

國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

章瑋麟

Wei-Lin Chang

指導教授：黎士瑋 博士

Advisor: Shih-Wei Li Ph.D.

中華民國 112 年 7 月

July, 2023

國立臺灣大學碩士學位論文

口試委員會審定書



本論文係章瑋麟君（R09922117）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 1 日承下列考試委員審查通過及口試及格，特此證明

口試委員：_____

（指導教授）

_____	_____
_____	_____
_____	_____
_____	_____

所 長：_____





Acknowledgements





摘要

中文摘要

關鍵字：LaTeX、中文、論文、模板





Abstract

Hypervisors are extensively utilized in cloud computing settings as they manage hardware resources for virtual machines, making their security a critical concern. An attacker that exploits vulnerabilities in the privileged hypervisor codebase can gain unfettered access to VM data, compromising their safety. Previous attempts to retrofit hypervisors into small trusted cores have limitations, as the security still relies on the implementation of the trusted core. Recently, Rust adoption has been increasing for its strong memory safety guarantees and performance efficiency. Leveraging Rust, our work focuses on rewriting SeKVM, a secure Linux KVM hypervisor, into KrustVM, the first Rust-based secure Linux/KVM hypervisor. KrustVM incorporates Rcore, a small trusted core, to protect VM confidentiality and integrity. We addressed challenges in incorporating Rust TCB into Linux, bringing up KrustVM on real hardware, and rewriting SeKVM's TCB in Rust. Additionally, we minimized unsafe Rust usage, enclosed unsafe code within safe abstractions, and utilized Rust's type system to ensure spatial memory safety. Our

implementation of KrustVM suggests a modest overhead compared to mainline KVM and SeKVM. Our work demonstrates the practicality of securing existing hypervisors through a C-to-Rust rewrite.



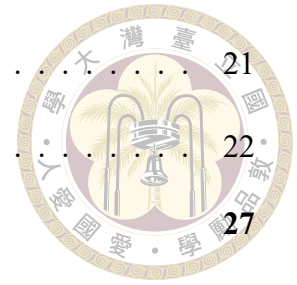
Keywords: LaTeX, CJK, Thesis, Template



Contents

	Page
Verification Letter from the Oral Examination Committee	i
Acknowledgements	iii
摘要	v
Abstract	vii
Contents	ix
List of Figures	xi
List of Tables	xiii
Chapter 1 Introduction	1
Chapter 2 Background	5
2.1 SeKVM	5
2.2 The Rust Programming Language	7
Chapter 3 KrustVM: An SeKVM Rust Rewrite	15
3.1 Integrating Rust and Linux	16
3.2 Bringing up KrustVM on Real Hardware	16
3.3 Rewriting C-based Kcore into Rust-based Rcore	18
Chapter 4 Securing Rcore Memory Accesses	19
4.1 Rcore Memory Regions	19

4.2	Rcore Metadata and Reference Getter Functions	21
4.3	Memory Region Isolation	22
Chapter 5	Evaluation	27
Chapter 6	Conclusions	31
References		33





List of Figures

Figure 3.1	Kcore overlaps the unusable hole on Rpi-4B	17
Figure 3.2	Overlap prevention	18
Figure 4.1	Memory Regions	20
Figure 5.1	Application Benchmark Performance	29





List of Tables

Table 4.1	Rcore metadata	20
Table 5.1	Application Benchmarks	28



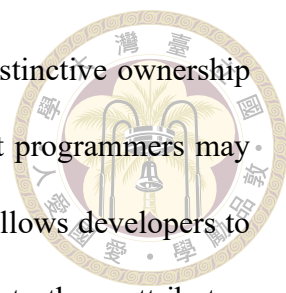


Chapter 1 Introduction

Hypervisors are essential to cloud computing. They manage the hardware resources to provide the virtual machine (VMs) abstraction and host these VMs in the cloud. The widely used commodity hypervisors, such as KVM [10] or Hyper-V [14], include a large and complex TCB to satisfy users' requirements in performance and functionality. These hypervisors were written in unsafe languages like C, making them vulnerable to safety bugs, such as out-of-bound memory access and use-after-free. For example, KVM integrates an entire Linux OS kernel inside its TCB. Attackers that successfully exploit hypervisor vulnerabilities may gain the ability to steal or modify secret VM data.

Previous work [12] has retrofitted commodity hypervisors into a small trusted core that enforces resource access control to ensure the confidentiality and integrity of VM data against hypervisor and host operating system exploits. However, the security of the whole system still depends on the implementation of the small trusted TCB. Any vulnerability in the trusted TCB can void the guarantees of VM data confidentiality and integrity. While [13] extended the work of [12] by formally verifying the smaller TCB, the approach is not scalable since all code modifications including the addition of new features, or code refactoring, requires a new proof.

Rust is an emerging programming language that ensures strong memory safety guar-



antees at compile time while offering performance efficiency. Its distinctive ownership and lifetime system effectively addresses potential safety issues that programmers may encounter. Further, similar to programming languages like C, Rust allows developers to directly manage low-level systems resources such as memory. Due to these attributes, various previous work has adopted Rust to implement systems software with critical security and performance requirements, including operating systems [3, 5, 11, 15], hypervisors [7, 19], web browsers [1], and TEEs [21, 22]. There has been recent adoption of Rust in the mainline Linux kernel. However, instead of replacing the existing Linux kernel code written in C with Rust, the current efforts were limited to developing new Rust-based device drivers.

