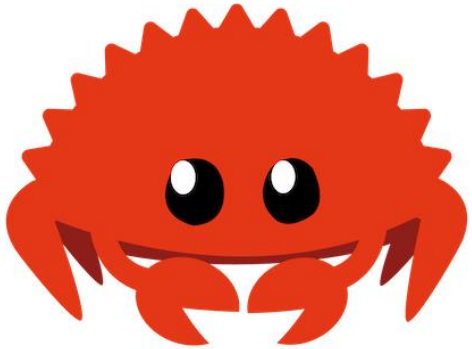
Securing a Multiprocessor KVM Hypervisor with Rust (SoCC '24)

Yu-Hsun Chiang, Wei-Lin Chang, Shih-Wei Li, Jan-Ting Tu @ National Taiwan University

paper slides created by Yulong Han @ Peking University

Background: Rust programming language



- low-level programming with **no GC**(Garbage Collection)
- emphasizing *performance*, *type-safety* and *concurrency*
- **ownership** model:
  - one value can only have one owner
  - reference can borrow the value as mutable/immutable
  - immutable references can have many, but only one mutable reference should exist
- variable **lifetime** check and notation
- fearless **concurrency** in [std] environment


```rust
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = s1;
    // println!("{}", s1); // ERROR: s1 is no longer valid
```

variable is immutable by default


```rust
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = s1;
    // println!("{}", s1); // ERROR: s1 is no longer valid

    let s3 = &s2; // Borrow s2
    println!("Immutable reference: {}", s3);
```

one variable can have multiple immutable references


```rust
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = s1;
    // println!("{}", s1); // ERROR: s1 is no longer valid

    let s3 = &s2; // Borrow s2
    println!("Immutable reference: {}", s3);

    let mut s4 = String::from("Mutable Rust");
    let s5 = &mut s4;
    s5.push_str(" is cool!");
    println!("Mutable reference: {}", s5);
}
```

only one mutable referece should exist (even s4 is invalid after s5)


```rust
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = s1;
    // println!("{}", s1); // ERROR: s1 is no longer valid

    let s3 = &s2; // Borrow s2
    println!("Immutable reference: {}", s3);

    let mut s4 = String::from("Mutable Rust");
    let s5 = &mut s4;
    s5.push_str(" is cool!");
    println!("Mutable reference: {}", s5);

    let s6 = takes_and_gives_back(s2);
    println!("After function: {}", s6);
}

fn takes_and_gives_back(s: String) -> String {
    println!("Taking ownership of: {}", s);
    s
}
```

functions take and return ownership

Standard Output

```
Immutable reference: Hello, Rust!
Mutable reference: Mutable Rust is cool!
Taking ownership of: Hello, Rust!
After function: Hello, Rust!
```

one more thing: Rust traits

```
1. trait Greet {
2.     fn greet(&self) -> String;
3. }
4.
5. struct Person {
6.     name: String,
7. }
```

one more thing: Rust traits

```
1. trait Greet {
2.     fn greet(&self) -> String;
3. }
4.
5. struct Person {
6.     name: String,
7. }
8.
9. impl Greet for Person {
10.     fn greet(&self) -> String {
11.         format!("Hello, {}!", self.name)
12.     }
13. }
14.
15. fn main() {
16.     let person = Person {
17.         name: String::from("Alice"),
18.     };
19.     println!("{}", person.greet());
20. }
```

