Securing a Multiprocessor KVM Hypervisor with Rust

Yu-Hsun Chiang
National Taiwan University
Taipei, Taiwan
ototot@csie.ntu.edu.tw

Wei-Lin Chang
National Taiwan University
Taipei, Taiwan
r09922117@csie.ntu.edu.tw

Shih-Wei Li
National Taiwan University
Taipei, Taiwan
shihwei@csie.ntu.edu.tw

Jan-Ting Tu
National Taiwan University
Taipei, Taiwan
b08902011@csie.ntu.edu.tw

ABSTRACT

As computations have increasingly shifted to virtual machines (VMs) running on a hypervisor, the security of the hypervisor is of critical concern. Rust has gained significant traction among developers due to its software safety guarantees and performance efficiency. This work explores building on Rust's safety features to construct a secure KVM hypervisor. We retrofit KVM to incorporate a Rust-based core to protect virtual machines. We build on Rust's type and lifetime system in a novel way to secure the core's memory accesses in a concurrent environment. Our resulting KVM implementation, KrustVM, incorporates a data race and deadlock-free core to protect VM confidentiality and integrity against privileged attackers who control the host Linux kernel while preserving KVM's commodity features and performance.

1 INTRODUCTION

Rust is an emerging programming language that offers robust security while retaining performance and efficiency. Rust introduces unique features, such as ownership and lifetimes, to effectively address memory safety bugs that programmers may encounter, such as out-of-bound memory access, use-after-free, and double-free. These bugs are prone to occur in unsafe programming languages like C. Rust's compiler checks against rules and properties to enforce its safety features at compile time. Various previous works [4, 10, 12, 14, 34, 48, 60, 67, 68] have adopted Rust to implement core systems software with critical security and performance requirements. Officials have urged [41] adoption of memory safety languages like Rust to eliminate bugs.

Modern systems software supports multiprocessor platforms, and concurrently accesses shared memory resources. Synchronization is necessary to protect the safety of these concurrent memory accesses. Rust enables fearless concurrency [54] by exposing synchronization primitives and message passing APIs to support concurrent programming. Rust provides a synchronization primitive, `Mutex`, and its API to enable mutual exclusive operations and mitigate data races. Nevertheless, Rust's concurrency API is only available to userspace programs, not low-level and privileged systems software like OS kernels or hypervisors. Even if Rust's concurrency API, like `Mutex`, is available to systems software, the API cannot satisfy critical security demands. Incorrect usage of Rust's concurrency API could lead to deadlock that compromises safety requirements and functionality. Rust cannot identify errors that double lock the same `Mutex`.

In addition, Rust's enforced rules are too restrictive for low-level systems software, which requires operations that make memory accesses through raw pointers to manage page tables and scrub reclaimed memory pages. These accesses are deemed unsafe by Rust and forbidden by the compiler. Rust allows developers to turn off the compile-time checks using the `unsafe` keyword, tasking developers to validate the safety of the unsafe code. A bug found in the unsafe usage could compromise the safety guarantee of the entire program. Previous work [9] has disclosed several memory safety bugs from unsafe Rust code.

Hypervisors are privileged software that manages hardware resources to provide the virtual machine (VMs) abstraction and host these VMs. In addition to their prevalent cloud deployment, recent efforts proposed deploying hypervisors to automotive [20] and mobile [26] systems to isolate critical components into VMs. Widely used commodity hypervisors, such as KVM [31] or Hyper-V [46], include a large and complex trusted computing base (TCB) to satisfy users' requirements in performance and functionality. For example, KVM and Hyper-V each integrate a privileged OS kernel in its TCB. These hypervisors were written in unsafe languages like C, making them vulnerable to memory safety bugs. Attackers that exploit vulnerabilities in the hypervisor TCB may gain the ability to steal or modify secret VM data or compromise the VM's execution.

This work explores building on Rust's unique language features to construct a secure Linux/KVM hypervisor. Instead

of rewriting the entire KVM with Rust, we restructured KVM to incorporate a small Rust-based TCB (denoted as *Rcore*) to protect VM confidentiality and integrity against the untrusted host Linux kernel in KVM. On top of Rust's existing memory safety guarantees, we leverage Rust's unique language features to further secure Rcore's memory accesses in a concurrent environment.

To address the limitations of Rust's concurrency API, we introduced `KMutex`, a new Rust type that implements the semantics of Rust's `Mutex`. Unlike `Mutex`, `KMutex` can be used by low-level systems software like hypervisors. Incorporating the design of *Reference Getter Function (RGF)*, `KMutex` enables the Rust compiler to detect deadlock at compile time. Programmers can associate a given data object type with an intended `KMutex`. We assign a unique `KMutex` instance to each shared object and abstract accesses to an object in its RGF. `KMutex` provides a safe `lock` method to provide a secure way to access the wrapped shared data. Accesses can be made via the `lock` method, including reading or writing to the shared object. An RGF invokes the `lock` method of the object's associated `KMutex`; it returns a reference to the wrapped object when its respective `KMutex` is acquired. The method incorporates the underlying architecture's synchronization primitive that guarantees exclusive access at the hardware level.

`KMutex` and RGFs are employed to secure concurrent memory accesses in low-level software. We leverage Rust's lifetime and ownership system in a novel way, enabling compile-time checks that guarantee lock ordering and prevent double locking the same `KMutex`. We mandate that all access to shared objects occurs exclusively through RGFs, and we implement a static approach to enforce consistent ordering across all RGF invocations. These measures, combined with `KMutex`'s built-in double locking prevention features, ensure that `KMutex` usages do not cause deadlocks, thus guaranteeing deadlock freedom. Furthermore, by restricting access to shared objects solely through RGFs, we ensure that accesses to shared objects in the software can only occur when the respective `KMutex` is held, thus guaranteeing data race freedom.

We further introduce the customized *Safe Pointer* types to hold raw pointers to secure their accesses. Safe pointers ensure that each designated type can exclusively store pointers used against the memory region containing the corresponding data. Raw pointer accesses in Rcore are guarded by the *Safe Pointer* types to ensure addresses lie in the intended memory region. An address-bound check is rigorously enforced inside those types during construction. The check is performed against the addresses that the raw pointers point to. This hardens against raw pointer exploitation since memory accesses can only be made with the raw pointer if the check is validated; out-of-bounds memory accesses are rejected. For instance, when Rcore uses raw pointers to update page tables, we ensure that Rcore never mistakenly updates VM execution state metadata.

We leverage a previous design [36] and utilize Arm's hardware virtualization extensions to retrofit KVM in Linux 5.15 into KrustVM. By employing the intended usage pattern of `KMutex` and RGFs, we secure Rcore's concurrently accesses to shared memory objects to ensure data race and deadlock freedom while utilizing fine-grained locking to optimize performance. We adopt a modular-based design to construct Rcore to leverage Rust's memory security guarantees. We use *Safe Pointer* types to secure Rcore's memory accesses. KrustVM supports multiprocessor VMs running on Arm64 multiprocessor hardware while retaining KVM's commodity features. Performance evaluation of KrustVM shows that it incurs modest overhead to application workloads compared to mainline KVM and the C-based secure KVM implementation that adopts the same design [36]. We demonstrate the practicality of securing a widely adopted commodity hypervisor using Rust.

In summary, this paper makes the following contributions:

- We leverage Rust's language features to extend its safety guarantees beyond memory safety. We developed a specialized `KMutex` type that, when adopted, restricts the program's access to shared memory through its interface. This restriction ensures deadlock and data race freedom in low-level systems at compile time, even at scale.
- We introduce the Safe Pointer type to securely store raw pointers, ensuring each pointer resides within its designated memory region. By constructing customized Rust types with *Safe Pointers*, we safeguard raw pointer access by enforcing an address-bound check during construction. This mechanism rejects out-of-bound memory accesses.
- We build on `KMutex` and *Safe Pointers* to construct KrustVM, a Rust-based secure multiprocessor KVM hypervisor that enforces robust VM protection. An evaluation of KrustVM shows that it results in modest performance overhead compared to mainline KVM and a KrustVM counterpart written in C, demonstrating Rust's practicality in low-level system software.

2 BACKGROUND

2.1 The Rust Programming Language

Rust prioritizes safety and speed. Unlike traditional languages like C/C++, Rust eliminates the need for manual memory management to ensure memory safety. Rust does not rely on garbage collection. Instead, it introduces lifetimes and ownership, requiring programmers to follow specific rules. This approach, where programming rules are statically enforced,

allows Rust to perform similarly to C. Rust's compiler has complete control over optimizing the machine code it emits. Rust's safety rules also guarantee the absence of memory safety bugs when followed, as the compiler automatically checks and prevents rule violations.

Ownership and Lifetimes. In Rust, each piece of data is said to be *owned* by a single variable, and it is automatically *dropped* (freed) when the variable's *lifetime* ends. A variable's lifetime ends as the program control flow exits the block in which the variable is declared.

Borrowing. Ownership lacks the flexibility in argument passing. A variable can borrow ownership from another variable to acquire a *reference* to the data. Rust's borrowing rule enforces *aliasing xor mutability* meaning there can be multiple shared references or a single exclusive reference. Shared references can only be read and not modified, and exclusive references allow for reading and modifying the value. In Rust, these mutable references that exclusively access the data can be used by declaring the intended data as mutable (using the `mut` keyword) and then using the `&mut` keyword to borrow the data mutably.

