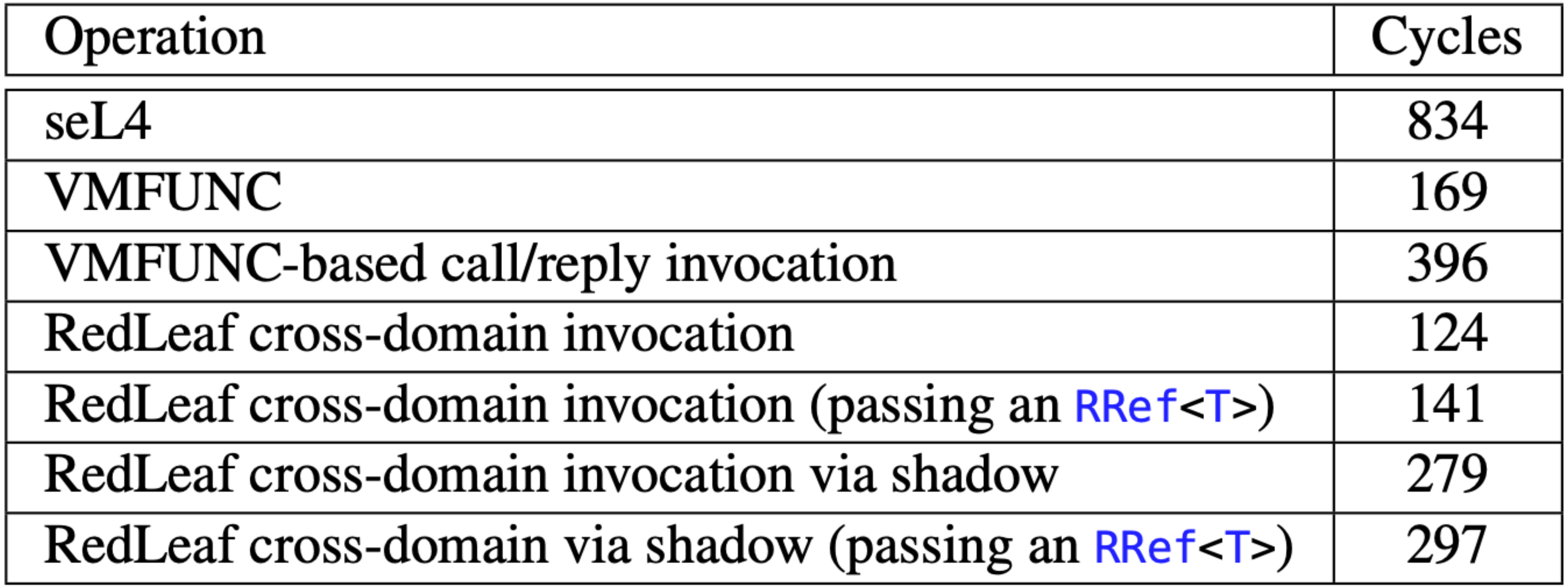
```
fn proxy_bar() {
  check_if_alive();
  create_cont();
  return Err(..);
}
```

```
fn bar(....) {
  do_work(&obj1, &obj2);

}
```