Alice

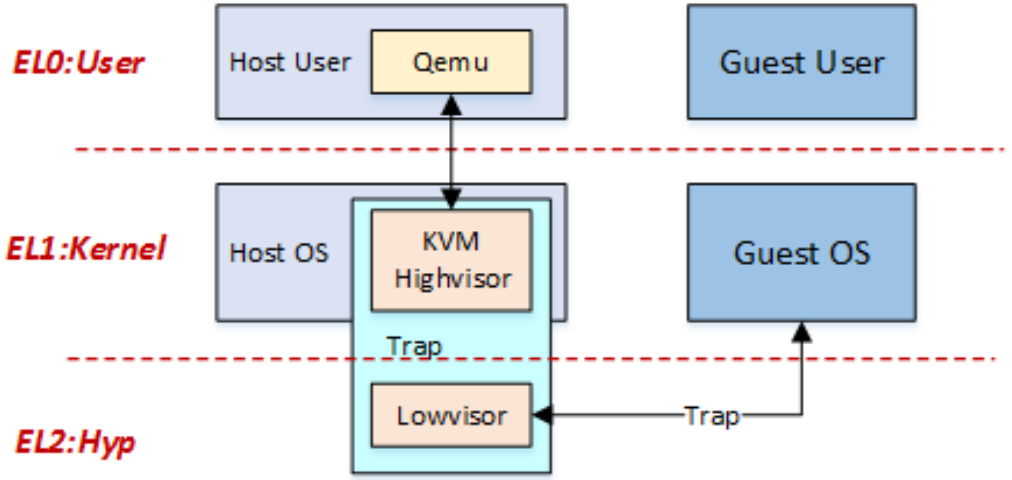
```
Hello, Alice!
```

Background: System Virtualization and Hypervisors

- System virtualization: provides multiple *virtual* hardware on a single hardware
- Key implementations:
  - **CPU** virtualization: vCPU
  - **Memory** virtualization: memory isolation between virtual machines(VMs)
  - **I/O** virtualization: pass-through, virtio, …
- **Hypervisors**(Virtual Machine Monitors, VMMs):
  - the software that provides the virtual HAL for VMs
  - Type-1 hypervisors (bare-metal)
    - `Xen, Hyper-V`
  - Type-2 hypervisors (as OS applications)
    - `VMware Workstation, Virtualbox`

Background: KVM (Kernel-based Virtual Machine)

- Part of the **Linux** kernel (built-in or as kernel modules)
- Implemented by each architecture's code(x86, arm64, riscv, ...)
  - KVM uses **ISA's virtualization extension**(VT-x, ARM's virtualization extension, RISCV's H extension)
  - KVM uses linux's scheduler to run VMs' **vCPUs** as **Linux processes**
- ARM64's privilege levels:
  - **EL0** : apps
  - **EL1** : os or VM os
  - **EL2** : hypervisors
  - **EL3** : secure monitor
- Highvisor/Lowvisor on ARM64's KVM:
  - highvisor runs in EL1 and take advantage of Linux's kernel
  - lowvisor runs in EL2 as a hypervisor
- in userspace, **QEMU** is used to interact with KVM



Challenge 1: problems of Mutex in Rust

- `std::sync::Mutex(std), spin::Mutex(no_std)`
- self-deadlocks
  - `lock(A)`
  - `... // maybe later or in another function`
  - `lock(A) // waits forever!`

- lock ordering problems
  - `CPU0: lock(A) ... lock(B) // wait for CPU1`
  - `CPU1: lock(B) ... lock(A) // wait for CPU0`

Challenge 2: raw pointers in unsafe Rust

- unsafe is common in low level Rust system programming:
  - `pagetable walking`
  - `memory allocation`
  - `atomic instructions/inline asm`
  - `memory-mapped I/O`
  - `...`
- can we wrap the **raw pointers** to safely access physical memory regions?

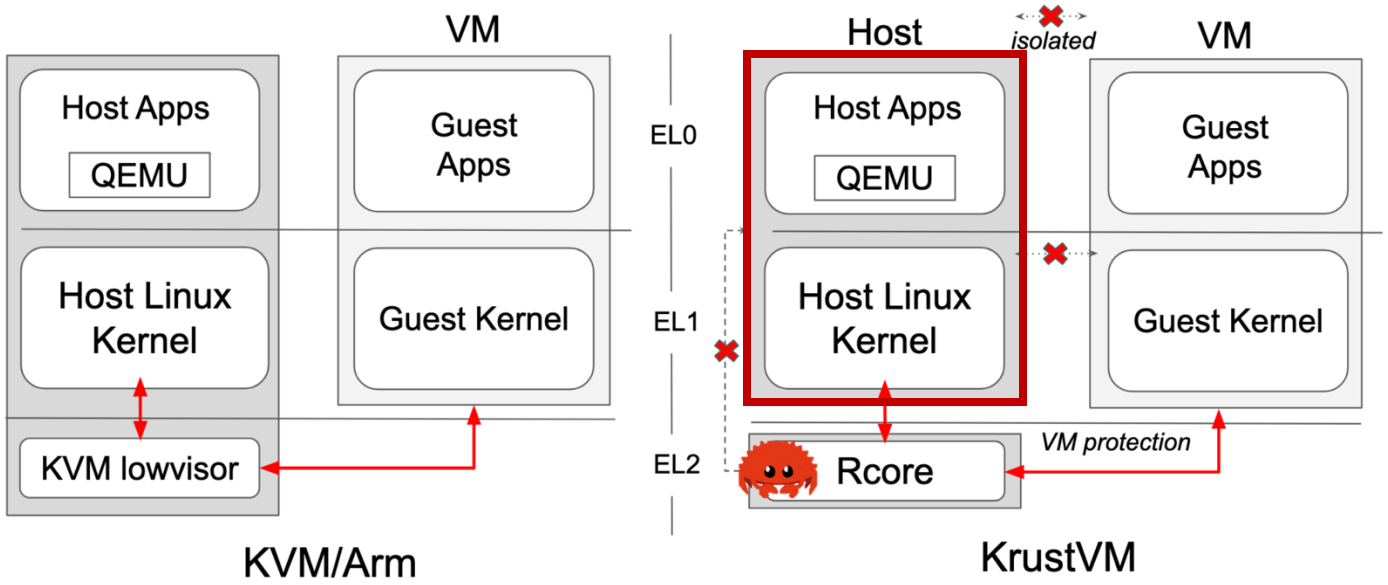
Challenge 3: big TCB(Trusted Computing Base) of KVM