In Listing 1, line 6 causes a compile error because it tries to create a mutable reference (`z`) to `x`, while `y` already borrowed `x` immutably. `y`'s lifetime ends on line 9 as it gets used for the last time; therefore, `z` can be created on line 11 and used on line 12. However, if line 16 is uncommented, `y`'s lifetime would be extended to line 16, making the creation of `z` on line 11 break the borrowing rules.

```
1  let mut x = vec![1, 2, 3];
2  let y = &x; // immutable borrow of x
3
4  /* this line would fail to compile
5     because x is borrowed immutably by y */
6  let z = &mut x;
7
8  println!("x = {:?}", x); // This line works
9  println!("y = {:?}", y); // This line works
10
11 let z = &mut x; // mutable borrow of x
12 z.push(4);
13
14 /* this line would fail to compile
15    because x is borrowed mutably by z */
16 // println!("y = {:?}", y);
```

Listing 1: Rust enforces *aliasing xor mutability*

Drop Traits. Rust traits are properties or interfaces that can be implemented on types; traits typically require the implementing type to supply function implementations for its trait methods. Some traits in Rust have intrinsic meaning to the compiler. For example, the `Drop` trait tells the compiler that a type has special freeing code, and the `Drop` trait's `drop` function should be invoked when an instance of the type's lifetime ends.

2.2 Arm Virtualization Extensions

Arm's Virtualization Extensions (VE) [5] introduced a higher privileged hypervisor mode (EL2) on top of the existing kernel mode (EL1) and user (EL0) mode. Arm VE provides EL2 registers to run software in an isolated execution context and address space from EL1 and EL0 to multiplex the execution context of the host and guest VMs. Arm provides stage 2 page tables (S2PTs) to support memory virtualization. S2PTs only affect the software running in the EL1 and EL0 modes, not EL2. The guest kernel manages the stage 1 page tables (S1PTs) to translate guest virtual addresses (gVAs) to guest physical addresses (gPAs). The hypervisor, e.g. KVM, enables stage 2 page tables when running the VMs, which translate the gPAs (or intermediate physical addresses (IPAs)) to host physical addresses (hPAs) to access the machine's memory. Arm provides an IOMMU, the System Memory Management Unit (SMMU) [6] to prevent Direct Memory Access (DMA) attacks.

3 ASSUMPTIONS AND THREAT MODEL

We assume an attacker or a curious administrator who aims to compromise the integrity and confidentiality of VM data, which includes code or sensitive data stored in the VM's CPU registers, memory, and I/O buffers. VM data exclude generic hardware configuration information, such as the power control and virtual interrupt states, since they do not contain secrets. Like other previous work [36–38, 74], we do not consider attacks against VM availability. We assume attackers have remote access to the hardware. A remote attacker can exploit bugs in the user space VMM and the host kernel integrated with KVM, or control the VM management interface to access VM data. We exclude physical [22] and side-channel attacks from the threat model. We assume a VM does not voluntarily reveal its sensitive data, intentionally or by accident. However, a compromised VM may try to attack other hosted VMs for which we provide protection.

4 OVERVIEW

This section first gives an overview of KrustVM, a multiprocessor KVM hypervisor that integrates a Rust-based TCB called Rcore to protect VMs. We next discuss `KMutex`, a novel Rust type with lock-safe API that protects concurrent memory accesses. `KMutex` facilitates the construction of data race-free and deadlock-free low-level systems software like hypervisors. In addition, we discuss the *Safe Pointer* type that we introduced to implicitly enforce bound checks against raw pointer usage to protect memory safety, complementing Rust's incapability in guaranteeing safety for raw pointers.

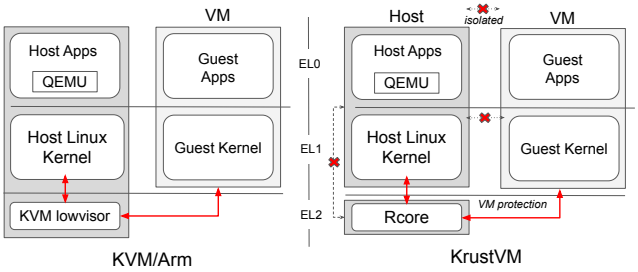


Figure 1: KrustVM Architecture.

4.1 KrustVM Overview

We leverage the hypervisor design introduced by our previous work, HypSec [36], to retrofit KVM into KrustVM. Figure 1 shows the architecture of KrustVM. Unlike KVM (shown on the left) that includes the host Linux kernel in its TCB, KrustVM relies on a Rust-based TCB, called Rcore to protect VM confidentiality and integrity against an untrusted KVM host that encompasses the host Linux kernel. We separate access control from resource allocation to reduce the TCB. Rcore focuses on protecting VM data in CPU and memory from the KVM host. KrustVM relies on the KVM host for resource allocation and scheduling. Rcore enforces access control to ensure resources assigned to the VM are inaccessible to the untrusted KVM host. Many applications are designed to use encrypted I/O channels. Like previous work [1, 26, 36–38], we assume VMs employ an end-to-end approach [56] to protect I/O data and avoids duplicating the protection efforts.

Given Arm's growing popularity in deployments [3, 51, 69], our current KrustVM implementation secures KVM for Arm. Section 6.1.1 details how we leverage Arm's VE features to retrofit KVM into KrustVM. Rcore protects VMs' CPUs by interposing VM exits. For events that require the host's functionality, Rcore saves the VM CPU registers from the hardware to its private memory, which the host cannot access, and then restores the host's CPU registers to the hardware. Rcore performs the same action in reverse before entering the VM.

To protect VM memory, Rcore uses Arm's S2PT to enforce memory access control. Rcore enables S2PTs when running the KVM host and VMs so that they do not have direct access to physical memory. Rcore manages and updates all S2PTs. Rcore allocates S2PTs for all entities from their respective page table pool to which the host and VMs have no access. Rcore employs an identity map in the KVM host's S2PT, translating each KVM host's IPA to an identical hPA. This allows KrustVM to reuse Linux's memory allocator to manage memory implicitly. Rcore unmaps VMs' private memory pages from the KVM host's or other VMs' S2PTs. Rcore also unmaps its memory pages from all S2PTs, making them inaccessible to VMs and the host. Unauthorized access to unmapped pages made by a compromised host results in an S2PT fault routed to Rcore, allowing it to reject such invalid host memory accesses.

A compromised KVM host can control devices to perform DMA to read or write VM memory. Rcore leverages Arm's SMMU to protect DMAs. Rcore allocates and manages SMMU page tables from its private memory for each DMA-capable device. It trap-and-emulates the host's access to the SMMU to prevent the host from controlling the SMMU.

4.2 KMutex

Locks are commonly used to synchronize concurrent shared memory accesses. However, incorrect usage of locks can compromise data integrity and service availability. To address these concerns, KrustVM leverages Rust's type system to guarantee freedom from data races and deadlocks at compile time. Our approach supports fine-grained locking and is designed for easy adoption, ensuring high performance and minimal development effort.

In addition to its promise of memory safety, Rust is known for its *Fearless Concurrency* [54]. Rust's ownership model carefully tracks the usage of every piece of data and automatically derives information on data shareability among threads. Consequently, Rust can preemptively detect and prevent data races at compile-time. Rust supports concurrent programming by exposing structures such as `mpsc` for message passing or `Mutex` for managing shared states.

However, Rust's `Mutex` has limitations that prevent direct use in building a secure multiprocessor hypervisor. First, the library only supports user space programs. For instance, on Linux, the `Mutex` in the Rust standard library is implemented with the `pthreads` library. This makes these data structures unsuitable for use in a hypervisor or other low-level systems. Second, Rust's synchronization primitives do not guarantee deadlock freedom, so their use might accidentally compromise a service's availability. We aim to develop a safer alternative that ensures deadlock freedom while preserving the ease of use.

4.2.1 KMutex Design. We developed a custom lock data type named `KMutex`, designed to safeguard the data within Rcore. `KMutex` is a generic data structure capable of holding arbitrary data alongside a lock. Every instance of `KMutex` will have its lock, so every `KMutex` can protect its inner data accordingly. Instead of exposing methods such as `new` to create `KMutex` instances, we require that all data and `KMutex` initialization occur during the boot time. This choice is deliberate, as it aligns with our objective of safeguarding static global metadata.

The sole exposed method of `KMutex` is `lock`. The program must call the `lock` method to gain access to the data encapsulated within `KMutex`. The `lock` method first acquires the lock and subsequently returns a *Resource Acquisition Is Initialization* (RAII) guard. This guard is an instance of a generic type. In Rust, generic type definitions can be used with various concrete data types, enabling code reuse. The guard contains a mutable reference to `T`, enabling us to access and manipulate the data encapsulated by `T`.

To release the lock, we have implemented the `Drop` trait for `Guard<T>`. This allows us to rely on the Rust compiler to track the use of the RAII guard and automatically insert the unlock call to release the lock. This removes the need to manually release the lock and eliminates concerns about releasing it at the wrong time. Consequently, we can access data protected by the lock with confidence, without worrying about the unlocking process.

We aim to reduce the code size of Rcore and avoid integrating a lock library with Rcore. Therefore, we implement both locking and unlocking mechanisms through Arm's exclusive access instructions [42]. We wrapped the instructions in inline assembly.

We design our lock API carefully to achieve deadlock freedom and eliminate misuse of `KMutex` that compromises the availability of KrustVM. The key idea of our approach is to leverage the ownership model of Rust to ensure that the `lock` method of `KMutex` cannot be called unless the existing RAII guard is dropped. The first three lines of Listing 2 show the function signature of the `lock` method from Rust's default `Mutex`. It only requires the caller to pass an immutable reference when calling the `lock` method. Since Rust permits immutable references to be created arbitrarily many times, a careless programmer may legally call `lock` when holding the respective `Mutex`, leading to a self-deadlock.