Transparent Device Driver Recovery √

4. Evaluation

4.1 Communication Cost

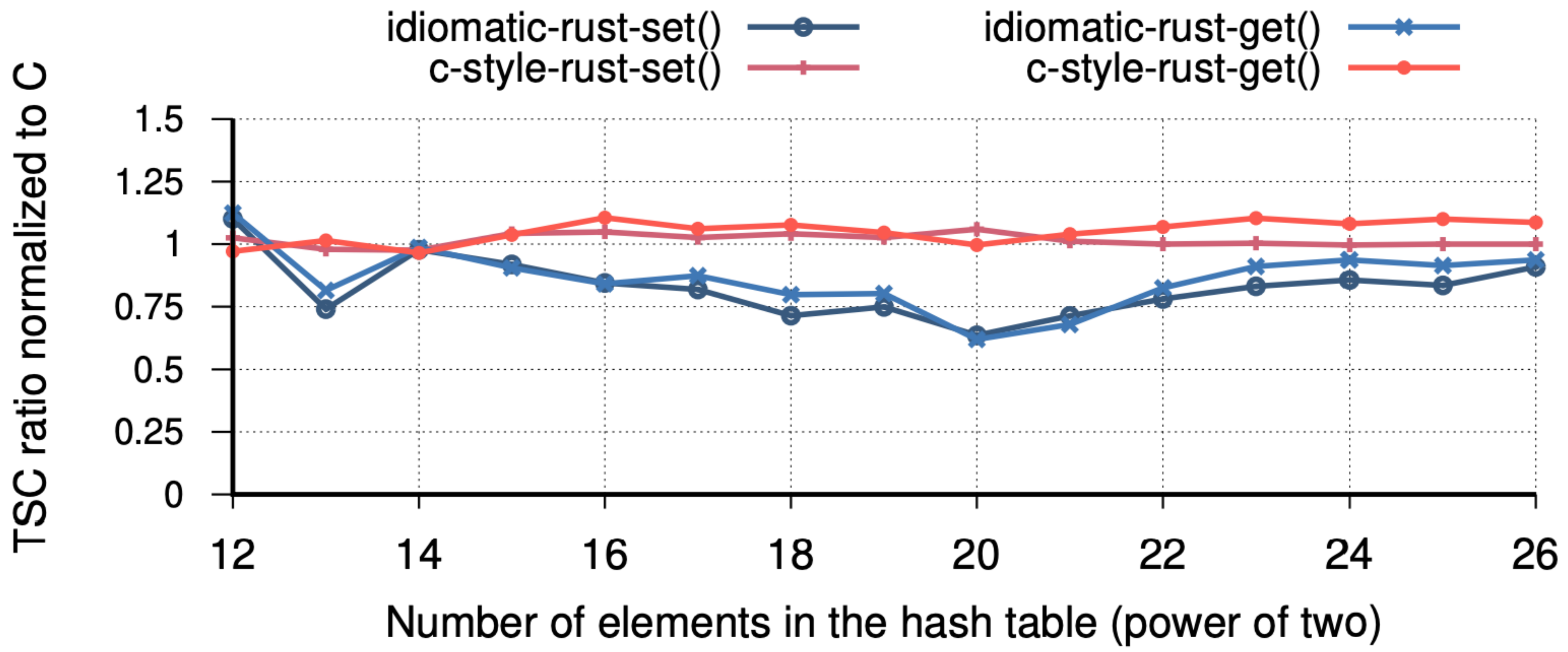


Operation	Cycles
seL4	834
VMFUNC	169
VMFUNC-based call/reply invocation	396
RedLeaf cross-domain invocation	124
RedLeaf cross-domain invocation (passing an `RRef<T>`)	141
RedLeaf cross-domain invocation via shadow	279
RedLeaf cross-domain via shadow (passing an `RRef<T>`)	297