- attackers can exploit bugs in KVM and Linux kernel to steal precious data from VMs.
- since KVM uses Linux kernel functionality like scheduling vCPUs, the TCB is the **whole kernel**.

KrustVM: KVM, but with a new Rust code core

Contributions

- designed **KMutex** to avoid self-deadlocks and ensure lock ordering
- designed **Safe Pointers** to guard raw pointer accesses
- rewrite KVM as **KrustVM** with some functionality in Rust as TCB (named **Rcore**)



KMutex

- **avoid self-deadlocks**

```
impl<T: ?Sized> KMutex<T> {
    pub fn lock(&mut self)->Guard<'_,Self>{...}
}
```

- `&mut` will force Rust **compiler** to check if the Guard has not been dropped, no one can access `KMutex`
- this is because in Rust there can only exist **one** mutable reference in the variable's lifetime.

KMutex(cont')

- **force lock ordering**
- manually designed a lock order graph for all Rcore types
- use trait CanGet* for ordering, for example:
  - `LEntry, VM info, PMEM info` will be implemented `CanGetS2PTInfo` trait
- the graph has no loop, so all locks will be forced in order

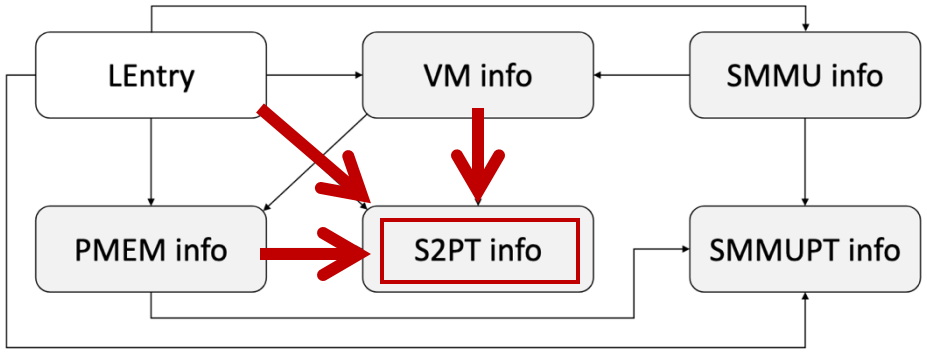


Figure: the order of getting locks

```
1  pub unsafe trait CanGetA {}
2  // SAFETY: We've manually verified the order
3  unsafe impl CanGetA for B {}
4
5  pub fn get_a<T:CanGetA>(_:&mut T)->&mut KMutex<A>{...}
6
7  fn foo(ref_b: &mut KMutex<B>) {
8    let mut b = ref_b.lock();  b: &mut B
9
10   /* b in the following line gets converted to
11     "&mut *b" by the compiler */
12   let ref_a = get_a(b);
13
14   /* this does not compi
15      because ref_a's lifetime is not over */
16   let ref_c = get_c(b);
17
18   let mut a = ref_a.lock();
19   a.do_a_work();
20
21   // we can use b after this
22   b.do_b_work();
23 }
```