The last three lines of Listing 2 show the function signature of the `lock` method in `KMutex`. This looks similar to `Mutex` since we still want the user of this type to have a similar experience as using the one from the Rust standard library. The main difference of the `lock` method between `KMutex` and `Mutex` is the presence of the `mut` keyword (see line 5 of Listing 2). We embed the mutable reference `mut` into `Guard`, so the Rust ownership model will release the `mut` reference once the `Guard` gets released. Meanwhile, since the guard is holding the `mut` reference and the Rust compiler rejects the creation of other `mut` references of `KMutex`, a program cannot call `lock` to create another guard while holding the one created in a previous call to `lock`, thus preventing self-deadlocks.

4.2.2 *Reference Getter Functions (RGFs).*

We assign a unique `KMutex` instance to each of Rcore's shared object types. Rcore must call the `KMutex`'s `lock` method to acquire the respective inner shared objects. To manage access to these shared memory objects, Rcore uses a set of RGFs; each RGF is bound to a specific shared object type. Once the lock is acquired, the corresponding RGF returns a reference to that type's `KMutex` instance. Rcore dereferences the reference to `KMutex` instance that wraps the actual shared object to perform the intended access. Accesses to shared objects in Rcore were made through the respective RGF. This approach ensures that access to shared objects in Rcore can only occur when the respective `KMutex` is held, guaranteeing data race freedom. In the following subsection, we discuss how we restrict the invocation order of RGFs in Rcore to ensure deadlock freedom. RGFs follow Rust's interior unsafe [49] design pattern. RGFs expose a safe interface but encapsulate unsafe operations in its actual implementation. This eliminates significant use of unsafe code blocks in Rcore's implementation. We discuss a concrete RGF implementation in Section 5.2.

```
1 impl<T: ?Sized> Mutex<T> {
2   pub fn lock(&self)->LockResult<MutexGuard<'_,T>>{...}
3 }
4 impl<T: ?Sized> KMutex<T> {
5   pub fn lock(&mut self)->Guard<'_,Self>{...}
6 }
```

Listing 2: Function signature of the lock method of `KMutex` and `Mutex`

4.2.3 *Enforcing Lock Order.*

We force the caller to pass a mutable reference to the `lock` method to rule out self-deadlocking. However, deadlock-freedom cannot be ensured yet since deadlocks can occur not only when a single lock is acquired twice but also when multiple locks wait for each other. We define a partial order among the equivalence classes of `KMutex` for the shared data types in Rcore. For a given partial order, if $A \leq B$, we ensure that it is impossible to acquire A after acquiring B. Failing to follow the order can result in lock order inversions. We utilize Rust's lifetime system and traits to safeguard against such inversions at compile time. Figure 2 presents the lock acquisition order in the Rcore implementation. Note that this order is valid, i.e., containing no deadlocks, since the graph does not contain cycles.

We impose restrictions on RGFs to enforce the intended partial order dependency when acquiring/releasing locks to avoid deadlocks when nested-locking multiple locks. An RGF that returns a mutable reference to `KMutex<A>` must be called with a mutable reference of `T` that implements the trait `CanGetA`, where `A` is the name of the type. For each shared type `A`, all types `T` where `T` has an edge that points to `A` in Figure 2 implements the trait `CanGetA`. Rust suggests marking traits as `unsafe` when improper implementation could result in a safety violation. For `CanGet*` traits, implementing any of them without following the desired partial order (Figure 2)

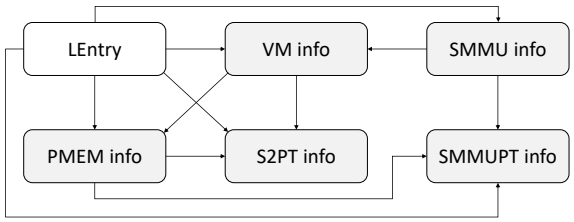


Figure 2: Lock acquisition order within Rcore. Here, a block A points to B means that $A \leq B$ is in the given partial order. Table 1 defines the respective data type for each block.

can lead to a cycle in the acquisition order, which could cause a deadlock. Therefore, we mark all of the `CanGet*` traits as `unsafe`, so that we check the implementation follows the desired partial order and ensure safety.

```
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();
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 compile
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 }
```

Listing 3: An example that violates the predefined lock order. (Assuming $B \rightarrow A$ is in the lock acquisition order graph)

For an RGF of `A`, we pass a `&mut T, where T: CanGetA` to get `&mut KMutex<A>`. For instance, in line 13 of Listing 3, we get a `&mut KMutex<A>` (`ref_a`) by passing a mutable reference `b` into the RGF `get_a` (`b` is of type `&mut B`, and `B` implements `CanGetA` on line 3). When `b` is passed to the RGF, instead of moving `b` into the function, the Rust compiler actually *reborrows* (i.e. changing `b` to `&mut *b`, line 12), and creates an anonymous mutable reference to `B`, then moves it into the RGF. The RGFs are implemented such that the required passed-in mutable reference (in this case, the anonymous reference) has the same lifetime as the returning mutable reference `ref_a` of type `&mut KMutex<A>`. The anonymous mutable reference must live as long as `ref_a`. In this case, from line 13, where the anonymous mutable reference is first created, to line 20, where `ref_a` is used for the last time. Between lines 13 to 20, which the anonymous mutable reference to `B` (`b`)'s lifetime spans, Rust does not allow any access to `B` other than the anonymous mutable reference, otherwise, two mutable reference to the same object will have their lifetimes overlap. And, since there is no way to use the anonymous mutable reference because it is moved into the RGF, any access to `B` can only be made via a different reference to `B`. Therefore, the Rust compiler forbids any access to `B` before `ref_a` gets dropped because the anonymous reference has the same lifetime as `ref_a`'s.

4.3 Safe Pointer

When implementing low-level systems, like hypervisors, raw pointer access is inevitable. However, incorrect raw pointer accesses can easily violate Rust's ownership model. Rust allows parts of a program to be marked as *unsafe*, disabling compiler checks to support operations such as raw pointer accesses. Many software bugs stem from discrepancies between the address a pointer references and the intended address it should point to. For example, the S2PT walking code must calculate the addresses of each level's page table entry. If the calculation is incorrect, a raw pointer access against the resulting addresses may inadvertently access or modify an unrelated memory region. Such unintended behavior could compromise VM security, potentially leading to writes to guest memory or other hypervisor metadata. To address this problem, we introduced *Safe Pointer* types to hold raw pointers. We bound-check the addresses that the pointers point to, ensuring that each designated type can exclusively store pointers within a designated memory region.

Raw pointers in Rust can be unaligned or null, but they must be non-null and aligned when dereferenced. Furthermore, references in Rust must be non-null, aligned, and pointing to memory containing a valid value. We extend these semantics and create a set of Safe Pointer types. Each Safe Pointer type is associated with a specific memory region. We require Safe Pointers to be non-null, aligned, and pointing to a memory inside the corresponding memory region. Instead of validating the address every time we access it, we validate a given address inside the constructor of a Safe Pointer type. If the given address does not fall within the desired area, we reject the construction and return `None` to the caller. Rust provides a more rigorous type safety guarantee than C. It forbids arbitrary type casts and ensures data are sanctioned by its type. Since a Safe Pointer type can only be constructed through its constructor, we can ensure each of its instance points to a desired memory region.

Table 1: Rcore Metadata

Name	Description of Data
VM info	The per-VM execution state metadata.
S2PT info	The S2PT pool allocation status.
PMEM info	The memory ownership and sharing status.
SMMU info	The SMMU management metadata.
SMMUPT info	The SMMUPT pool allocation status.

Safe Pointer types simplify programmers' efforts to reason about the validity of a pointer. Based on the type semantics and Safe Pointer types, programmers can dereference safe pointers and update data inside them without worrying about corrupting an unexpected memory region. This costlessly improves the security of systems that require many direct memory accesses.

5 SECURING KRUSTVM WITH RUST

5.1 Rcore Metadata

The resulting Rcore implementation supports multiprocessor VMs running on multiprocessor hardware. Rcore concurrently runs on different processor cores and may access shared memory. We build on Rust's safety checks while leveraging `KMutex`'s lock-safe API and Safe Pointer types, achieving strong memory safety guarantees. To achieve the best performance possible, Rcore uses fine-grained locking to protect concurrent memory accesses. Rcore concurrently accesses Rcore Metadata and the Page Table Pool. As its name suggests, the Rcore's Page Table Pool keeps private pools of physical pages for S2PTs and SMMU page tables. Table 1 shows Rcore's Metadata. The metadata is used for tracking page table allocation status (`S2PT info` and `SMMUPT info`), physical memory page ownership (`PMEM info`) and VM states management (`VM info`), SMMU page table metadata (`SMMU info`), etc. We constructed custom types to store these Rcore Metadata.

5.2 Modularizing Rcore

In our implementation, we modularized Rcore into two layers: Safety and Func layers. The Safety layer includes Rcore's functionality that requires unsafe Rust. By encapsulating unsafe Rust, we could implement the complex features for VM protection in the Func layer in safe Rust, leveraging Rust's compile-time check to enforce safety guarantees.

Safety Layer. The Safety layer follows Rust practices and exposes a safe API to the Func layer that wraps the unsafe usage. This includes RGFs and Rcore's custom Safe Pointer types. Section 4.2 discusses the approaches we employed to secure data race and deadlock freedom for Rcore memory accesses to shared data within the Rcore Area, including the Rcore Metadata and page tables (see Figure 3). Section 4.3 discusses the Safe Pointer types we created to ensure spatial memory safety by imposing bound checks.