Our work leverages the Rust programming language and rewrite SeKVM [13], a secure Linux KVM hypervisor in Rust, so that the resulting hypervisor benefits from the strong safety guarantees that Rust automatically provides. Our implementation, KrustVM, incorporates a small TCB called Rcore to protect VM confidentiality and integrity against the large and untrusted hypervisor codebase that encompasses KVM's host Linux kernel. We identified and overcame the challenges that arose when trying to incorporate a Rust TCB inside Linux, bring up KrustVM on real hardware, and rewrite SeKVM's TCB in Rust. We also redesigned Rcore to minimize the amount of unsafe Rust, and enclosed the unsafe code within a safe abstraction and exposed a safe API in order to implement complex functionalities in safe Rust, including CPU, memory, VM boot protection, VM exit, and hypercall handlers. Further, Rust's type system is leveraged to ensure spatial memory safety of Rcore's memory accesses by dividing physical memory into multiple disjoint regions and guaranteeing all memory accesses done by Rcore are located in the predefined regions. For the Rcore metadata region, we enforce Rcore to access the shared

metadata via a set of reference getter functions (RGF). Each RGF wraps a given Rcore's raw pointer usage and returns a mutable reference to the associated shared metadata object after the caller acquires the corresponding lock. For the remaining memory regions, we built customized Rust types for each memory region that enforces bound-check to accesses, and mandate that Rcore accesses a memory region via each corresponding type.

KrustVM is the first secure Linux/KVM hypervisor written in Rust. We spent less than one person year rewriting SeKVM into KrustVM. By rewriting a C-based hypervisor to a Rust-based implementation, we shift the responsibility of human auditing to the compiler. This results in safer code and a more straightforward development process. Performance evaluation of KrustVM on real Arm64 hardware shows that KrustVM incurs modest performance overhead to application workloads compared to mainline KVM and SeKVM. We demonstrate the practicality of securing an existing commodity hypervisor by a C-to-Rust rewrite.

The rest of the thesis will be organized as follows. Background will be discussed in chapter 2. The rewriting process of KrustVM and the techniques used are described in chapter 3. chapter 4 presents how we utilize Rust's safety features to design and secure Rcore memory accesses. Evaluation of KrustVM and its comparison with mainline KVM and SeKVM is covered in chapter 5. At last, we conclude the thesis in chapter 6.






Chapter 2 Background

2.1 SeKVM

Our work explores the possibility of rewriting a hypervisor TCB in Rust, we chose to base on the implementation of SeKVM [13], because its artifact [8] is available, and also because it has a reduced TCB compared to mainline KVM, which minimizes the attack surface of the codebase, leading to a more secure hypervisor. SeKVM is a formally verified KVM hypervisor, it leveraged an earlier design [12] to retrofit and secure KVM, it relies on a small TCB called Kcore to protect VM confidentiality and integrity against an untrusted KVM host that encompasses the host Linux kernel integrated with KVM. To reduce the TCB, the design separates access control from resource allocation. Kcore has full access to hardware resources to perform access control to protect VM data. SeKVM assumes VMs employ an end-to-end approach to encrypt I/O data. Since VMs already protect their I/O data, SeKVM focuses on protecting VM data in CPU and memory. The KVM host provides device drivers and complex virtualization features such as resource allocation, scheduling, and VM management. Kcore is responsible for protecting VM data, ensuring VM CPU registers and memory allocated to the VM are inaccessible to the untrusted KVM host.

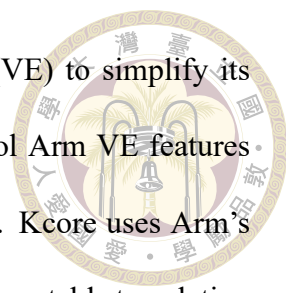
Kcore leverages hardware virtualization support to deprive the KVM host at a



lower privileged level, ensuring the untrusted host cannot disable or control privileged hardware features. Kcore enables the nested page tables (NPT) when running the KVM host and VMs so that they do not have direct access to physical memory. Kcore unmaps its own private memory pages from the respective NPTs, making them inaccessible to VMs and the host. Kcore unmaps a given VM's memory pages from the KVM host's or other VMs' NPTs to isolate these pages. Kcore allocates NPTs for the KVM host and VMs from its memory pool, to which the host and VMs have no access. Since VM and Kcore memory is unmapped from the host NPT, a compromised host that accesses these memory pages causes an NPT fault that traps to Kcore. Kcore routes NPT faults to itself, allowing it to reject invalid host memory accesses.

SeKVM reuses the device drivers from the KVM host to manage I/O devices and provide I/O virtualization. An attacker from the host can control devices to perform Direct Memory Access (DMA) to read or write VM memory. To protect VM memory against such DMA attacks, SeKVM leverages IOMMU to restrict all devices' memory accesses. Kcore allocates and manages IOMMU page tables from its private memory. Kcore trap-and-emulates the KVM host's access to the IOMMU and manages the IOMMU page tables for each DMA-capable device.

Kcore runs in a higher privileged CPU mode than the host to interpose VM exits and interrupts, ensuring the host cannot compromise VM data. VM exit handling may require the KVM host's functionality. Before entering the host, Kcore saves VM CPU registers from the hardware to its private memory then restores the host's CPU registers to the hardware; vice versa is done before entering the VM. An attacker has no access to Kcore's private memory, and thus, the VM CPU registers.



SeKVM leverages Arm's hardware Virtualization Extensions (VE) to simplify its implementation. Kcore runs in the hypervisor mode (EL2) to control Arm VE features to deprive the KVM host in a less privileged kernel mode (EL1). Kcore uses Arm's NPTs, stage 2 page tables to enforce memory access control. Stage 2 page table translation only affects the software running in Arm's kernel and user (EL0) mode, but not software running in EL2. Kcore employs an identity map in the KVM host's stage 2 page tables, translating each host machine's physical addresses (hPA) to an identical hPA. This allows SeKVM to reuse Linux's memory allocator to manage memory implicitly. Running in EL2 allows Kcore to isolate itself from the host and VMs. First, running in EL2 isolates Kcore in a separate address space from EL1 and EL0. Second, Arm provides banked system registers to EL1 and EL2. The host cannot access EL2 registers to disable Kcore's protection. For instance, the host cannot modify the register VTTBR_EL2, which stores the base address of the stage 2 page tables. Kcore exposes a set of required hypercalls [12] for the host to request services that require EL2 privileges. For example, once the host schedules a virtual CPU, it makes a hypercall to Kcore to context switch to the VM. Kcore leverages Arm's IOMMU, the System Memory Management Unit (SMMU) [2], to protect against DMA attacks.