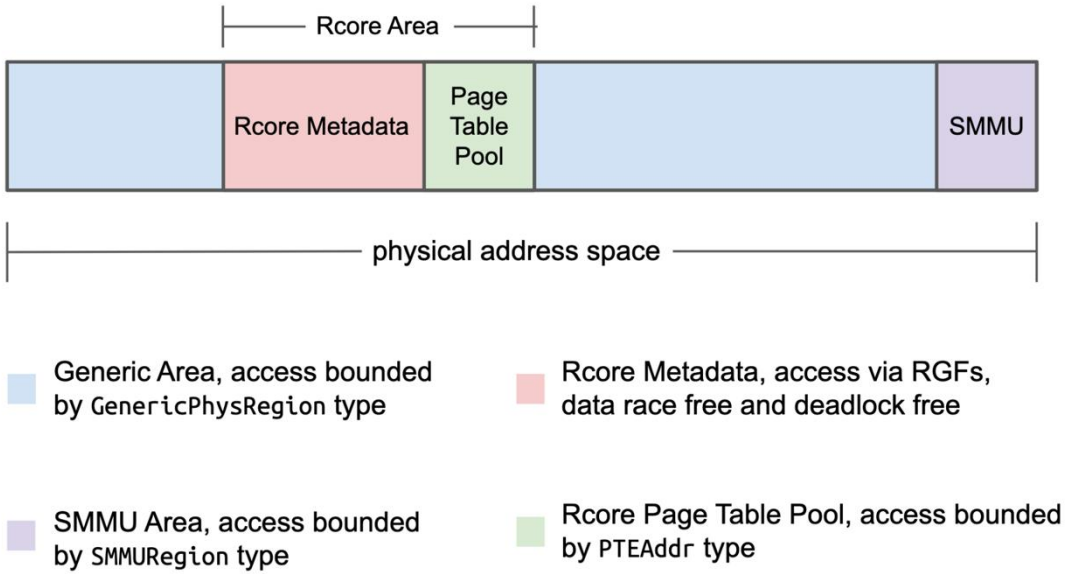
reborrowed here **[line 12]** as anonymous &mut
and should live as long as ref_a **[line 12-19]**

B requests lock first,
but can only be used after anonymous &mut
of B(by ref_a) is dead ☺

1. https://quinedot.github.io/rust-learning/st-reborrow.html
2. https://github.com/rust-lang/reference/issues/788

Safe Pointers

- partition physical memory as regions
- manually define *Pointers* for these regions with range check



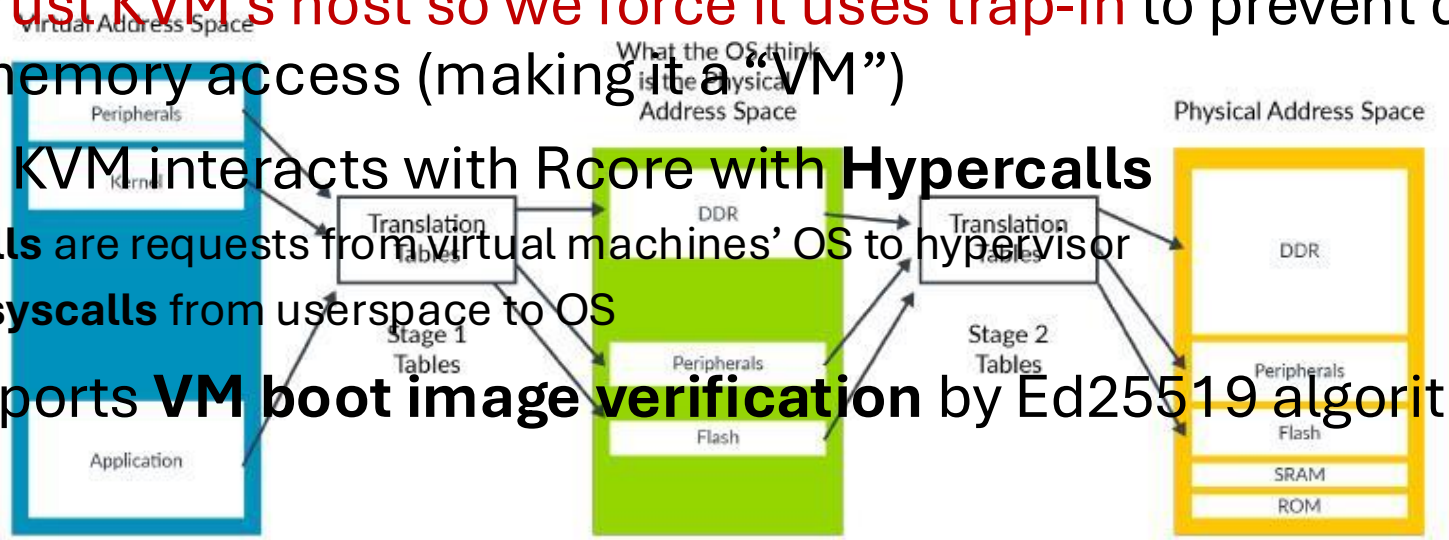

```rust
impl GenericPhysRegion {
  pub fn new(start_addr: usize, size: usize) ->
  ↪ Option<Self> {
    let end = start_addr + size;
    // overlap check
    if (end > RCORE_AREA_START &&
        RCORE_AREA_END > start_addr) ||
       (end > SMMU_AREA_START &&
        SMMU_AREA_END > start_addr) {
      None
    } else {
      Some(Self { start_addr, size })
    }
  }

  // returns a mutable `u8` slice for the caller
  // to access generic area memory
  pub fn as_slice(&self) -> &'static mut [u8] {
    // convert the physical address to the virtual
    ↪ address
    let va = pa_to_va(self.start_addr);
    unsafe {
      core::slice::from_raw_parts_mut(
        va as *mut u8, self.size,
      )
    }
  }
}
```