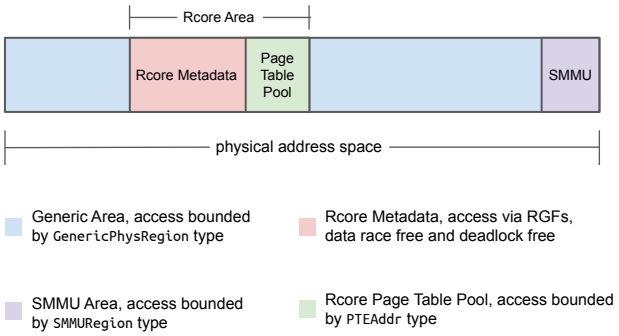


Figure 3: KrustVM Memory Regions.

Func Layer. Rcore categorizes metadata and memory objects into different Rust types. The Func layer implements the methods for these types. For instance, Rcore implements the methods for the type that corresponds to `PMEM info` (see Table 1) to track memory ownership for each memory page. Each type stores a group of related data, and each method implements a function that acts on the instance's fields. Types serve as a valuable tool for constraining the operations performed on data based on their concrete semantics. This makes it easier for programmers to understand operations they can perform on the data and avoid misuse.

The Func layer uses the API exposed by the Safety layer to build complex hypervisor functionality. For example, it invokes the safe RGF (see Listing 4) to obtain a mutable reference to a given Rcore object, such as the `PMEM info` metadata, then invokes its methods to fulfill virtualization tasks, such as assigning page owner to a VM in page fault handling. The Func layer builds on the safety guarantees enforced by the Safety layer.

5.3 Adopting KMutex and Safe Pointers

5.3.1 KMutex and RGFs. The enforced lock acquisition order within Rcore is shown in Figure 2. For functions that take a lock without already holding a lock (the entry node), a zero-sized type `LEntry` which implements every `CanGet*` traits defined in Rcore.

We associate an instance of `KMutex` for each data type in Table 1. We create an instance of `KMutex` for `VM info` and `S2PT info` for each VM. For each of the other data types, a single instance of `KMutex` is used to protect concurrent memory access. We allocate all `KMutex` instances during the boot time of the host machine. As discussed in

Section 4.2.1, this prevents attackers from maliciously creating new `KMutex` instances that override existing ones to compromise lock safety.

We aggregate Rcore metadata structures into a single big structure `RcoreMetadata` (line 1 in Listing 4) to simplify the memory region used by these metadata. All CPU cores share metadata in Rcore; some are per CPU. Data shared by all CPU cores should be protected by `KMutex`. Hence, all of them are of type `KMutex<T>` (an example is line 3 in Listing 4), where `T` is the type that actually stores Rcore metadata. Since there is no existing memory allocator in a hypervisor environment, we directly inform where in the address space Rcore should use. Specifically, we predefined the address to the instance of `RcoreMetadata` and manually initialize this memory region at boot time so it can be used safely (line 7).

```
1  struct RcoreMetadata {
2    [...] // other fields omitted
3    pub pmem_info: KMutex<PMemInfo>,
4    [...] // other fields omitted
5  }
6
7  const RCORE_METADATA_PTR: *mut RcoreMetadata
8        = /* Rcore's memory address */;
9
10 // the RGF of pmem_info
11 pub fn get_pmem_info<T: CanGetPMemInfo>(_: &mut T)
12         -> &mut KMutex<PMemInfo> {
13   // SAFETY: The memory pointed has been initialized.
14   // Also, the data is properly wrapped in a KMutex
15   // and the caller have the permission to get PMemInfo
16   unsafe {
17     &mut (*RCORE_METADATA_PTR).pmem_info
18   }
19 }
```

Listing 4: Rcore Metadata and Reference Getter Function

The Func layer can use RGFs to access metadata with safe Rust, allowing the Func layer to implement all the functionality without using `unsafe` Rust. Each RGF returns a mutable reference to one of the fields in `RcoreMetadata`. Line 10 of Listing 4 shows an example. It returns the mutable reference of the type `KMutex<PMemInfo>`. The RGF is implemented by:

(1) dereference the raw pointer using the `*` operator
(2) pick the `pmem_info` field of `RcoreMetadata`
(3) prepend `&mut` to take the mutable reference of the field and return the mutable reference; once acquiring the reference, Rcore can access the wrapped data within `pmem_info`

Listing 5 shows a function in the Func layer that clears a page's mapping in the host's S2PT. At line 2, the RGF `get_s2pt_info` is called and it returns a reference of `KMutex<S2PTInfo>`, which stores the S2PT information of the host. After obtaining the reference to `KMutex<S2PTInfo>`, the `lock` method is called to lock the structure. At line 5, the method `walk_s2pt` is called to walk the S2PT to check if the leaf entry is zero, and `set_s2pt` is called on line 6 to clear the S2PT entry if not. This demonstrates how the Func layer utilizes the Safety layer.

```
1  fn clear_pfn_host<T: CanGetS2PTInfo>
2      (caller: &mut T, pfn: usize) {
3    let s2pt_info = get_s2pt_info(caller, HOST);
4    let mut guard = s2pt_info.lock();
5
6    if guard.walk_s2pt(pfn << PAGE_SHIFT) != 0 {
7      guard.set_s2pt(pfn << PAGE_SHIFT, 0);
8
9      [...] // details omitted
10   }
11 }
```

Listing 5: An example function in the Func layer

5.3.2 Safe Pointers. We encapsulated unsafe raw pointer usages in Rcore with Safe Pointer types. As shown in Figure 3, Rcore accesses four disjoint memory regions in the physical address space: Rcore Metadata, Page Table Pool, SMMU Area, and Generic Area. The Rcore Area includes the Rcore Metadata and Page Table Pool. The SMMU Area corresponds to the SMMU hardware. The Generic Area refers to the memory region in the physical address space other than the Rcore and SMMU areas. KrustVM relies on the KVM host to allocate memory to VMs from the Generic Area. Rcore reads from or writes to memory from this region to perform several tasks, including authenticating VM boot images and scrubbing memory pages of a terminated VM before returning the pages to the KVM host. We created a set of *Safe Pointer* types to hold addresses respective to the four memory regions in Figure 3 to secure Rcore memory accesses. We discuss how we protect Rcore's raw pointer accesses to the generic area as the case study.

Raw Pointer Access: Generic Area. Generic area accesses are done by calculating raw addresses and writing to them via raw pointers. Raw pointers are necessary here because system RAM is just a range of flat address space to Rcore. We created a Safe Pointer type called `GenericPhysRegion` (as shown in Listing 6) to secure Rcore's access to memory in the generic area. `GenericPhysRegion` can only point to a memory range in the generic area. We carefully check the address provided during construction and reject any invalid address to ensure that every instance of `GenericPhysRegion` points to a valid memory region. This eliminates out-of-bound memory accesses to other areas.

The constructor of `GenericPhysRegion`, namely the `new` method at line 2 in Listing 6, verifies whether the memory range specified by the arguments (start address `start_addr` and access size `size`) is contained within the bounds of the generic area. If the specified range overlaps with the Rcore Area or the SMMU area, the constructor returns a

```
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,
      )
    }
  }
}
```

Listing 6: `GenericPhysRegion` guarantees that every instance points to a valid generic area range

`None` variant, indicating that the construction has failed. Listing 7 shows an example usage of `GenericPhysRegion`. The function `clear_page` from the Func layer takes a physical frame number (pfn) and scrubs the page. `GenericPhysRegion::new()` at line 2 takes the physical address of the page (`pfn « PAGE_SHIFT`) and its size (`PAGE_SIZE`) as arguments to perform the bound check at the constructor. The caller of `GenericPhysRegion::new()` gets a `GenericPhysRegion` if the check passes; otherwise, `clear_page` returns an `Error` type. In the former case, `clear_page` clears the page contents at line 4. Otherwise, `clear_page` returns an error at line 2, effectively preventing the unbounded memory access and propagating the absence of a value up the call stack.

```
fn clear_page(pfn: usize) -> Result<()> {
  let page = GenericPhysRegion::new(pfn << PAGE_SHIFT,
  ↪ PAGE_SIZE).ok_or(Error::InvalidPfn)?;
  // the `fill` method for type &[u8] fills the slice
  ↪ with the value passed in
  page.as_slice().fill(0);
  Ok(())
}
```

Listing 7: Example usage of `GenericPhysRegion`

6 IMPLEMENTATION

Our KrustVM implementation supports KVM in mainline Linux 5.15. The resulting Rcore consists of 3.8K LOC of safe Rust, 0.2K LOC of `unsafe` Rust, 0.4K LOC of ARM assembly, and 10K LOC of C. The ARM assembly code is sourced from the mainline KVM for world switch and exception vectors. The 10K LOC of C is attributed to the verified HACL* [75] crypto library for Ed25519 and AES implementation linked with Rcore. We added and modified roughly 1K LOC to Linux to install Rcore. Since our codebase is based on Linux 5.15, we leverage the LTO support from LLVM [40] to optimize the Rust code.

The `unsafe` Rust code in Rcore consists of the following: (1) unsafe assembly routines to acquire and release locks, (2) unsafe trait implementations for Rcore metadata types to ensure lock order (Section 4.2.2), (3) unsafe LEntry usages (Section 4.2.2), (4) unsafe raw pointers for accessing the isolated memory regions (Section 4.3), (5) unsafe C function calls to the HACL* library, (6) unsafe assembly routines for world switch and cache/TLB maintenance (Section 6.1.3), and (7) unsafe raw pointers for accessing Per-CPU variables (Section 6.2).