2.2 The Rust Programming Language

Rust, compared to C, is a relatively young programming language aiming to be safe and fast. It enables programs to be memory-safe without requiring programmers to painstakingly manage memory, as in traditional languages (e.g., C/C++). Unlike other memory-safe languages (e.g., Python, Go, etc.), Rust does not leverage garbage collection mechanisms to ensure memory safety. Instead, it introduces the concepts of lifetimes

and ownership to mandate the programmer to follow specific rules. This paradigm of statically enforcing programming rules empowers Rust to perform comparably to C since Rust's compiler has complete control over the code that runs during runtime and can optimize it accordingly. Additionally, Rust's safety rules ensure that no memory safety bugs will be present when satisfied, and the compiler automatically checks and prevents any violation of these rules.

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. In Listing 1, *y*'s lifetime starts at line 5 and ends at line 7 as the block closes. Hence, the `println!` macro is unable to find the value *y*, whose lifetime has already ended. Ownership can be transferred or *moved*. For example, assigning the owning variable to a new variable moves the ownership of the data to the new variable. And passing the variable into a function also moves the data ownership into the function. In both situations, the original variable returns to the uninitialized state, and using it would result in a compilation error.

```
1 // this code sample does *not* compile
2 {
3     let x = 1;
4     {                // create new scope
5         let y;
6         y = x;
7     }                // y is dropped
8
9     // compilation error, y's lifetime has ended
10    println!("The value of 'y' is {}", y);
11 }
```

Listing 1: Rust lifetime example

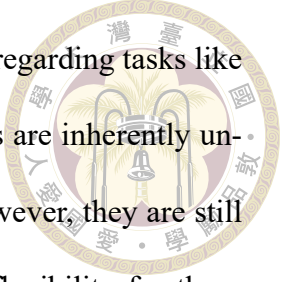
Borrowing. Ownership lacks the flexibility of argument passing. Rust addresses

this by *borrowing*, a mechanism that allows accessing data without gaining ownership. A variable can borrow ownership from another variable to acquire a *reference* to the data. References can be divided into two categories, *shared* references and *exclusive* references. The reference can only be read and not modified with a shared reference. Nevertheless, multiple shared references for a specific value can be held simultaneously. On the other hand, exclusive references allow reading from and modifying the value. However, having any other kind of reference active simultaneously for that value is not permitted.

In summary, Rust's borrowing rule enforces *aliasing xor mutability* meaning there can be multiple shared references or a single exclusive reference. In Listing 2, line 6 would not compile because it tries to create a mutable reference (z) to x, while y already borrowed x immutably. y's lifetime ends on line 8 as it gets used for the last time; therefore z can be created on line 10 and used on line 11. However, if line 13 is uncommented, y's lifetime would be extended to line 13, making the creation of z on line 10 break the borrowing rules.

```
1 {
2   let mut x = vec![1, 2, 3];
3   let y = &x; // immutable borrow of x
4
5   // this line would fail to compile because x is already 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 because x is borrowed mutably by z
15  /* println!("y = {:?}", y); */
16 }
```


Listing 2: Rust enforces *aliasing xor mutability*



unsafe Rust. Rust's safety checks are sometimes too restrictive regarding tasks like low-level hardware access or special optimizations. These operations are inherently unsafe and hence impossible to follow the rules mandated by Rust. However, they are still necessary for low-level software such as hypervisors. To provide flexibility for these operations, Rust allows parts of the program to opt out of its safety checks via the *unsafe* keyword. Traits, functions, and code blocks can be marked as unsafe to disable the checks that the compiler would normally enforce. However, using unsafe code also means that the responsibility for ensuring memory safety is shifted from the compiler to the programmer. Therefore, it is crucial to exercise caution when using unsafe code to avoid introducing bugs or security vulnerabilities.

Interior unsafe. While most low-level code is written in unsafe code, Rust introduces the concept of *interior unsafe* [16]. A function is considered interior unsafe if it exposes a safe interface but contains unsafe blocks in implementation. This allows unsafe operations to be encapsulated into safe abstractions. For instance, in Listing 3, Rust's `replace` function can be called by safe Rust, but it is implemented using unsafe raw pointer operations. At line 6, `ptr::read` is used to copy a bit-wise value from `dest` into `result` without moving it, and at line 7, `ptr::write` overwrites the memory location pointed to by `dest` with the given value `src` without reading or dropping the old value. Lastly, at line 8, `result` is returned to the function's caller.

This leads to a design practice that interior unsafe functions should provide the necessary checks that prevent the unsafe code from producing any undefined behavior or memory safety bugs. The callee in the safe world hence bears no responsibility to ensure safety.



```

1 pub const fn replace<T>(dest: &mut T, src: T) -> T {
2     // SAFETY: We read from `dest` but directly write `src` into it
3     → afterward,
4     // such that the old value is not duplicated. Nothing is dropped and
5     // nothing here can panic.
6     unsafe {
7         let result = ptr::read(dest);
8         ptr::write(dest, src);
9         result
10    }

```

Listing 3: interior unsafe in Rust's replace function

Interior Mutability. Mutating the underlying data via an immutable reference is forbidden in Rust. However, this might be too restrictive for implementing efficient algorithms or data structures. For example, programmers might want to add a cache in a read-only search data structure to optimize the search time. Nevertheless, updating the state of the cache implies the need for mutability, which violates the read-only constraint. Hence, we need the ability to mutate states even under a read-only scene. To address this issue, the Rust standard library provides some special types that can mutate the underlying data even if we only have read-only access to the data holder. This design pattern is known as Interior Mutability. Implementing these types requires `unsafe` operations to bend Rust's usual rules that govern mutation and borrowing. To avoid violating the virtue of the Rust safety assumptions, these types ensure that the borrowing rules, i.e., one mutable borrower at the same time and no mutable borrowers when read-only borrowers exist, will be followed at runtime. If these rules get violated, the implementation of these types has the responsibility to stop the behavior to avoid safety issues. For example, the type `Mutex` in Rust is a type that provides interior mutability. It uses a lock to ensure that only one borrower of the inner data can appear simultaneously to enforce safe mutation of the inner data, even without explicit mutability to `Mutex`. More precisely, when attempting to

