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

doi:10.5566/NTU2023XXXXX

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：______________________

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Acknowledgements

常到外國朋友家吃飯。當蠟燭燃起，菜肴布好，客主就位，總是主人家的小男孩或小女孩舉起小手，低頭感謝上天的賜予，並歡迎客人的到來。

我剛到美國時，常鬧得尷尬。因為在國內養成的習慣，還沒有坐好，就開動了。

以後凡到朋友家吃飯時，總是先囑咐自己；今天不要忘了，可別太快開動啊！幾年來，我已變得很習慣了。但我一直認為只是一種不同的風俗儀式，在我這方面看來，忘或不忘，也沒有太大的關係。

前年有一次，我又是到一家去吃飯。而這次卻是由主人家的祖母謝飯。她雪白的頭髮，顫抖的聲音，在搖曳的燭光下，使我想起兒時的祖母。那天晚上，我忽然覺得我平靜如水的情感翻起滔天巨浪來。

在小時候，每當冬夜，我們一大家人圍著個大圓桌吃飯。我總是坐在祖母身旁。祖母總是摸著我的頭說：「老天爺賞我們家飽飯吃，記住，飯碗裡一粒米都不許剩，要是蹧蹋糧食，老天爺就不給咱們飯了。」

剛上小學的我，正在念打倒偶像及破除迷信等為內容的課文，我的學校就是從前的關帝廟，我的書桌就是供桌，我曾給周倉畫上眼鏡，給關平戴上鬍子，祖母的話，老天爺也者，我覺得是既多餘，又落伍的。

doi:10.5566/NTU2023XXXXX

不過，我卻很尊敬我的祖父母，因為這飯確實是他們掙的，這家確實是他們立的。我感謝面前的祖父母，不必感謝渺茫的老天爺。

這種想法並未因為年紀長大而有任何改變。多少年，就在這種哲學中過去了。

我在這個外國家庭晚飯後，由於這位外國老太太，我想起我的兒時，由於我的兒時，我想起一串很奇怪的現象。

祖父每年在「風裡雨裡的咬牙」，祖母每年在「茶裡飯裡的自苦」，他們明明知道要滴下眉毛上的汗珠，才能撿起田中的麥穗，而為什麼要謝天？我明明是個小孩子，混吃混玩，而我為什麼卻不感謝老天爺？

這種奇怪的心理狀態，一直是我心中的一個謎。

一直到前年，我在普林斯頓，瀏覽愛因斯坦的我所看見的世界得到了新的領悟。

這是一本非科學性的文集，專載些愛因斯坦在紀念會上啦，在歡迎會上啦，在朋友的喪禮中，他所發表的談話。

我在讀這本書時忽然發現愛因斯坦想盡量給聽眾一個印象：即他的貢獻不是源於甲，就是由於乙，而與愛因斯坦本人不太相干似的。

就連那篇亙古以來嶄新獨創的狹義相對論，並無參考可引，卻在最後天外飛來一筆，「感謝同事朋友貝索的時相討論。」

其他的文章，比如奮鬥苦思了十幾年的廣義相對論，數學部份推給了昔年好友的合作：這種謙抑，這種不居功，科學史中是少見的。

我就想，如此大功而竟不居，為什麼？像愛因斯坦之於相對論，像我祖母之

doi:10.5566/NTU2023XXXXX

於我家。

幾年來自己的奔波，做了一些研究，寫了幾篇學術文章，真正做了一些小貢獻以後，才有了一種新的覺悟：即是無論什麼事，得之於人者太多，出之於己者太少。因為需要感謝的人太多了，就感謝天罷。無論什麼事，不是需要先人的遺愛與遺產，即是需要眾人的支持與合作，還要等候機會的到來。越是真正做過一點事，越是感覺自己的貢獻之渺小。

於是，創業的人，都會自然而然的想到上天，而敗家的人卻無時不想到自己。

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

摘要

中文摘要

關鍵字： LaTeX、中文、論文、模板

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Abstract

Abstract

Keywords: LaTeX, CJK, Thesis, Template

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Contents

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

List of Figures

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

List of Tables

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Denotation

HPC	高性能計算 (High Performance Computing)
cluster	集群
Itanium	安騰
SMP	對稱多處理
API	應用程序編程接口

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

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 [16], 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 [13] 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 [14] extended the work of [13] 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-

doi:10.5566/NTU2023XXXXX

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 [2, 4, 11, 17], hypervisors [6, 21], web browsers [1], and TEEs [23, 24]. 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 [14], 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 TODO: elaborate. We also redesigned the Rcore TCB 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. This is achieved with customized Rust types for each memory region that enforces bound-check to accesses, and

doi:10.5566/NTU2023XXXXX

mandating 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 paper will be organized as follows. Background and related work will be discussed in chapter 2. The migration 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 paper in chapter 6.