We also implemented HypSec [36] in C based on Linux kernel 5.15 (denoted as HypSec-5.15). HypSec-5.15's TCB consists of 3.4K LOC of C and 400 LOC in ARM assembly, linked with the formally verified HACL* [75] library. Rcore is slightly larger than HypSec-5.15's TCB due to the language features that Rcore leverages, such as the traits we defined and implemented to protect memory accesses. We extended QEMU v4.0.0 based on the open source artifact [16] released by previous work [37] that also follows Hypsec's design to support KrustVM and HypSec-5.15.

6.1 Adapting KVM to KrustVM

6.1.1 Retrofit KVM/Arm for VM Protection. As discussed earlier, KrustVM supports Arm-based hardware with VE support. As shown in Figure 1, Rcore runs in EL2 to isolate itself from the KVM host, which the host Linux kernel and VMM run in less privileged EL1 and EL0. In KVM [18], the host has full access to the memory used by the lowvisor in EL2. In KrustVM, the host is isolated from Rcore that runs in EL2. Rcore sets up Arm's HCR_EL2 register to enable S2PT when running the host and trap VM exits and interrupt to EL2. The untrusted host cannot access EL2 registers to disable or control privileged hardware features. Rcore follows HypSec's design and exposes a set of hypercalls for the host to request services that require EL2 privileges. For example, once the host schedules a virtual CPU, it makes a hypercall to Rcore to execute the VM.

KrustVM reuses the device drivers from the KVM host to manage I/O devices and provide I/O virtualization. KrustVM trap-and-emulates VMs' MMIO accesses in the host. VMs perform MMIOs to configure and manage virtual hardware and they involve no sensitive information. Paravirtual I/O

frameworks such as virtio [53] require the hypervisor to access VM memory. Virtio devices from the hypervisor communicate with the VM via shared ring buffers for sending and receiving encrypted data stored in I/O buffers in VM memory. Based on HypSec's design, Rcore exposes the `grant` and `revoke` hypercalls for VMs to share and unshare their memory with the KVM host. This is crucial since the host, which provides virtio devices, cannot access VM memory by default. KrustVM follows HypSec's design and relies on the KVM host to provide interrupt virtualization.

KrustVM builds on the VM secure boot mechanism proposed by HypSec that leverages public key cryptography to verify the signature of the signed VM images. KrustVM ensures that only trusted VM boot images can run in the VM. Rcore allows the host Linux to make the `VM CREATE` and `VM BOOT` hypercalls to create a new VM and load boot images, such as the kernel binary. Rcore uses Ed25519 from the integrated HACL* library to authenticate the integrity of the boot images. The VM's public key is securely loaded into Rcore's private memory before VM boot, so Rcore can use the key to verify against the respective VM image signature. Rcore stops VM boot if the verification fails. We enlightened QEMU to use KrustVM's secure VM boot API.

Rcore exposes hypercalls to KVM host for exporting encrypted VM data. The KVM host utilizes these hypercalls to implement features such as swapping VM memory to disk or VM management functions, including VM snapshots and migration.

6.1.2 Rcore Area Protection. We modified KVM to allocate a physically contiguous memory region as the Rcore Area. We allocated a reserved memblock [30] during boot time to correspond to the Rcore Area. This prevents the host Linux kernel from allocating free pages from this area. We followed HypSec's design and rely on the host Linux to install Rcore to EL2 at boot time. After Rcore is installed, it subsequently deprivileges the host Linux kernel and activates stage 2 page tables to restrict the host's memory access. It ensures the Rcore Area is unmapped from the host's stage 2 page table.

6.1.3 Hardware Accesses. Rcore encapsulates the functionality of KVM's lowvisor to world switch CPU states. As mentioned in Section 4, Rcore multiplexes the execution context of the host kernel and guest on the Arm hardware. The context comprises CPU states, including the general purpose and Arm's EL1 system registers. Rcore employs Rust's syntax to wrap the inline assembly code used for world switching the EL1 system registers. Rcore's other functionality that requires inline assembly, including the primitives enforcing mutual exclusion for `KMutex` and TLB/cache invalidation, were also wrapped in Rust's inline assembly syntax.

6.2 Integrating Rcore with KVM

Efforts were made to fulfill the goal of running our custom Rust code in EL2. Firstly, LLVM requires basic functions, such as `memcpy` and `memset` even in a freestanding environment [62]. Since Rcore minimized codebase does not contain Linux's implementations of these functions, we manually implemented these functions and redirected ELF symbols to our implementations. Furthermore, like KVM, we link code to be executed in EL2 in a separate ELF section. This was done by annotating functions through Rust's attribute `link_section` to locate them in the right section.

Finally, Rcore builds on top of KVM's existing support to provide access to Per-CPU variables to software running in EL2. Rcore creates the metadata called `vCPU context` for every VCPU of a given VM. Rcore uses a Per-CPU variable to track the scheduling status to ensure the same VCPU cannot simultaneously run on two different CPU cores. KVM allocates a memory buffer to store all per-CPU variables. Similarly, Rcore stores a unique offset to each core's `TPIDR_EL2` register. We create a helper class to assist Rcore to access them correctly. We successfully encapsulated three `unsafe` statements into a higher-level primitive. This includes reading the address of the extern static variable first, then reading `TPIDR_EL2` via inline assembly, and lastly, dereferencing the calculated address. Regarding memory safety, `TPIDR_EL2`'s value for each CPU is not modified after boot time initialization. Hence, no out-of-bound accesses can happen because the address for each Per-CPU variable remains static.

6.3 Compiling KrustVM

As Linux 5.15 does not have built-in Rust support, we implemented Rcore in a single crate on the `no_std` environment and compiled it into a single static library. Our implementation is compatible with the Linux kernel codebase. For example, the page size definition is identical in Rcore and KVM. Also, types like `kvm_vcpu` are shared between Linux and Rcore. These type definitions were generated automatically with the tool `bindgen` [11]. For constants that are used by both Linux and Rcore, we copied them from C to Rust manually. Due to the limited support of macros in `bindgen` and the heavy usage in Linux, we did not use it to generate constants. Regarding alignment, field layout order, and padding of custom types, Rust provides an attribute `#[repr(C)]` that ensures the data layout of the marked type has the same layout as in C.

6.4 Contributing the Community

During our implementation, we found that Rust and LLVM lack some ARMv8-specific features. Specifically, we encountered compilation errors when attempting to access specific system registers in the default Rust configuration. However,

Table 2: Application Benchmarks

Name	Description
Kernbench	Compilation of the Linux 6.0 kernel using `tinyconfig` for Arm with GCC 9.4.0.
Hackbench	`hackbench` [52] using Unix domain sockets and 50 process groups running in 50 loops.
Netperf	`netperf` [29] v2.6.0 running the netserver on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_MAERTS (throughput), and TCP_RR (transmission rate).
Apache	`Apache` v2.4.41 Web server running `ApacheBench` [64] v2.3 on the remote client, which measures the number of handled requests per second (throughput) when serving the 41 KB index.html file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.5.22 using the `memtier` [50] benchmark v1.2.3 with its default parameters. Measures operations per second (throughput).
YCSB-Redis	`redis` v7.0.11 using the `YCSB` [13] benchmark v0.17.0 with its default parameters. Measures operations per second (throughput).

these features are crucial for a hypervisor like KrustVM that programs low-level hardware states. To mitigate this issue, we took the initiative to contribute patches [71, 72] to both LLVM and Rust rather than relying on workarounds. We believe the contribution benefits the community, facilitating Rust's adoption in Arm-based systems.

7 EVALUATION

We evaluated the performance of KrustVM, HypSec-5.15, and mainline KVM. The bare metal environment is also tested using the same benchmarks to establish a baseline reference. All workloads were run on the Raspberry Pi 4 Model B development board, with a Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC at 1.5GHz, 4GB of RAM, and a 1 GbE NIC device.

KrustVM, HypSec-5.15, and the mainline KVM are based on Linux 5.15. QEMU v4.0.0 was used to start the virtual machines on Ubuntu 20.04. The guest kernels also used Linux 5.15, and all kernels tested employed the same configuration. We used the enlightened QEMU (see Section 6) to run VMs on KrustVM and HypSec-5.15. For KVM, we used the mainline QEMU 4.0.0. We applied the same patch from HypSec to the guest Linux kernel to enable virtio. `rustc` version 1.68.0-nightly was used to compile Rcore, while clang 15.0.0 was used to compile the remaining components of KrustVM, the mainline KVM, and the guest kernels.

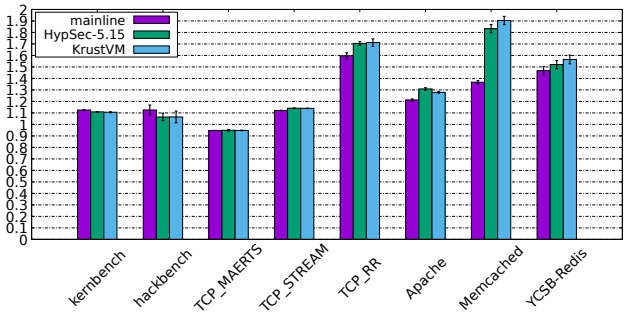
7.1 Performance Evaluation

Application Performance. We ran the benchmarks listed in Table 2 in a single VM running on KrustVM, HypSec-5.15 and the mainline KVM, and bare-metal. We configured the hardware with two physical CPUs and 1 GB of RAM for the bare metal setup, for testing KrustVM, HypSec-5.15 and mainline KVM, each VM is equipped with two virtual CPUs and 1 GB RAM, and the hypervisor runs on the full hardware available. For client-server experiments, servers were run on the Raspberry Pi, while the clients were run on a separate x86 Linux machine with 8 CPU cores and 16 GB RAM, with a 1 GbE cable connecting the two. Vhost-net is enabled for all VMs. Figure 4 shows the results of VM performance normalized against bare-metal performance. 1.00 refers to no virtualization overhead (i.e., the same as bare-metal). A higher value means higher overhead. The performance of these real application workloads show modest overhead overall for KrustVM compared to mainline KVM and HypSec-5.15.