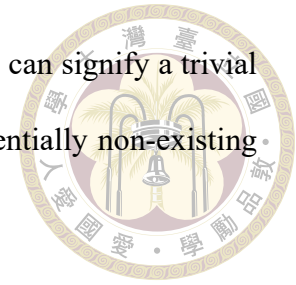
borrow data that has already been borrowed, the `Mutex` enforces a busy wait until the data is returned, thereby allowing only one borrower at a time. However, if a thread borrows the inner data of `Mutex` while it is already borrowing it, `Mutex` will wait forever, i.e., result in a self-deadlock.¹

Generics and Traits. In addition to the safety mechanisms, Rust, as a modern programming language, provides handy features to make programming easier. `Generic` allows code to work with type parameters, reducing the effort of writing similar code for multiple types. For example, using generics, the `Mutex` type can hold and lock any arbitrary type. 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. Additionally, combined with `Generic`, a trait can be treated as a restriction on type specifications such as function arguments or struct fields. The restriction is called a *trait bound*. For example, the `Clone` trait requires the implementing type to provide implementations for its `clone` and `clone_from` functions to make copies of themselves. A `Generic` function or type can use a trait bound to require its type argument to implement `Clone`, so that it can invoke the `clone` function that the argument implements.

Graceful Error Handling. Rust offers a graceful approach to error handling. The `Result<T, E>` and `Option<T>` types in Rust explicitly admit the possibility of errors. `Result` represents the outcome as either `Ok(T)` or `Err(E)`, with `T` denoting the desired result and `E` representing the error reason. Programmers are obligated to handle `E` when accessing `T`, and not doing so would result in a compilation error. To simplify error handling, Rust provides a convenient syntactic sugar, the `?` operator. It permits the retrieval of `T` from `Result` if it is `Ok(T)`, or early return of `E` if it is `Err(E)`. Similarly, `Option` simplifies

¹This is the behavior when using `Mutex` on Linux. On Windows, `Mutex` might panic.

error handling with two possibilities: `Some(T)` or `None`, where `None` can signify a trivial error. These types prevent unexpected errors when accessing a potentially non-existing value in the program.



Copy and Drop Traits. 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 goes out of scope. And the `Copy` trait, when implemented for a type indicates that the type should be byte-by-byte copied when the assignment (`=`) operator is used instead of Rust's typical semantic of moving the ownership to the new variable. Interestingly, Rust forbids a type from being `Drop` and `Copy` simultaneously, the designers of the language observed that if a type requires special deallocating code (the `drop` function), then it should also require a special copying function, rather than just copying it byte-by-byte. For instance, a type that holds a reference to the heap requires a `drop` function that frees the data pointed to by the reference, copying the object of the type in a byte-by-byte manner introduces risks of double-free, use-after-free, etc.





Chapter 3 **KrustVM: An SeKVM**

Rust Rewrite

The goal of this work is to leverage Rust’s safety features in a hypervisor TCB by rewriting SeKVM’s Kcore in Rust. The resulting implementation, KrustVM, therefore also leverages the design described in section 2.1. We first forward ported SeKVM from its original Linux 4.18 version to the newest long term support version Linux 5.15 at the time of development. By forward porting we benefit from Linux’s advancements including performance optimizations such as Link-Time-Optimization (LTO) and energy aware scheduling. And new kernel security features including `clang` shadow call stacks, branch target identification, control flow integrity (CFI), ARM Memory Tagging Extension (MTE), ARM pointer authentication, and randomized stack offset per system call.

Once the forward port of SeKVM to Linux 5.15 is done, we then rewrote the implementation in Rust. This chapter describes the challenges that arose when trying to rewrite Kcore in Rust, and the techniques we employed to solve them.



3.1 Integrating Rust and Linux

Linux 5.15, which is the latest long term support kernel version at the time of KrustVM development, does not support Rust as a development language. Therefore, we had to integrate Rust code with the rest of the Linux kernel. We implement Rcore in a single crate on the `no_std` environment and compile it into a single static library. The static library is then linked with the rest of the kernel to create the final kernel image. KVM separates EL2 code from EL1 by grouping EL2 code in a section `.hyp.text`, then mapping that section in EL2's address space at initialization. In Rcore, attribute `#[link_section = ".hyp.text"]` is prepended to all code that should be run in EL2, so that they get placed in the `.hyp.text` section as well. Our implementation is compatible with the Linux kernel codebase. For example, we ensure the page size definition is identical in Rcore and KVM. Also, we share types like `kvm_vcpu` between Linux and Rcore. These type definitions are generated automatically with the tool `bindgen` [4]. For constants that are used by both Linux and Rcore, we copy them from C to Rust manually. Due to the limited support of macro in `bindgen` and the heavy usage of Linux, we do not use it to generate constants. Regarding alignment, field layout order, and padding of custom types, we use the Rust attribute `#[repr(C)]` that ensures the data layout of the marked type has the same layout as in C.

3.2 Bringing up KrustVM on Real Hardware

We chose the Raspberry Pi model 4B (Rpi-4B) to verify our implementation on real hardware. SeKVM's trusted core Kcore originally reserved its private memory by defin-

ing global symbols whose addresses reside right after the kernel image, in the Linux kernel linker script. Kcore then references those symbols to access and utilize the reserved memory. However, there exists an unusable hole in Rpi-4B's physical memory address space, and the bootloader of Rpi-4B places the kernel image before the hole, resulting in an overlap of Kcore's private memory and the unusable hole (Figure 3.1). This makes SeKVM unable to initialize on Rpi-4B.