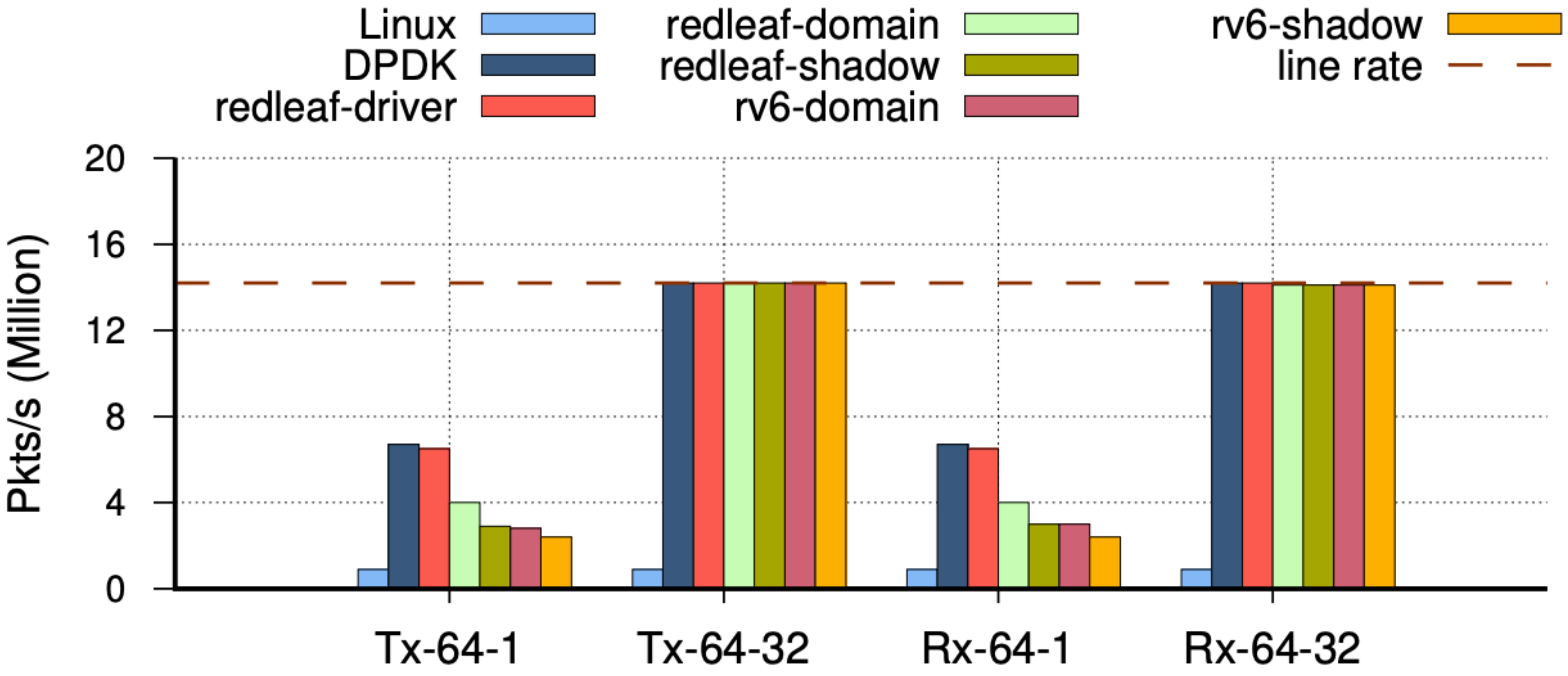
4.2 Language Overhead

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions **usize, usize**
- Idiomatic Rust - **Option<(usize, usize)>**
- Vary the size (2^{12} to 2^{26} at 75% full)