In the `TCP_MAERTS` benchmark, it can be observed that mainline KVM, HypSec-5.15 and KrustVM all outperformed the bare-metal setup. The benchmark evaluates the TCP send throughput on the server. We suspect this is because the virtio driver in the guest kernel batches multiple packet sends before submitting the data to the backend virtio driver in the hypervisor host, which transfers the batched data to the NIC. In contrast, the bare-metal driver transmits packet data to the NIC for each transmission, leading to a higher overhead.

KrustVM showed modest overhead compared to KVM. In most cases, the overhead is within 10%. KrustVM suffered higher overhead in networking benchmarks (i.e., Netperf, Apache, Memchached, YCSB-Redis) that are more complex. KrustVM does not allow the host kernel to access the I/O data stored in VM memory by default. Therefore, to enable virtio, we follow HypSec [36]'s approach and modified the guest kernel to make the `grant` and `revoke` hypercalls to Rcore to share/unshare memory with the host explicitly for each I/O transaction. VMs on KrustVM and HypSec-5.15 suffered from additional hypercall overhead while running network workloads. This effect can be observed to be larger for TCP_RR and Memcached since these two benchmarks are more I/O intensive compared to other networking benchmarks, making the latency caused by `grant` and `revoke` hypercalls more noticeable. Future optimizations, such as batching `grant` and `revoke` hypervisor during I/O transactions, could be incorporated to enhance performance.

Multi-VM Performance. We evaluated the performance of workloads running in multiple concurrent VMs. We used three configurations for each hypervisor setting. We ran Hackbench in 1 VM and concurrently in 2 and 4 VMs. In the multi-VM setups, we started Hackbench in all VMs simultaneously. Because the Raspberry Pi 4 model B hardware has only 4 GB

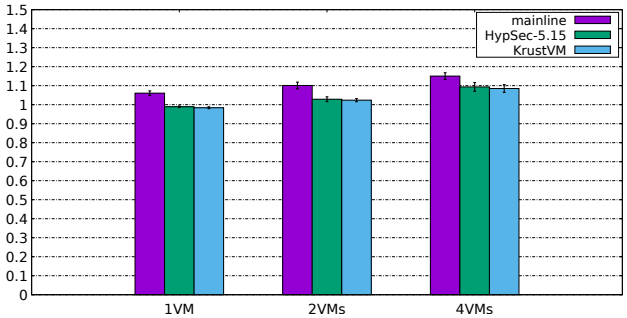


(results are normalized to bare-metal performance)

Figure 4: Application Benchmark Performance.

of RAM to spare, we allocated 512 MB of RAM to each VM. Each VM is configured with 1 VCPU. The number of process groups started by Hackbench is also lowered from 50 to 20 in response to the more resource-constrained VM setups. The results are shown in Figure 5. Similarly, we normalized the results to bare-metal performance using the same number of CPUs and RAM. 1.00 refers to no virtualization overhead. As expected, overhead gradually increased for all hypervisors as the number of VMs running in parallel increased. Hackbench causes memory pressure. It forks multiple processes to perform IPCs. Concurrent VMs running Hackbench enlarge the memory footprint, resulting in stage 2 page faults due to more VM access to unmapped gPAs, thus suffering from higher page fault handling overhead.

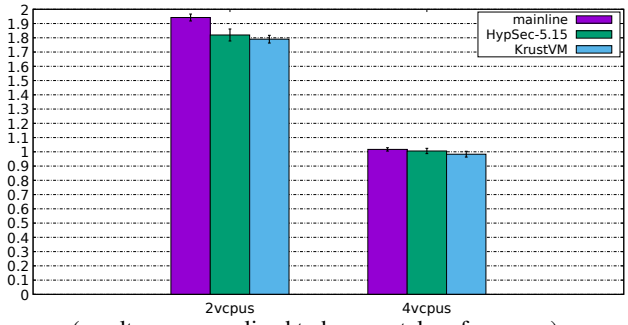
In our evaluation, Hackbench ran slower on KVM compared to HypSec and KrustVM. We suspect this is because HypSec and KrustVM employ a more simplistic counter-based algorithm than KVM to track page table allocation status, thus causing less performance overhead compared to KVM, which relies on Linux's memory allocators to allocate page tables.



(results are normalized to bare-metal performance)

Figure 5: Multi-VM Hackbench Performance.

Scaling VCPUs. We tested Hackbench on VMs with a higher number of VCPUs: 2 and 4 VCPUs, on each hypervisor configuration. Each VM uses 1 GB of RAM. The results are shown in Figure 6. The results are normalized to bare-metal performance using 4 physical CPUs and 1 GB of RAM. The setups with 2 VCPUs exhibit virtualization overhead close to 100% because they only had half the number of CPUs to run the benchmark. All three hypervisors demonstrated their ability to scale from 2 to 4 VCPUs, each running around twice as fast when assigned with 4 VCPUs.



(results are normalized to bare-metal performance)

Figure 6: Multi-VCPU Hackbench Performance.

7.2 Safety Analysis

Our goal is to ensure the Rcore implementation is data race-free and deadlock-free. We elucidate below how the synergistic utilization of `KMutex` and RGFs help to achieve these two properties.

Data race freedom. We ensure that every access or mutation to shared metadata in Rcore is guarded by `KMutex` to eliminate data races. Rust permits unsafe reads or writes to shared global data. We ensure unsafe shared data accesses do not exist in Rcore. All shared data reads or writes must be conducted via `KMutex`. Rcore is required to use the `lock` method to access the wrapped shared data protected by `KMutex`. Our implementation provides no safe Rust alternatives for accessing shared data to Rcore. As mentioned earlier, `lock` invokes the Arm assembly to impose mutual exclusion. Data race freedom of stage 2 page table accesses is guaranteed by making sure that Rcore only accesses stage 2 page tables within `S2PTInfo`'s methods, thereby protecting the stage 2 page table metadata and the page tables themselves with the same lock. Additionally, Rcore does not provide any way for VMs to share stage 2 page tables, therefore chances for inter-VM stage 2 page table data races are also eliminated.

Deadlock freedom. We identify two causes that could result in a deadlock scenario: *multiple same lock acquisition* and *incorrect lock acquire & release ordering*. We discuss how Rcore prevents the scenarios to deliver deadlock freedom.

Multiple same lock acquisition: `KMutex`'s `lock` method requires the caller to pass a mutable reference. Rust prohibits the creation of multiple mutable references from the same

object while any of them is still in use. Consequently, this design empowers the Rust compiler to detect and reject self-deadlocking scenarios when utilizing `KMutex`.

Incorrect lock acquire & release ordering: We ensure that Rcore always follows the pre-defined partial order to acquire and release `KMutex`es. We ensure the acquisition order of `KMutex`es always follows the desired partial order. The acquired `KMutex`es are automatically released by the corresponding `Drop` trait in the reverse of the acquisition order, thus ensuring `KMutex`es are released accordingly to the desired order.

A lock order inversion is impossible due to two reasons. First, we implemented every `CanGet*` trait based on the partial order between locks. If a given order indicates that Rcore should not acquire `KMutex<B>` after acquiring `KMutex<A>`, the type `A` must not implement the `CanGetB` trait. Assume Rcore acquires `KMutex<A>` and gets a mutable reference to type `A`, `&mut A`. Since the partial order is strictly followed, type `A` does not implement `CanGetB`. Therefore, Rust forbids passing `&mut A` to `B`'s RGF, preventing such lock inversion.

Second, we ensure that, for a single CPU, at most, one `KMutex` can be active at any time. Two factors (**F1** and **F2**) uphold this guarantee. **F1:** `KMutex` does not have a constructor, and neither do any types that implement any of the `CanGet*` traits. **F2:** as shown in Listing 3, the design of any RGFs for `T` ensures that a passed-in mutable reference to an object that implements a `CanGet*` cannot be used until the returned `&mut KMutex<T>` gets dropped. The only way to construct a type with any of the `CanGet*` traits implemented is via the `unsafe` constructor of `LEntry`, which have manually ensured is called only once at the entry point of Rcore.

A given type may implement multiple different `CanGet*` traits. For instance, a type `B` may implement `CanGetA` and `CanGetC` traits, where `A` is different from `C`. Assume a mutable reference to `B` (`b`) is passed to the RGF of type `A` to get a `&mut KMutex<A>` (`ref_a`). **F2** ensures that `b` cannot be active before `ref_a`'s lifetime ends. The example shown in line 19 of Listing 3 passes `b` to type `C`'s RGF. Rust prohibits the usage since it requires `b` to be active before the passing.

Preserving Rust's Safety Requirements To prevent data races, Rust's ownership model forbids multiple mutable references to the same object to coexist simultaneously if no mechanism is used to synchronize these reads and writes to the object. We explain why the design of `KMutex` and RGF preserves Rust's safety requirements.