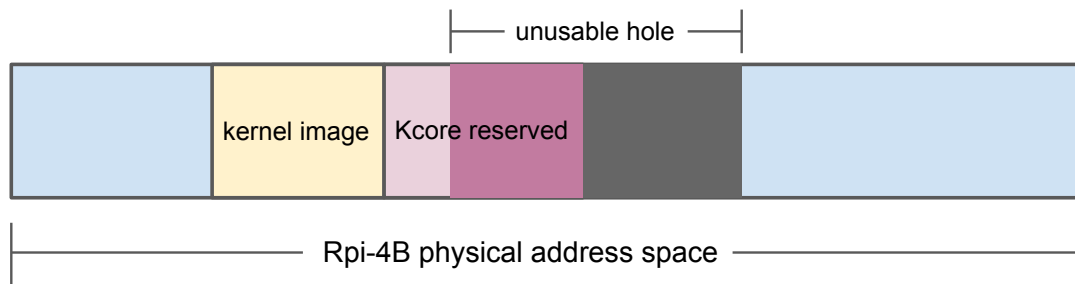


Figure 3.1: Kcore overlaps the unusable hole on Rpi-4B

To solve this issue for KrustVM, instead of allocating memory in the linker script, we first locate a range of memory which does not overlap with the unusable hole of Rpi-4B and the kernel image, then add a new memblock that to correspond to the Rcore's private memory. We mark it as reserved by calling `memblock_reserve`, so that the kernel does not accidentally access this memory range (Figure 3.2). The global symbols previously defined the Linux kernel linker script have also been changed to macros that expand into addresses in the reserved range for KrustVM's Rcore usage.

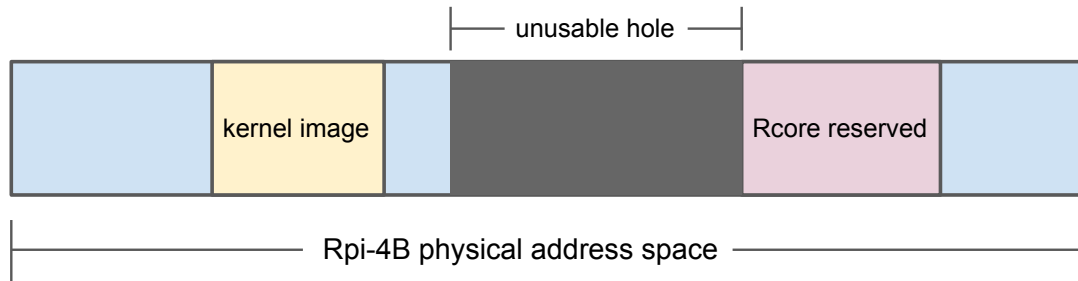


Figure 3.2: Overlap prevention

3.3 Rewriting C-based Kcore into Rust-based Rcore

Given the high complexity of the KVM hypervisor and Kcore, it is clear from the beginning that a top-down approach to a Rust rewrite would be error-prone and difficult to test. Therefore, we elected to start the rewriting effort bottom-up, where all previous C functions are rewritten in Rust, one by one. This incremental approach allows us to test one rewritten function at a time, reducing the risk of introducing bugs. One major downside of this approach is the difficulty of rewriting individual functions in a manner that adheres to Rust’s idiomatic practices. Furthermore, it may result in a lot of `unsafe` blocks. We solve these issues by adding a second phase to the Rust rewrite; after the initial function by function rewrite, we removed unnecessary `unsafe` blocks, refactored the code to be more Rust-idiomatic, and leveraged Rust features to enhance Rcore memory safety. We discuss the features used to secure Rcore memory accesses in chapter 4.



Chapter 4 Securing Rcore Memory

Accesses

4.1 Rcore Memory Regions

Rcore's memory accesses are categorized into four disjoint regions: *Rcore Metadata*, *Page Table Pool*, *SMMU Area*, and *Generic Area*. Rcore metadata and Rcore Page Table pool combined are referred to as the *Rcore area* in the following.

Rcore Area. Rcore needs a reserved memory region separated from the host Linux kernel and all other VMs, named *Rcore area*, to provide its functionality. The Rcore area comprises the Rcore Metadata and the Rcore Page Table Pool. The Rcore Page Table Pool, as its name suggests, keeps private pools of physical pages for NPTs and SMMU page tables so that Rcore has complete control over the permissions and the virtual-to-physical mappings of the memory accessed by the host Linux kernel, VMs, and I/O devices. The Rcore metadata, on the other hand, is used for storing Rcore metadata such as NPT information, physical memory page ownership, VM states, SMMU page table metadata, etc. We constructed custom types to store these Rcore metadata (Table 4.1).

SMMU Area. SMMU is accessed via MMIOs. Rcore unmaps the SMMU from the

Name	Decription of Data
vCPU context	The array that stores the state of each vCPU register.
VM info	The per-VM execution state metadata.
NPT info	The NPT pool allocation status.
PMEM info	The physical memory ownership and sharing status.
SMMU info	The SMMU management and page tables metadata.
SMMUPT info	The SMMU page table pool allocation status.

Table 4.1: Rcore metadata

host NPT to trap-and-emulate its access to the SMMU. This approach assures Rcore has exclusive access to the SMMU.

Generic Area. The *Generic Area* refers to memory outside the Rcore area and the SMMU area. Rcore needs to access this area to modify memory pages belonging to the host or guests for VM services, such as zeroing a page before transferring ownership from a guest back to the host during VM termination.