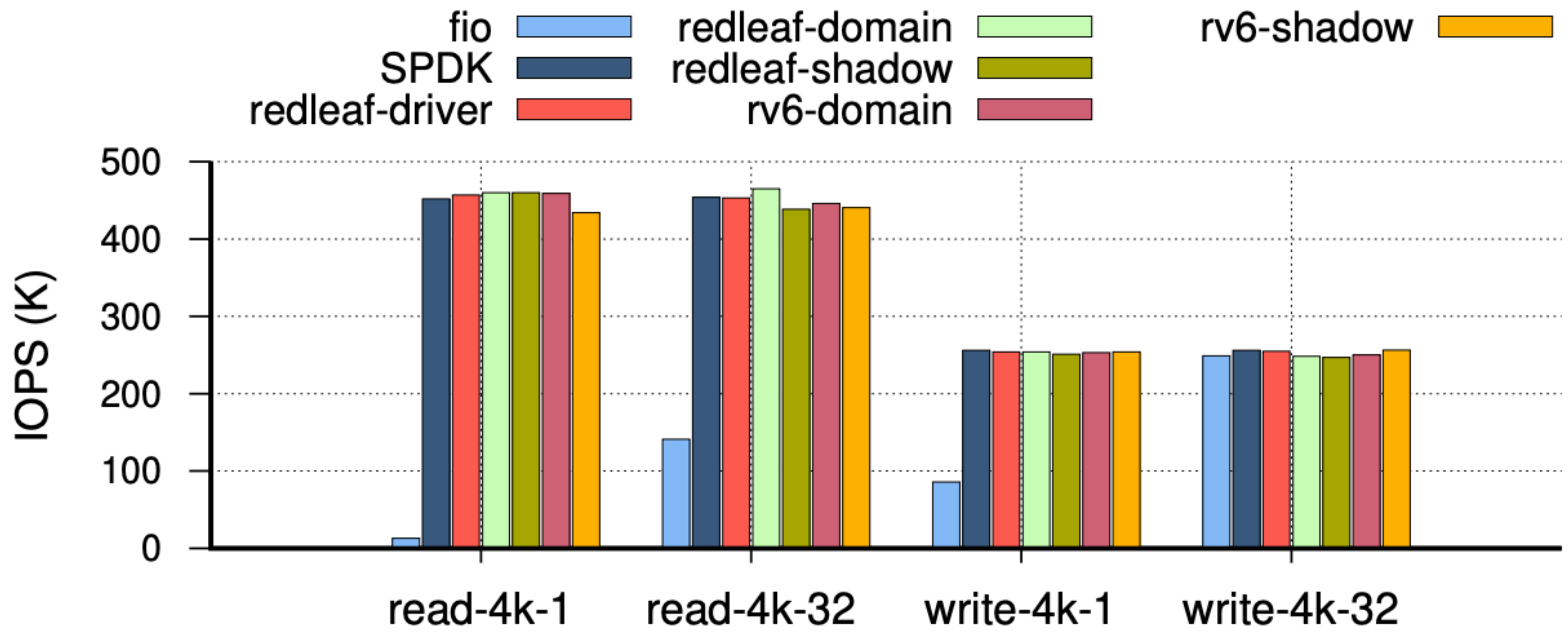
4.2 Language Overhead



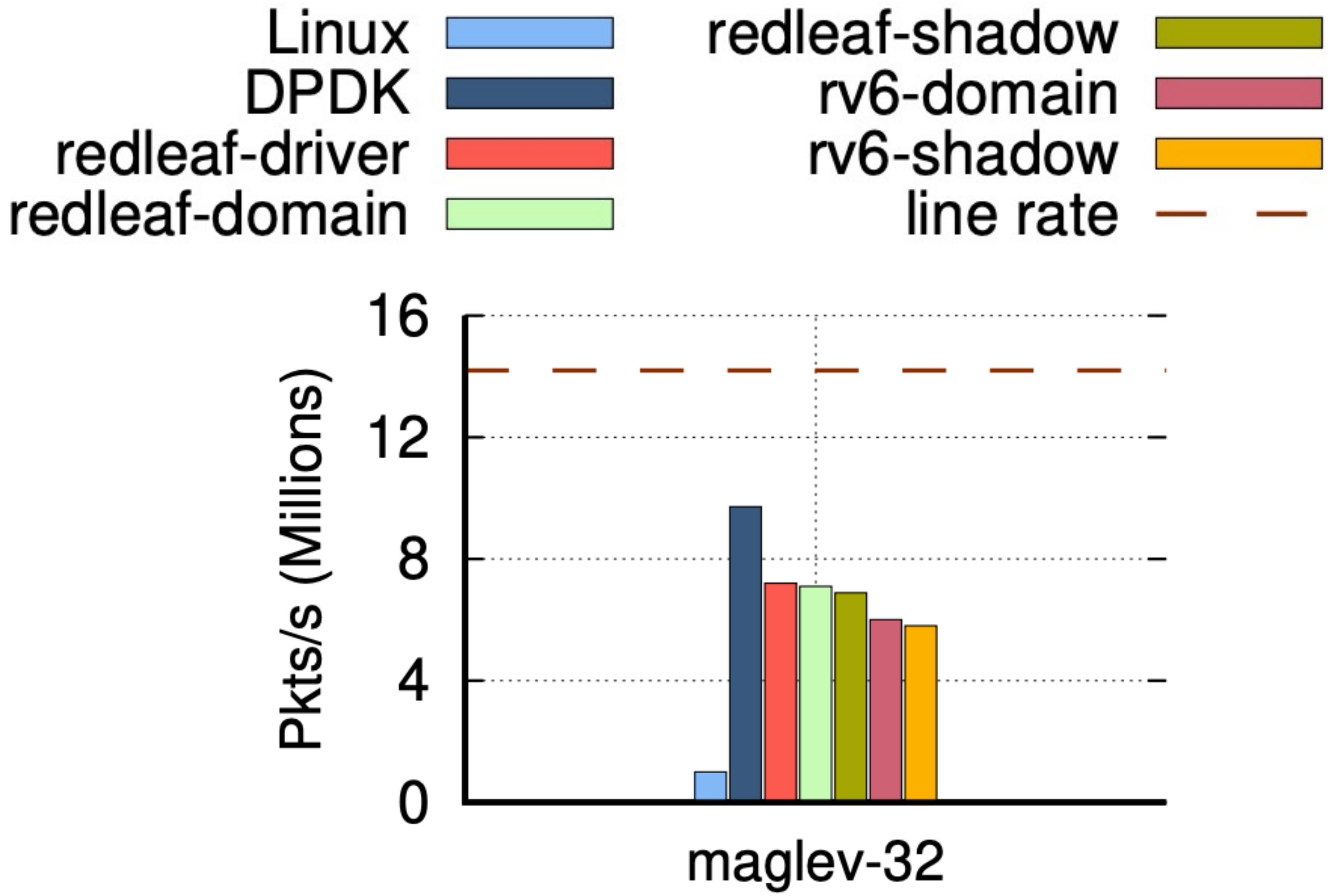
4.3 Device Drivers: ixgbe



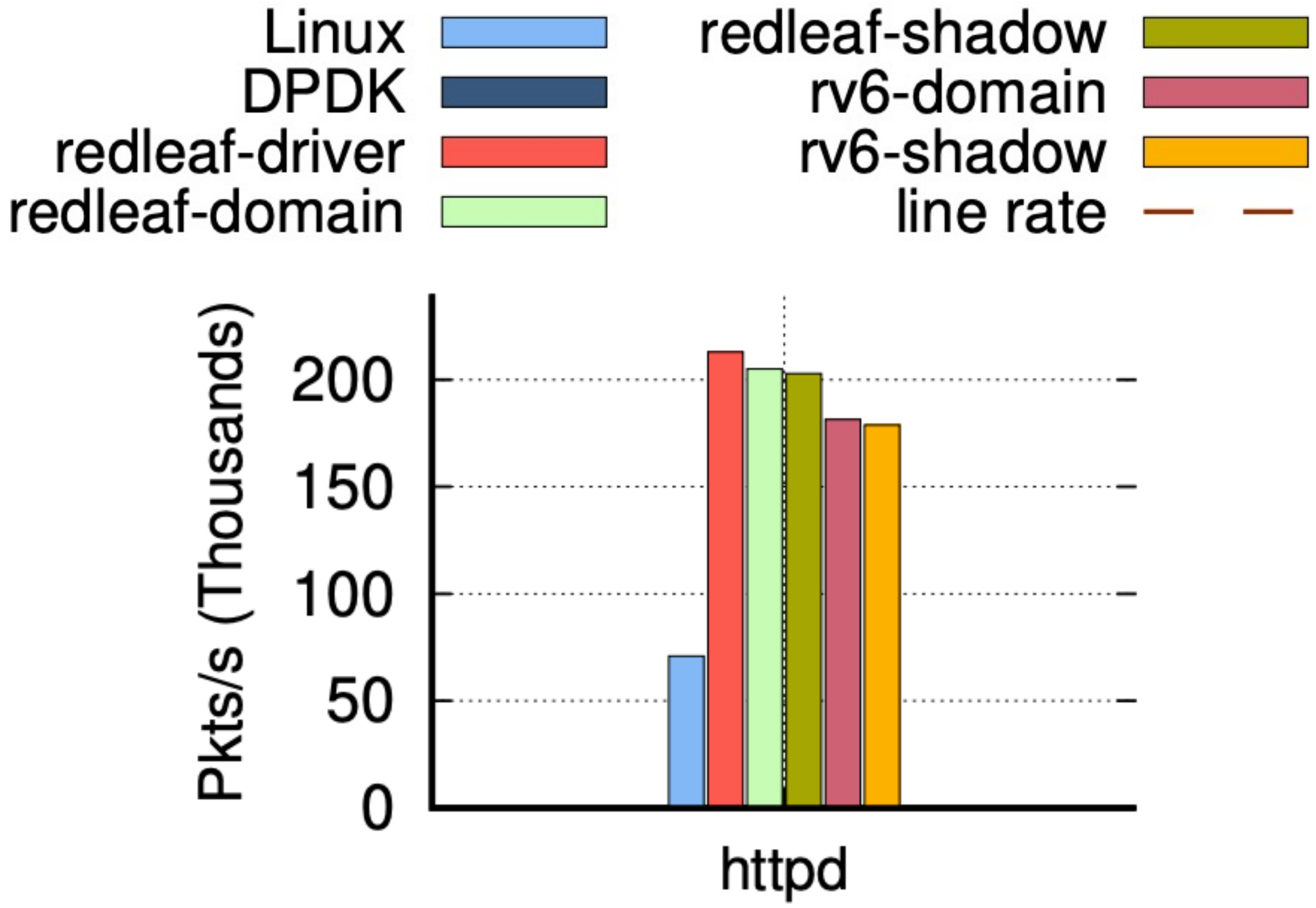
4.3 Device Drivers: NVMe