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Chapter 2 Background and Related Work

2.1 VM Protection

TODO: WIP, and introduce KrustVM design here. Various previous work [8, 12, 14, 15, 25, 26] redesigned the hypervisor to protect VMs. Unlike our work, none of them used Rust to secure their hypervisor implementation. KrustVM and SeKVM [14] both leveraged an earlier design [13] 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.

doi:10.5566/NTU2023XXXXX

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 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.

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

doi:10.5566/NTU2023XXXXX

```rust
// this code sample does *not* compile
{
  let x = 1;
  {              // create new scope
    let y;
    y = x;
  }              // y is dropped

  // compilation error, y's lifetime has ended
  println!("The value of 'y' is {}.", y);
}
```

Listing 1: Rust lifetime example

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 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.

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

doi:10.5566/NTU2023XXXXX

```rust
{
  let mut x = vec![1, 2, 3];
  let y = &x; // immutable borrow of x

  // this line would fail to compile because x is already borrowed immutably by y
  /* let z = &mut x; */

  println!("x = {:?}", x); // This line works
  println!("y = {:?}", y); // This line works

  let z = &mut x; // mutable borrow of x
  z.push(4);

  // this line would fail to compile because x is borrowed mutably by z
  /* println!("y = {:?}", y); */
}
```

Listing 2: Rust enforces *aliasing xor mutability*

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* [18]. 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 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

doi:10.5566/NTU2023XXXXX

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

Listing 3: interior unsafe in Rust's `replace` function

memory safety bugs. The callee in the safe world hence bears no responsibility to ensure safety.

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

doi:10.5566/NTU2023XXXXX

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

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

doi:10.5566/NTU2023XXXXX

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.

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Chapter 3 KrustVM: An SeKVM Rust Rewrite

Our goal is to enhance the security of the trusted Kcore in SeKVM by rewriting it in Rust. 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 syscall.

This chapter describes the challenges that arose when trying to rewrite Kcore in Rust, and the techniques we employed to solve them.

3.1 Integration with 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

doi:10.5566/NTU2023XXXXX

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` [3]. 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, Rust provides an 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 defining 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.

doi:10.5566/NTU2023XXXXX

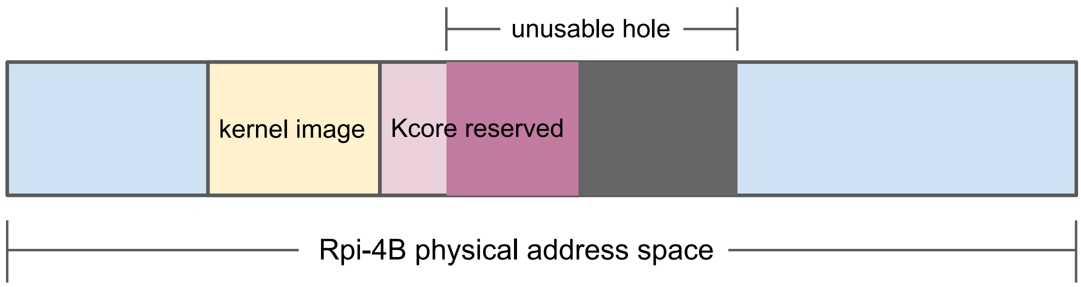

Figure 3.1: Kcore overlaps the unusable hole on Rpi-4B

To solve this issue, 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 call `memblock_reserve` to mark the range of memory as reserved so that the kernel does not accidentally access this memory range (Figure 3.2). The global symbols have also been changed to C macros that expand into addresses in the reserved range for KrustVM's Rcore usage.

doi:10.5566/NTU2023XXXXX

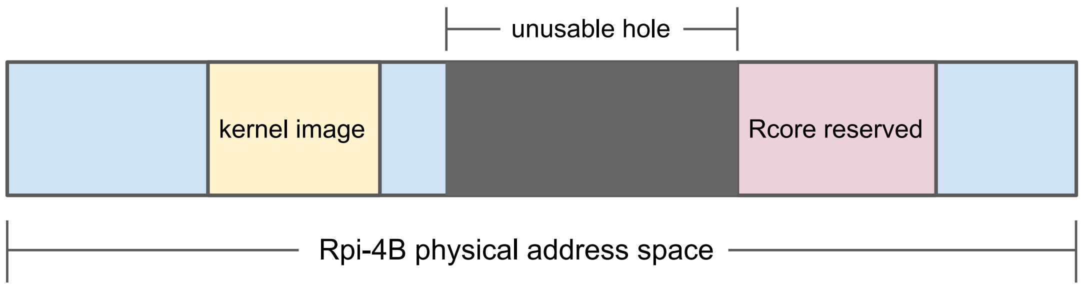

Figure 3.2: Overlap prevention

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 our progress of function rewrite in succession, 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's type system to let Rust automatically check for safety properties.