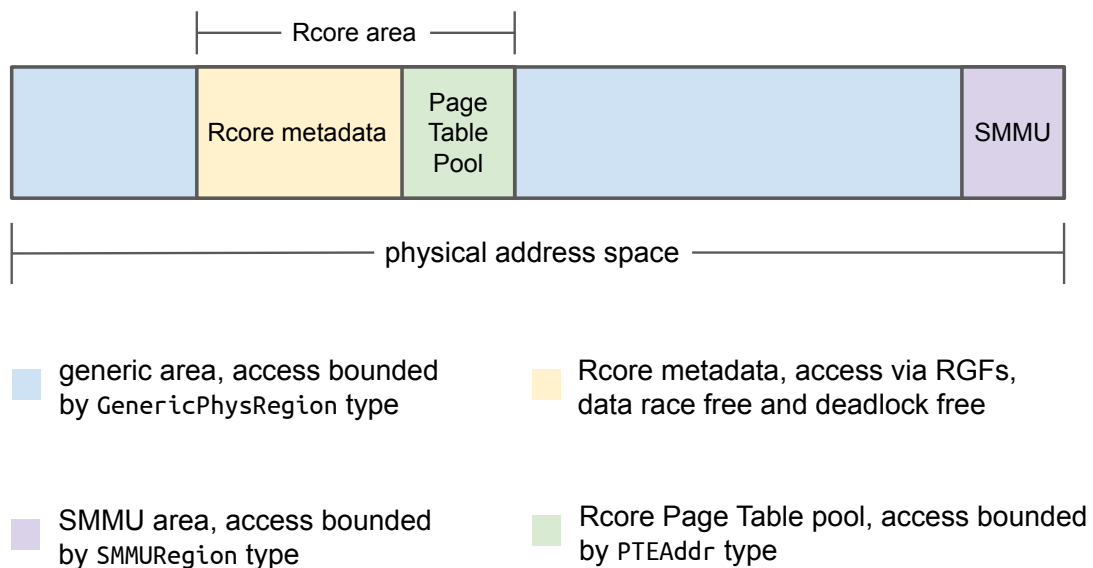
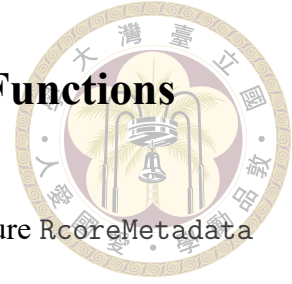


Figure 4.1: Memory Regions

4.2 Rcore Metadata and Reference Getter Functions



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. Fields shared by all CPU cores in `RcoreMetadata` are defined as type `KMutex<T>` (an example is line 3 in Listing 4), where `T` is the type that actually stores Rcore metadata. Rcore's custom `KMutex` is a generic type which can hold any arbitrary type alongside a lock. The only way to access the data wrapped in `KMutex` is by calling the `lock` method of `KMutex` reference. Since there is no existing memory allocator in a hypervisor environment, we directly inform where in the address space Rcore should use. Specifically, we pre-defined the address to the instance of `RcoreMetadata` and manually initialize this memory region at boot time, so it can be used safely thereby. Using that raw address, we implemented several functions that transform the raw pointer to `RcoreMetadata` into mutable references to each of its fields and return them to the caller.

We implement a set of reference getter functions (RGFs). Rcore can use the RGFs to safely access `RcoreMetadata` with safe Rust. Each RGF returns a mutable reference to one of the fields in `RcoreMetadata`, line 10 of Listing 4 is an example of an RGF, 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. take the mutable reference of the field by prepending `&mut`

4. return the mutable reference



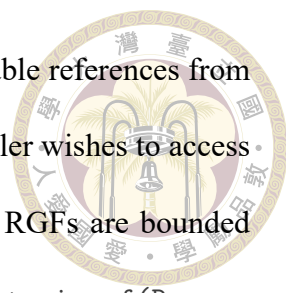
By defining fields of `RcoreMetadata` as `KMutex<T>`, and with the RGFs, most of `Rcore` is free from directly using raw pointers to access `Rcore` metadata, and proper locks are guaranteed to be held when accessing them.

```
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 = /* Rcore's memory address
   ↳ */;
8
9 // the RGF of pmem_info
10 pub fn get_pmem_info<T: CanGetPMemInfo>(_: &mut T) -> &mut
   ↳ KMutex<PMemInfo> {
11     // SAFETY: The pointer points to an initialized memory.
12     // The data is properly wrapped in a KMutex
13     // and the caller have the permission to get PMemInfo
14     unsafe {
15         &mut (*RCORE_METADATA_PTR).pmem_info
16     }
17 }
```

Listing 4: Rcore metadata and Reference Getter Function

4.3 Memory Region Isolation

Raw pointer accesses are prohibited in safe Rust as they easily violate Rust's ownership model. As detailed in the upcoming paragraphs, we examine the need for raw pointers for accessing the four regions described in section 4.1 and the measures taken to guarantee their isolation, even when employing unsafe Rust in their implementation. We also deliberately made the amount of unsafe code that contains raw pointer accesses small (~ 50 LOC).



Raw Pointer Access: Rcore Metadata. The RGFs return mutable references from a raw pointer, thus encapsulating the raw pointer usages when the caller wishes to access Rcore metadata (`RcoreMetadata`). All memory accesses done via RGFs are bounded in the range from `RCORE_METADATA_PTR` to `RCORE_METADATA_PTR + sizeof(RcoreMetadata)`, as accesses to non-array fields will not go out of bounds, and Rust automatically adds runtime checks for the indices when array fields are accessed. We manually check this range is only accessible by Rcore and disjoint from the page table pool and SMMU area by checking it is within the memory range unmapped from the host Linux kernel for Rcore and comparing the addresses with the page table pool area and SMMU area. Hence, it is impossible for Rcore metadata accesses to access the other three regions accidentally.