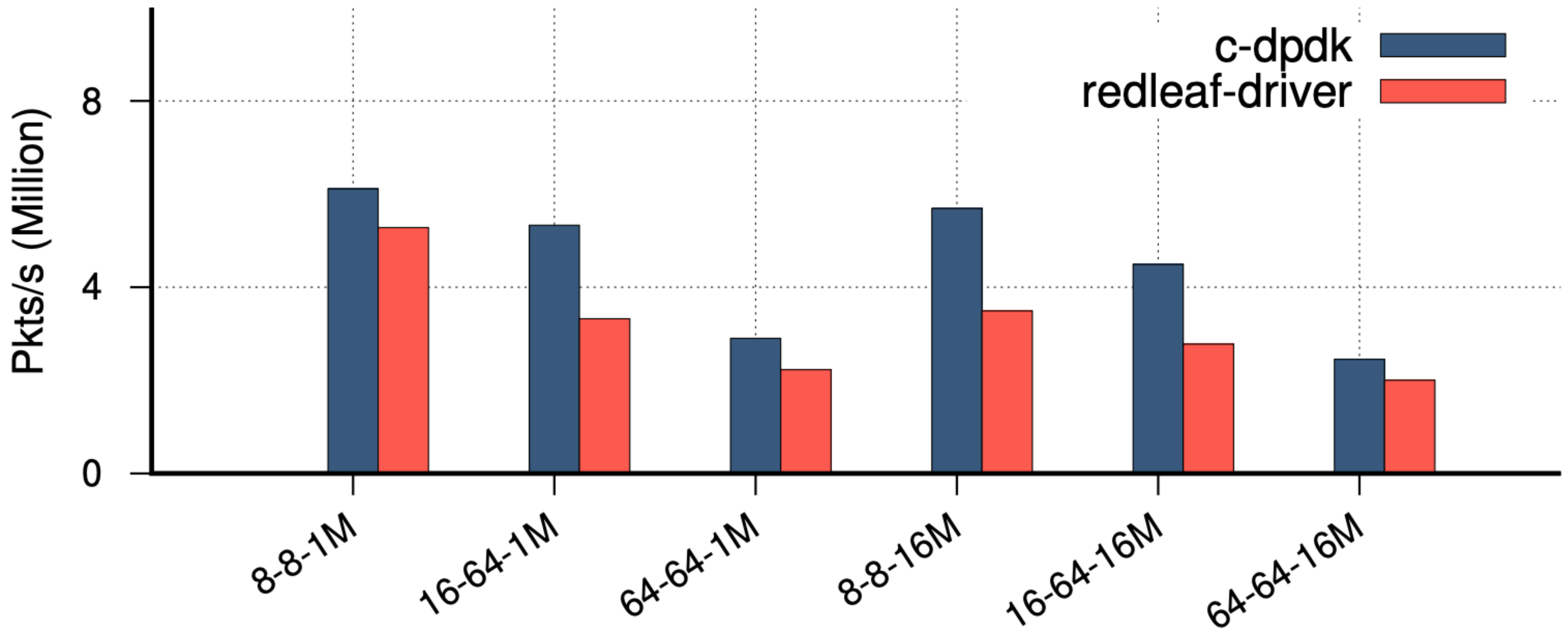
4.4 Application Test: Maglev



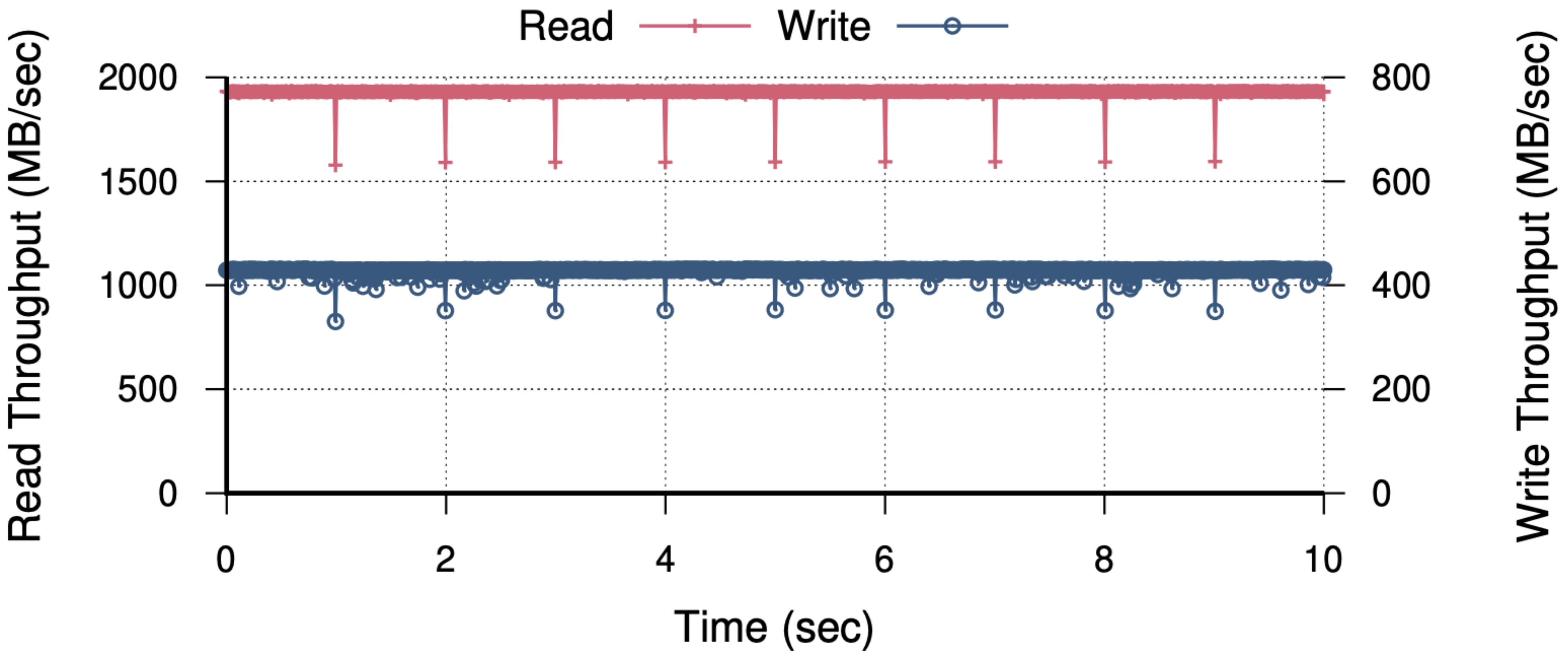
4.4 Application Test: Httpd



4.4 Application Test: KV-Store



4.5 Device Driver Recovery



5. Conclusion & Insight

5.1 Conclusion

- Heap isolation, exchangeable types, ownership tracking, interface validation, cross-domain call proxying
- Provides a collection of mechanisms for enabling isolation
- A step forward in enabling future system architectures
  - Secure kernel extensions
  - fine-grained access control
  - transparent recovery