doi:10.5566/NTU2023XXXXX

We discuss the usage of the type system to secure Rcore in chapter 4.

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Chapter 4 Securing Rcore

Continues from Code Migration (the redesign after the function-by-function rewrite), talk about Rcore here.

Rust has safe and unsafe code -> segregate unsafe code so that most of the hypervisor is written in safe Rust

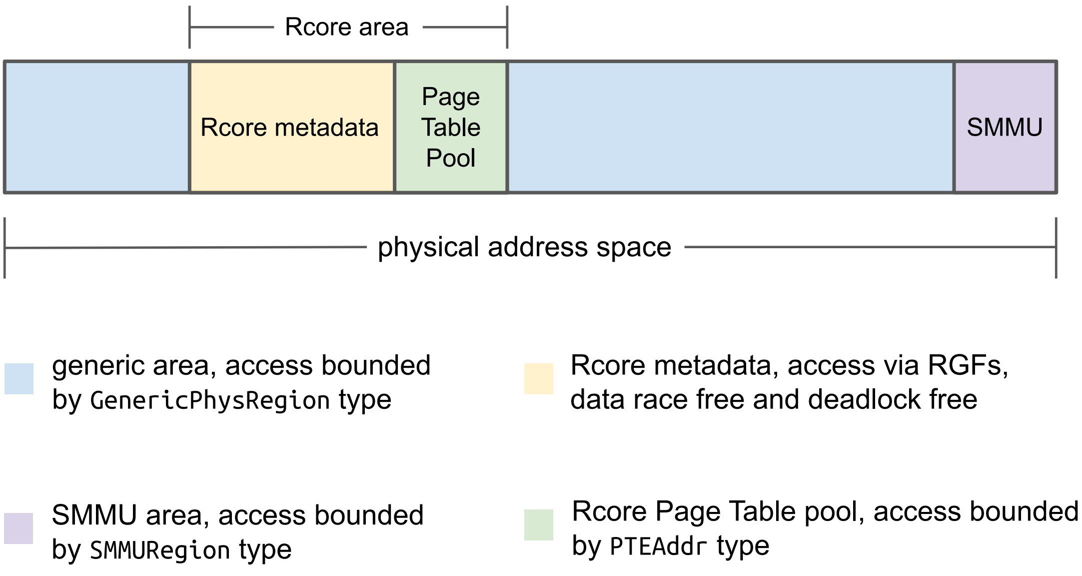

Figure 4.1: Memory Regions

doi:10.5566/NTU2023XXXXX

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

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.

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

doi:10.5566/NTU2023XXXXX

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.

4.2 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 are 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 calcu-

doi:10.5566/NTU2023XXXXX

lating 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` (4) 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 4. 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. 5 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.

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' 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

doi:10.5566/NTU2023XXXXX

```rust
impl GenericPhysRegion {
  pub fn new(start_addr: usize, size: usize) -> Option<Self> {
    let end = start_addr + size;
    // overlap check
    if (end > RCORE_AREA_START && RCORE_AREA_END > start_addr)
    || (end > SMMU_AREA_START && SMMU_AREA_END > start_addr) {
      return None;
    }
    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 4: `GenericPhysRegion` guarantees that every instance points to a valid generic area range

```rust
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 5: Example usage of `GenericPhysRegion`

doi:10.5566/NTU2023XXXXX

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.

doi:10.5566/NTU2023XXXXX

Chapter 5 Evaluation

TODO: add comparison with SeKVM We evaluated the performance of various application benchmarks on a VM running on KrustVM 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 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 extended QEMU v4.0.0 based on the artifact from [7] to support secure VM boot on KrustVM. We requested the authors of [13] 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 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

doi:10.5566/NTU2023XXXXX

Name	Description
Kernbench	Compilation of the Linux 6.0 kernel using `tinyconfig` for Arm with GCC 9.4.0.
Hackbench	`hackbench` [20] using Unix domain sockets and 50 process groups running in 50 loops.
Netperf	`netperf` [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` [22] v2.3 on the remote client, which measures the number of handled requests per second 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` [19] benchmark v1.2.3 with its default parameters.
YCSB-Redis	`redis` v7.0.11 using the `YCSB` [5] benchmark v0.17.0 with its default parameters.