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. To ensure that code accessing the generic area does not accidentally access the Rcore area, a new type called `GenericPhysRegion` (Listing 5) has been created, which can only point to a memory range in the generic area. `GenericPhysRegion` only has one constructor, namely the `new` method at line 2 in Listing 5. This method 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 `None` variant, indicating that the construction has failed. Listing 6 shows an example usage of `GenericPhysRegion`, which is a function that takes a physical frame number (pfn), and clears the contents of the page. The `GenericPhysRegion::new()` function is called at line 2 with the physical address of the

page (pfn << PAGE_SHIFT) and its size (PAGE_SIZE) as arguments and returns a type of Option<GenericPhysRegion>. Next, we transform Option to Result type through ok_or. and use the ? operator on the Result type to return the contained value to page if it is an Ok variant. Otherwise, clear_page immediately returns Error without executing anything after line 2, effectively propagating the absence of a value up the call stack. The caller of GenericPhysRegion::new() gets a GenericPhysRegion if the check passes; otherwise, clear_page returns an Error type. If successful, the page contents are cleared at line 4.

```

1 impl GenericPhysRegion {
2     pub fn new(start_addr: usize, size: usize) -> Option<Self> {
3         let end = start_addr + size;
4         // overlap check
5         if (end > RCORE_AREA_START && RCORE_AREA_END > start_addr)
6         || (end > SMMU_AREA_START && SMMU_AREA_END > start_addr) {
7             return None;
8         }
9         Some(Self {
10             start_addr,
11             size,
12         })
13     }
14
15     // returns a mutable `u8` slice for the caller
16     // to access generic area memory
17     pub fn as_slice(&self) -> &'static mut [u8] {
18         // convert the physical address to the virtual address
19         let va = pa_to_va(self.start_addr);
20         unsafe {
21             core::slice::from_raw_parts_mut(
22                 va as *mut u8, self.size,
23             )
24         }
25     }
26 }

```

Listing 5: GenericPhysRegion guarantees that every instance points to a valid generic area range

Raw Pointer Access: Page Table Pool. Rcore manages the host's and each VM's NPTs to control their access to physical memory. SMMU page tables control I/O devices'

```

1 fn clear_page(pfn: usize) -> Result<()> {
2   let page = GenericPhysRegion::new(pfn << PAGE_SHIFT,
    ↪   PAGE_SIZE).ok_or(Error::InvalidPfn)?;
3   // the `fill` method for type &[u8] fills the slice with the value
    ↪   passed in
4   page.as_slice().fill(0);
5   Ok(())
6 }

```



Listing 6: Example usage of GenericPhysRegion

memory access. We also leveraged Rust's type system and created the type `PTEAddr` (Page Table Entry Address). Each instance of type `PTEAddr` points to an entry in the Rcore Page Table Pool region. Similar to `GenericPhysRegion`, `PTEAddr`'s constructor verifies whether the physical address provided as an argument for the constructor is within the page table pool region in the Rcore area. If the address falls within the range, it is translated to the corresponding virtual address and stored in a field of the `PTEAddr` instance. Otherwise, the construction fails, and a `None` is returned. This type encapsulates the raw pointer address translation and bound checks so for example the NPT walking code, can guarantee it is accessing NPT entries in the Rcore page table pool area by using `PTEAddr`.

Raw Pointer Access: SMMU. In a manner analogous to the generic area and page table pool, the type `SMMURegion` for accessing SMMU is created. Rcore uses `SMMURegion` whenever it reads or writes SMMU registers. `SMMURegion`'s `new` method takes the MMIO address and verifies its inclusion within the SMMU region. By consistently utilizing this type for SMMU accesses, SMMU accesses are guaranteed to access the correct address region.





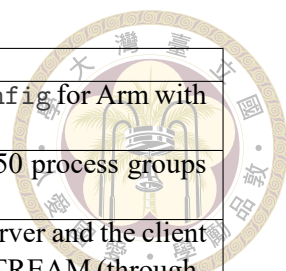
Chapter 5 Evaluation

We evaluated the performance of various application benchmarks on a VM running on KrustVM, SeKVM, and mainline KVM. We also tested the same benchmarks on bare metal environment performances to establish a baseline reference of the benchmark results. We ran the workloads 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, SeKVM, 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 requested the authors of [12] and got a patch for the Linux guest 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, SeKVM, and the mainline KVM.

We configured the hardware with 2 physical CPUs and 1 GB of RAM for the bare metal setup. Each VM that equips with 2 virtual CPUs for the VM setup, and 1 GB of RAM runs on the full hardware available.

We ran the benchmarks listed in Table 5.1 in the VMs on both KrustVM and the mainline KVM. Figure 5.1 shows the normalized results. We normalized the results to



Name	Description
Kernbench	Compilation of the Linux 6.0 kernel using <code>tinyconfig</code> for Arm with GCC 9.4.0.
Hackbench	<code>hackbench</code> [18] using Unix domain sockets and 50 process groups running in 50 loops.
Netperf	<code>netperf</code> [9] 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 (latency).
Apache	Apache v2.4.41 Web server running ApacheBench [20] v2.3 on the remote client, which measures the number of handled requests per second when serving the 41 KB <code>index.html</code> file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	<code>memcached</code> v1.5.22 using the <code>memtier</code> [17] benchmark v1.2.3 with its default parameters.
YCSB-Redis	<code>redis</code> v7.0.11 using the YCSB [6] benchmark v0.17.0 with its default parameters.

Table 5.1: Application Benchmarks

bare-metal performance. 1.00 refers to no virtualization overhead. A higher value means higher overhead. The performance on real application workloads show modest overhead overall for KrustVM compared to SeKVM and mainline KVM.