We consider two aspects: the data protected by `KMutex` and mutable references to `KMutex` (`&mut KMutex`). For the former, `KMutex`'s lock method enforces mutual exclusion to synchronize and protect concurrent data accesses. For the latter, we show that no modification can be made to `KMutex` itself. We next explain why such a condition is adequate through the Tree Borrows (TB) [66] aliasing model. First, `KMutex` exposes no methods that can mutate itself. Second, we cannot replace an existing `KMutex` with a newly created one since `KMutex` does not provide a constructor; this prevents a locked object from being replaced. Finally, only one `KMutex` object can be active for a single CPU, so we cannot swap two existing `KMutex`s as swapping requires both mutable references to be active.

Based on the TB model, the creation of mutable references across different CPUs is treated as foreign read accesses in our case. Upon creation, mutable references initially exist in the Reserved state, allowing both foreign and child reads without restriction. Even if a reference transits to the Frozen state due to a protector, these reads remain permissible at any time. Consequently, the creation and reading of multiple mutable references, without modification, are consistently valid operations.

We employed the TB version Miri [39] to analyze our code and validate our approach. Given Miri's limitation in directly verifying multi-CPU programs, we simulated a multi-CPU environment by spawning two threads. This simulation enabled us to evaluate the robustness of our approach in a concurrent execution context. Miri revealed no UB.

Additionally, compiler optimizations would not break any of our security foundations. First, the borrow checker executes before the optimizer so that optimizations won't affect the enforced lock orders. Second, the TB model stipulates that compilers cannot arbitrarily introduce write operations where no explicit writes to a given mutable reference exist in the original code. Consequently, we can guarantee that operations on KMutex mutable references will be limited to read access.

8 DISCUSSION

8.1 Security

KrustVM assume the same threat model as previous work [36]. Thus, KrustVM protects VMs against the CVEs addressed in that work [36]. The Rust compiler cannot check for logical errors, such as those resulting from hardware misconfiguration. On top of Rust's existing security guarantees, such as automatic bound checks for array types, we extended its capability to eliminate data races and deadlocking and isolate memory regions.

The unsafe Rust usage in Rcore, as shown in Section 6, encompasses raw pointer usages safeguarded by Safe Pointers and those for accessing predefined addresses. For the latter, we manually validate the source code to eliminate unintended memory accesses. The unsafe usages also include machine state operations (e.g., TLB flushes and world switches) that require inline assembly, which is orthogonal to memory safety and thus beyond Rust's safety scope. Finally, for unsafe traits

and object creations, as shown in Section 4.2.2, we have carefully examined their usages to ensure they are safe.

8.2 Scalability

In our current implementation, it requires manual effort to implement traits carefully to ensure the validity of the partial order. With more locks involved, the effort may grow significantly. However, it is possible to automate this process with Rust's macro system, as we have clear naming rules for helper traits. For example, we can convert the partial order to a total order and implement helper traits between any two entities accordingly. Since all the acquisition edges are generated automatically, if we correctly implement the macro rule, we can safely assert the order contains no cycle so no deadlock can happen.

Further, though KrustVM currently allocates `KMutex` instances statically, incorporating dynamic allocation is possible. We could extend RGFs to support dynamic allocation, reuse its interface, and avoid high complexity. Specifically, we can adapt RGFs to first check the existence of the required object and allocate it if it is not present; otherwise, return the object directly. This would not break `KMutex`'s soundness. Since the allocation happened in RGFs, we must provide `CanGet*` to the RGF to get the newly allocated objects. This means that at any time, we can only hold at most one mutable reference to a `KMutex` in a single CPU. Hence, even if the object is dynamically allocated, we still do not have any means to mutate it or use it to mutate other objects (e.g., swap to `KMutex`).

KrustVM's current implementation partitions the address space into distinct regions. We employ Safe Pointer to constrain memory access to these regions. The bound checks enforced when constructing a Safe Pointer can be extended to support finer granularity. For instance, we could further split the Page Table Pool into multiple regions, each serving as a pool for a single page table level (e.g., page directory, page middle directory, etc.), then represent each region with a Safe Pointer type. Safe Pointer can be generalized to secure raw pointer accesses in Rust to ensure these accesses are confined within pre-defined bounded memory regions. Although raw pointer accesses in Rust can significantly impact memory safety, they are rarely used [8]. The cost of Safe Pointer exists only during construction but not on later memory accesses. Therefore, utilizing Safe Pointer incurs minimal overhead while enhancing security.

8.3 Portability

`KMutex` and safe pointer types are generalizable to other low-level systems, including Linux components. `KMutex` provides a reusable, architecture-agnostic interface for wrapping shared objects. The main challenge lies in encoding lock order into Rust's type system, which requires careful analysis of lock relationships. This involves constructing a call graph of lock usage sites, which can be automatically translated into Rust traits and implementations to facilitate implementing RGFs. For safe pointers, raw pointer accesses are typically region-confined (e.g., Linux's slabs [44]) in low-level systems. Safe pointers offer a reusable interface that simplifies pointer integrity validation compared to direct address handling or reference conversion. Developers can adjust bounds checks based on system-specific requirements, enhancing memory safety for low-level systems dealing with raw pointers.

Porting KrustVM to an x86-64-based platform is also possible. Overall, our extensions to Rust are architectural-independent. We believe that the extensions can be ported to x86-based with modest efforts. Developers only need to replace the Arm-based synchronization primitive used by the `lock` method for `KMutex` with an x86-64-based implementation, such as an x86-based spinlock. Furthermore, the hypervisor design [36] that we adopted in this work is compatible with an x86-64 environment. Like Armv8, x86-64 provides similar virtualization features, including nested paging (e.g., NPT/EPTs) and hardware-based control structures (e.g., VMCS) to deprivilege the host Linux and interpose VM exits. We expect an x86-64-based implementation of KrustVM to leverage these features.

8.4 Verification

The Kani Rust Model Checker [65] is an open-source tool to verify the Rust code. Kani operates as a Rust compiler backend, exhibiting decent execution performances compared to other Rust-based model-checking tools. Kani's efficiency stems from the fact that it utilizes Rust code's type information within Mid-level Intermediate Representation (MIR), a radically simplified form of Rust that is used widely inside the Rust compiler, to conduct bit-precise model checking, enabling itself to prune invalid branches but maintain the correctness of the verification by examining all possible values.

In this work, we have explored the integration of Kani into our workflow. We have formally verified the correctness of the bound condition embedded in custom Safe Pointer types in KrustVM. In other words, we have verified that the implementation delivers the intended behavior to enforce bound checks for three custom types, `GenericPhysRegion`, `PTEAddr`, and `SMMURegion` In future work, we intend to explore Kani's full capability in verifying KrustVM's unsafe Rust code blocks.

9 RELATED WORK

VM Protection. Various previous work [26, 35, 37, 45, 70, 74] redesigned the hypervisor to protect VMs. Unlike our work, none of them used Rust to secure their hypervisor implementation. KrustVM and SeKVM [37] both leveraged

an earlier design [36] to retrofit and secure KVM, providing the same level of VM protection. SeKVM included a formally verified core to protect VMs against an untrusted host Linux kernel, while KrustVM relies on a Rust-based Rcore to protect VMs. Formal verification of the concurrent C-based SeKVM core requires significant effort. The authors took two person-years to complete the correctness and security proofs. In contrast, our Rust-based implementation took less than one person-year while ensuring properties verified systems provide, including memory safety, data race, and deadlock freedom. Arm's Confidential Compute Architecture (CCA) [7] provides similar VM protection guarantees to KrustVM. To support CCA, Arm introduced a new Realm world to run protected VMs, and secure firmware called the Realm Management Monitor (RMM); the latter controls the CCA hardware to prevent a compromised hypervisor from tampering with VM safety. To our knowledge, no Arm hardware has been implemented in CCA. KrustVM utilizes Arm's VE features available on the current Arm hardware without CCA to host protected VMs.

KrustVM is compatible with future CCA-featuring hardware. KrustVM could be extended to run simultaneously with CCA and offer an alternative approach for hosting protected VMs. On CCA, the RMM running on the EL2 mode of the Realm world hosts VMs and protects them by interposing their enters/exits. Like KrustVM, the RMM could transfer control to the hypervisor on a VM exit to use its functionality. VM enters and exits to the hypervisor (which runs in the normal world) on CCA involves more costly world/mode switches (KVM (normal EL1) ↔ EL3[1] ↔ RMM (realm EL2) ↔ VM (realm EL1/EL0)) than KrustVM, which requires a simpler path that is exclusively in the normal world (KVM (normal EL1) ↔ Rcore (normal EL2) ↔ VM (normal EL1/EL0)). Although both CCA and KrustVM offer similar VM protection, KrustVM's efficient VM/hypervisor transitions could result in better performance.

Rust-based Systems. Recent work extended existing C/C++ systems with a Rust binding to enable a Rust-based programming environment. Rust-SGX [68] and RusTEE [67] wrapped the C/C++ TEE SDK and exposed a safe Rust API to enable Rust programming in TEE environments such as SGX and TrustZone. Similarly, the Rust-for-Linux [19] project added abstraction layers to the Linux kernel to facilitate Rust driver programming with Rust. Besides building a Rust binding, previous work re-implemented C-based components in virtualization systems with Rust. rust-vmm [55] rewrote a subset of QEMU's functionalities and separated them into libraries in Rust crates. Firecracker [2], crosvm [21], Cloud Hypervisor [43], and VMSH [63] extended the rust-vmm project with extra functionalities. These previous works built on top of existing core systems. In contrast, our work retrofitted Linux/KVM with a Rust-based TCB. HyperEnclave [28] relies on a Rust-based security monitor to enforce isolation between enclave TEEs. Unlike our work, the authors did not discuss the Rust monitor's implementation and its unsafe Rust usage.