range check

Rcore uses **u8 slice** to access this memory

KVM host is untrustable

- Rcore unmap *itself*, all *VM's S2PT* and *host's S2PT* from host
  - **S2PT(Stage-2 Pagetables)** translate VM's guest Physical Address to real Physical Address
  - used for memory isolation
- we don't trust KVM's host so we force it uses trap-in to prevent direct physical memory access (making it a "VM")
- Therefore, KVM interacts with Rcore with **Hypercalls**
  - **hypercalls** are requests from virtual machines' OS to hypervisor
  - just like **syscalls** from userspace to OS
- Rcore supports **VM boot image verification** by Ed25519 algorithm



Evalution

- benchmarks: `hackbench, netperf, ApacheBench, memcached, redis`
- Comparable to KVM with max 10% overhead

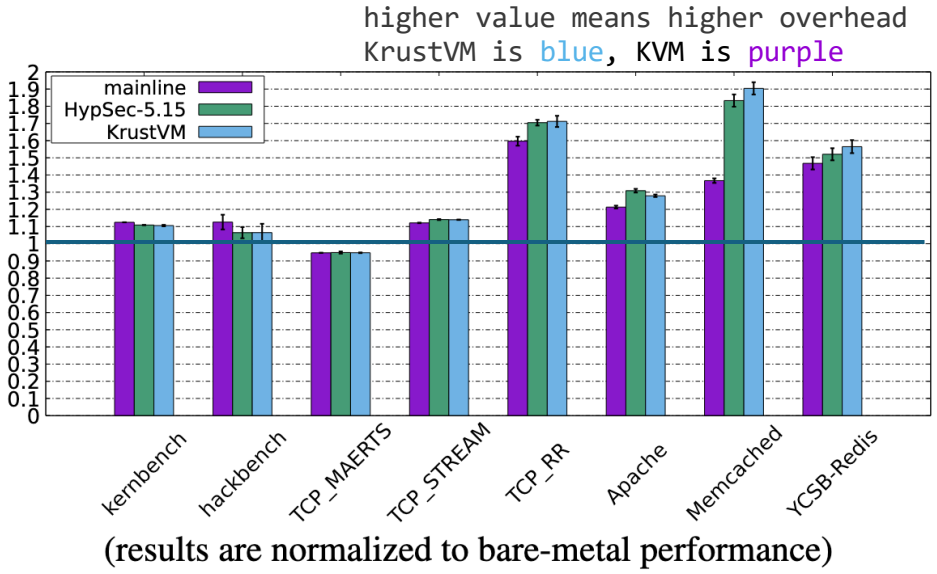


Figure 4: Application Benchmark Performance.

My comments on the paper

- reminder of previous paper presentation:
  DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency

- KrustVM and DRust both take advantage of **Rust**:
  - Leveraging Rust's ownership and lifetime check for (distributed) system programming
  - and more?
- KrustVM's Rcore uses a **formal verification** tool to verify part of its code:
  - Formal verification has become more and more important for system software

Thanks for Watching

paper slides created by Yulong Han @ Peking University