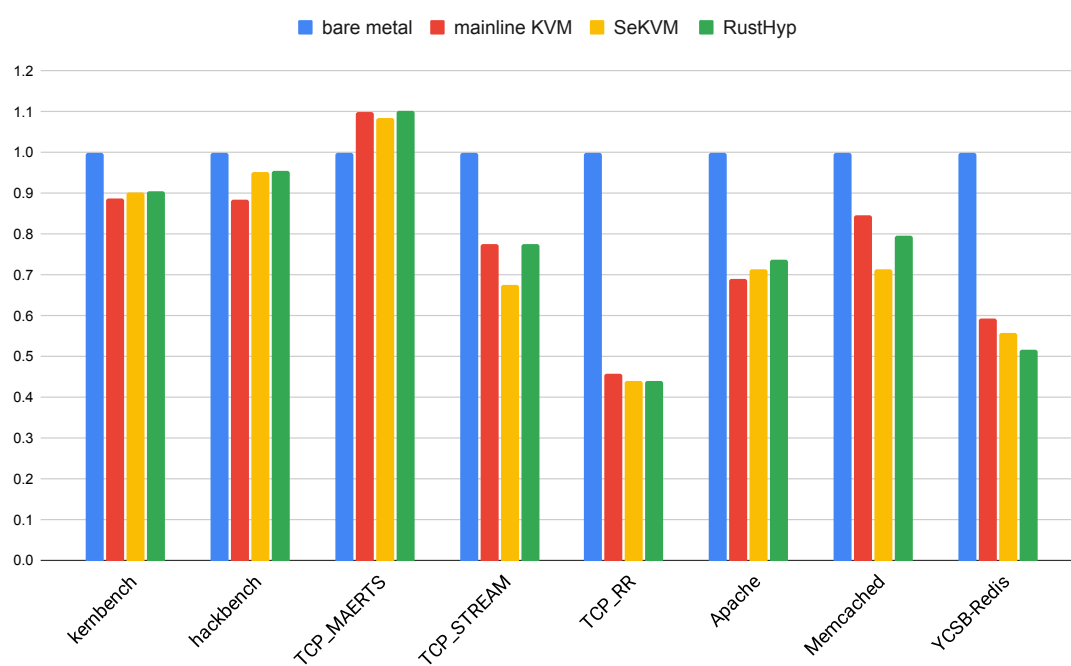


Figure 5.1: Application Benchmark Performance





Chapter 6 Conclusions

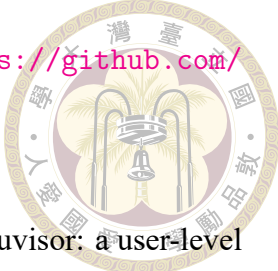
We have presented KrustVM, the first Rust-based secure KVM hypervisor that is rewritten from the C-based SeKVM. Similar to SeKVM, KrustVM delivers VM confidentiality and integrity protection against an untrusted Linux host kernel integrated with KVM. We overcame challenges that surfaced during the C-to-Rust rewrite. We integrated Rust into the Linux codebase, brought up KrustVM on Rpi-4B by changing the method used for reserving memory, and rewritten the SeKVM's TCB in Rust. Moreover, raw pointer accesses are segregated into a small amount of unsafe code to allow most hypervisor functionalities to be implemented in safe Rust. We also leverage Rust's compile-time checks to eliminate memory safety bugs of the TCB. Rust's type system is used to enforce memory region isolation via custom types. KrustVM preserves the performance efficiency of KVM, demonstrating the practicality for deployments.



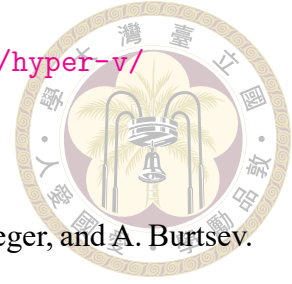


References

- [1] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the servo web browser engine using rust. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), pages 81–89, 2016.
- [2] ARM Ltd. ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ih0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, June 2016.
- [3] A. Bhardwaj, C. Kulkarni, R. Achermann, I. Calciu, S. Kashyap, R. Stutsman, A. Tai, and G. Zellweger. Nros: Effective replication and sharing in an operating system. In OSDI, pages 295–312, 2021.
- [4] bindgen maintainer. bindgen. <https://github.com/rust-lang/rust-bindgen>, 2023.
- [5] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong. Theseus: an experiment in operating system structure and state management. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 1–19. USENIX Association, Nov. 2020.

- 
- [6] Brian Cooper. Yahoo! Cloud Serving Benchmark. <https://github.com/brianfrankcooper/YCSB>, Feb. 2021.
- [7] J. Chen, D. Li, Z. Mi, Y. Liu, B. Zang, H. Guan, and H. Chen. Duvisor: a user-level hypervisor through delegated virtualization, 2022.
- [8] Columbia University. SOSP 21: Artifact Evaluation: Verifying a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. <https://github.com/VeriGu/sosp-paper211-ae>, Sept. 2021.
- [9] R. Jones. Netperf. <https://github.com/HewlettPackard/netperf>, June 2018.
- [10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In In Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007), Ottawa, ON, Canada, June 2007.
- [11] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 234–251, 2017.
- [12] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from commodity hypervisor and host operating system exploits. In Proceedings of the 28th USENIX Conference on Security Symposium, SEC’19, page 1357 – 1374, USA, 2019. USENIX Association.
- [13] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Zhuang Hui. A secure and formally verified linux kvm hypervisor. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1782–1799, 2021.
- [14] Microsoft. Hyper-V Technology Overview. [https://docs.](https://docs.microsoft.com/en-us/hyper-v/)

microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview, Nov. 2016.



- [15] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. Redleaf: Isolation and communication in a safe operating system. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, pages 21–39, 2020.
- [16] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. 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, PLDI 2020, page 763 – 779, New York, NY, USA, 2020. Association for Computing Machinery.
- [17] Redis Labs. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark, Apr. 2015.
- [18] R. Russell. Hackbench. <http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c>, Jan. 2008.
- [19] M. Sung, P. Olivier, S. Lankes, and B. Ravindran. Intra-unikernel isolation with intel memory protection keys. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.
- [20] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. <http://httpd.apache.org/docs/2.4/programs/ab.html>, Apr. 2015.
- [21] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He. Rustee: Developing memory-safe arm trustzone applications. In Annual Computer Security Applications Conference,

ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery.



- [22] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin. Towards memory safe enclave programming with rust-sgx. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, page 2333–2350, New York, NY, USA, 2019. Association for Computing Machinery.