Others [10, 12, 34, 48, 73] took a clean-slate approach to build an OS from scratch. Similar to our work, they aimed to limit unsafe Rust usage in their codebase and separated unsafe Rust in a small trusted component from the safe Rust codebase [10, 34, 48]. Our work goes beyond limiting and confining unsafe Rust and guarantees memory and lock safety.

Synchronization Primitives. Message-passing-based synchronization enables concurrent executions to coordinate and synchronize through sending and receiving messages. Rust's standard library provides a message-passing API. Rust's type system allows developers to employ session types [24, 25] to enforce specified message-passing protocols. Past studies proposed customized session types to provide varying degrees of expressiveness. This includes but is not limited to basic binary session type [27], shared sessions [15], asynchronous message reordering [17], and exception handling [32, 33]. KrustVM adopts shared-state synchronization instead of message-passing for several reasons. First, while message-passing-based synchronization is prevalent in microkernel-based hypervisors [23, 59], Linux/KVM leverages shared-state synchronization. We extend this familiar mechanism, aiming to implement safe locking strategies that ensure data race freedom and maintain system availability. This approach preserves the performance efficiency inherent to shared-state synchronization. Unlike message-passing, which can incur significant overhead due to cross-core message exchange among concurrent executions, our locking-based approach facilitates efficient information exchange among multiple cores via shared memory regions. Moreover, our goal is to maintain Rcore's simplicity while providing essential VM protection functionalities. Message-passing support, such as Rust's existing message-passing APIs, typically relies on OS features like threading abstraction unavailable in Rcore's minimalist environment. Integrating message-passing synchronization into Rcore could unnecessarily complicate its codebase.

Other works [47, 58] have also implemented deadlock-free mechanisms for Rust. However, unlike our approach, these mechanisms do not support nested locking. On the other hand, [57, 61] ensure deadlock freedom for Rust while supporting nested locking. However, they rely on runtime resolution checks, which can introduce significant overhead during execution. In contrast, our approach that combines `KMutex` and RGFs adopts a static approach to eliminate deadlocks at compile time.

[1] CCA integrates a secure monitor running in Arm's EL3 mode to mediate switches between Realm and normal world.

10 CONCLUSIONS

We have presented KrustVM, a secure multiprocessor KVM hypervisor that integrates a Rust-based Rcore to protect VM confidentiality and integrity against a compromised host Linux kernel. We exploited Rust's unique safety features, including the type system and lifetimes, and extended them in a novel way to ensure that Rcore's codebase is data race and deadlock-free. Additionally, we introduced the Safe Pointer type to secure Rcore's memory accesses. KrustVM accomplishes substantial security enhancements over KVM while preserving KVM's functionality and performance efficiency.

ACKNOWLEDGMENTS

We thank our shepherd, Daniel Bittman, and other anonymous reviewers for your insightful feedback. This research was supported by the National Science and Technology Council of Taiwan under research grants 111-2218-E-002-015-MBK and 112-2634-F-002-001-MBK.

REFERENCES

[1] Advanced Micro Devices. 2018. Secure Encrypted Virtualization API Version 0.16. https://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Amazon Web Services, Inc. 2018. Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances/.

[4] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 81–89.

[5] ARM Ltd. 2013. ARM Architecture Reference Manual ARMv8-A DDI0487A.a.

[6] ARM Ltd. 2016. ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf.

[7] ARM Ltd. 2022. Introducing Arm Confidential Compute Architecture Version 1. https://developer.arm.com/documentation/den0125/0100/What-is-Arm-CCA-.

[8] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. https://doi.org/10.1145/3428204

[9] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

[10] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 295–312. https://www.usenix.org/conference/osdi21/presentation/bhardwaj

[11] bindgen maintainer. 2023. bindgen. https://github.com/rust-lang/rust-bindgen.

[12] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[13] Brian Cooper. 2021. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[14] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Security and Performance in the Delegated User-level Virtualization. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 209–226. https://www.usenix.org/conference/osdi23/presentation/chen

[15] Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. 2022. Ferrite: A Judgmental Embedding of Session Types in Rust. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.22

[16] Columbia University. 2021. SOSP 21: Artifact Evaluation: Verifying a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. https://github.com/VeriGu/sosp-paper211-ae.

[17] Zak Cutner, Nobuko Yoshida, and Martin Vassor. 2022. Deadlock-free asynchronous message reordering in rust with multiparty session types. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 246–261. https://doi.org/10.1145/3503221.3508404

[18] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 333–348. https://doi.org/10.1145/2541940.2541946

[19] Rust for Linux Team. 2023. Rust for Linux. https://rust-for-linux.com/.

[20] Andrea Gallo. 2021. Software Defined Vehicles and the need for standardization. https://static.linaro.org/assets/automotive_white_paper_0921.pdf

[21] Google. 2023. ChromiumOS Virtual Machine Monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm/.

[22] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (May 2009), 91–98. https://doi.org/10.1145/1506409.1506429

[23] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-pacific Workshop on Workshop on Systems (APSys 2010)*. New Delhi, India, 19–24.

[24] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

[25] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[26] Jake Edge. 2020. KVM for Android. https://lwn.net/Articles/836693/.

[27] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2808098.2808100

[28] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. 2022. HyperEnclave: An Open and Cross-platform Trusted Execution Environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 437–454. https://www.usenix.org/conference/atc22/presentation/jia-yuekai

[29] Rick Jones. 2018. Netperf. https://github.com/HewlettPackard/netperf.

[30] The kernel development community. 2023. Boot time memory management. https://docs.kernel.org/core-api/boot-time-mm.html.

[31] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, Vol. 1. Ottawa, ON, Canada, 225–230.

[32] Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science* 304 (Sept. 2019), 48–60. https://doi.org/10.4204/eptcs.304.4

[33] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.4

[34] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[35] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 638–654. https://doi.org/10.1145/3477132.3483554

[36] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1357–1374.

[37] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

[38] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 465–484. https://www.usenix.org/conference/osdi22/presentation/li

[39] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541

[40] Linus Torvalds. 2021. Linux Kernel Mailing List. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=79db4d2293eba2ce6265a341bedf6caecad5eeb3.

[41] Bob Lord. 2023. The Urgent Need for Memory Safety in Software Products. https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products.

[42] Arm Ltd. 2023. Exclusive access instructions. https://developer.arm.com/documentation/100934/0100/Exclusive-access-instructions.

[43] Cloud Hypervisor maintainers. 2023. Cloud Hypervisor - Run Cloud Virtual Machines Securely and Efficiently. https://www.cloudhypervisor.org/.

[44] Mel Gorman. 2007. Slab Allocator. https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[45] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2020. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1695–1712. https://www.usenix.org/conference/usenixsecurity20/presentation/mi

[46] Microsoft. 2016. Hyper-V Technology Overview. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[47] Mike White. 2024. HappyLock: Deadlock Free Mutexes. https://crates.io/crates/happylock.

[48] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[49] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[50] Redis Labs. 2015. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[51] Reuters. 2018. Cloud companies consider Intel rivals after the discovery of microchip security flaws. https://www.cnbc.com/2018/01/10/cloud-companies-consider-intel-rivals-after-security-flaws-found.html.

[52] Rusty Russell. 2008. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[53] Rusty Russell. 2008. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review* 42, 5 (July 2008), 95–103.

[54] Rust-book. 2023. Fearless Concurrency. https://doc.rust-lang.org/book/ch16-00-concurrency.html

[55] rust-vmm maintainers. 2023. rust-vmm. https://github.com/rust-vmm.

[56] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (Nov. 1984), 277–288.

[57] Shelby Doolittle. 2021. cooptex. https://crates.io/crates/cooptex.

[58] Stefan Mack. 2024. Deadlocker. https://crates.io/crates/deadlocker.

[59] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th*

European Conference on Computer Systems (EuroSys 2010). Paris, France, 209–222.

[60] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3381052.3381326

[61] Tanishq Jain. 2023. JThread-rs - deadlock-free mutex lock library. https://crates.io/crates/jthread.

[62] The Clang Team. 2024. clang - the Clang C, C++, and Objective-C compiler, Description: -ffreestanding. https://clang.llvm.org/docs/CommandGuide/clang.html#cmdoption-ffreestanding

[63] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. 2022. VMSH: Hypervisor-Agnostic Guest Overlays for VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 678–696. https://doi.org/10.1145/3492321.3519589

[64] The Apache Software Foundation. 2015. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html.

[65] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/3510457.3513031

[66] Neven Villani. 2023. Tree Borrows. https://github.com/Vanille-N/tree-borrows/blob/master/full/main.pdf.

[67] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 442–453. https://doi.org/10.1145/3427228.3427262

[68] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2333–2350. https://doi.org/10.1145/3319535.3354241

[69] Chris Williams. 2017. Microsoft: Can't wait for ARM to power MOST of our cloud data centers! Take that, Intel! Ha! Ha! https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup/.

[70] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 441–453. https://doi.org/10.1109/HPCA.2018.00045

[71] Yu-Hsun (Tommy) Chiang. 2022. [MC][AArch64] Enable '+v8a' when nothing specified for MCSubtargetInfo. https://github.com/llvm/llvm-project/commit/4a31af88a26726f4662a2923618fe45977d09356.

[72] Yu-Hsun (Tommy) Chiang. 2022. v8a as default aarch64 target. https://github.com/rust-lang/rust/commit/382dba52ee0c6142d9a3774d735962797c043fab.

[73] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. San Diego, CA.

[74] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/2043556.2043576

[75] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1789–1806. https://doi.org/10.1145/3133956.3134043