Table 5.1: Application Benchmarks

mainline KVM. Figure 5.1 shows the normalized results. We normalized the results to 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 mainline KVM.

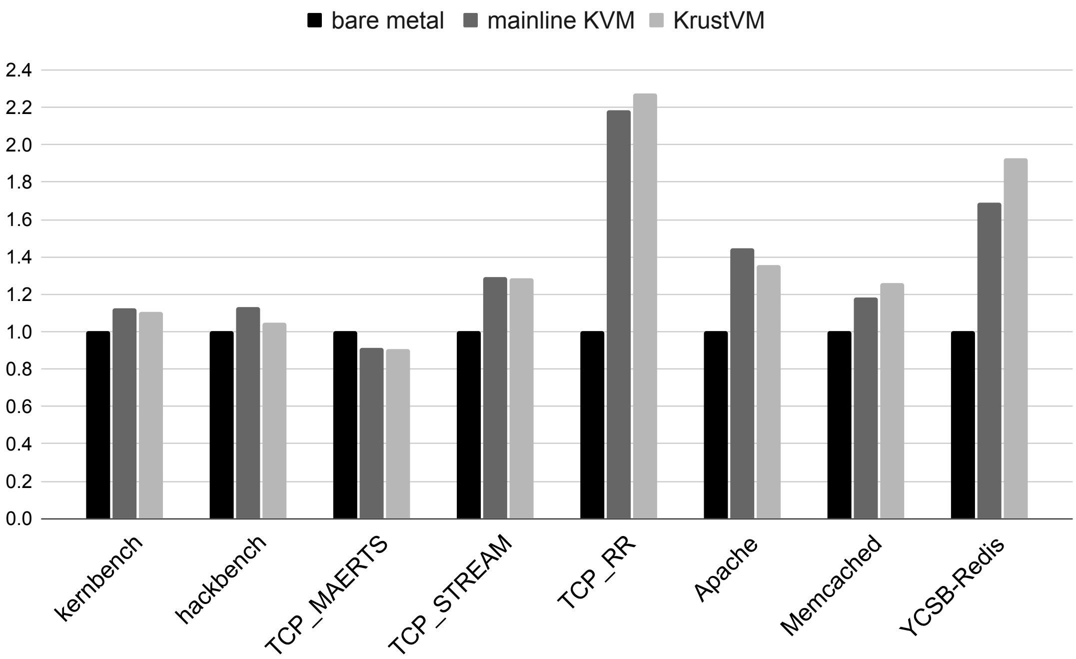

Figure 5.1: Application Benchmark Performance

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Chapter 6 Conclusions

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

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] 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.

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

[4] 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.

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

[6] J. Chen, D. Li, Z. Mi, Y. Liu, B. Zang, H. Guan, and H. Chen. Duvisor: a user-level hypervisor through delegated virtualization, 2022.

doi:10.5566/NTU2023XXXXX

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

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

[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] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan. Twinvisor: Hardware-isolated confidential virtual machines for arm. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 638–654, New York, NY, USA, 2021. Association for Computing Machinery.

[13] 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.

[14] 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.

doi:10.5566/NTU2023XXXXX

[15] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan. (mostly) exitless VM protection from untrusted hypervisor through disaggregated nested virtualization. In 29th USENIX Security Symposium (USENIX Security 20), pages 1695–1712. USENIX Association, Aug. 2020.

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

[17] 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.

[18] 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.

[19] Redis Labs. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark, Apr. 2015.

[20] R. Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, Jan. 2008.

[21] 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

doi:10.5566/NTU2023XXXXX

International Conference on Virtual Execution Environments, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

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

[23] 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.

[24] 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.

[25] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive vm protection against untrusted hypervisor through retrofitted amd memory encryption. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 441–453, 2018.

[26] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011), pages 203–216, Cascais, Portugal, Oct. 2011.

doi:10.5566/NTU2023XXXXX

Appendix A — Introduction

A.1 Introduction

A.2 Further Introduction

doi:10.5566/NTU2023XXXXX

doi:10.5566/NTU2023XXXXX

Appendix B — Introduction

B.1 Introduction

B.2 Further Introduction

doi:10.5566/NTU2023XXXXX