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

實作基於 Rust 之安全 Linux KVM 虛擬機器監測器

On Implementing a Secure Rust-based Linux KVM Hypervisor

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:10.6342/NTU202301822

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

實作基於 Rust 之安全 Linux KVM 虛擬機器監測器

On Implementing a Secure Rust-based Linux KVM Hypervisor

本論文係章瑋麟君(學號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11日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 11 July 2023 have examined a Master's thesis entitled above presented by WEI-LIN CHANG (student ID: R09922117) candidate and hereby certify that it is worthy of acceptance.

口試委員Oral examination committee:

裴士凱	蕭旭君	黃敬群
(指導教授Advisor)		

系主任/所長Director: 洪士灝

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

Acknowledgements

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

摘要

通用的虛擬機器監測器在雲端計算環境中發揮著至關重要的作用，它們負責監管虛擬機器的硬體資源。然而，其日益複雜的設計和廣泛的攻擊面引發了重大的安全憂慮。攻擊者如果利用特權虛擬機器監測器的漏洞，就能夠不受限制地訪問虛擬機器中的數據，從而危及其資訊安全。以前嘗試將虛擬機器監測器重構為小型受信任核心的嘗試存在局限性，因為安全性仍然依賴於受信任核心的實現。此外，對 TCB 的形式化驗證需要大量的人力投入，難以適用於快速發展的軟體專案。最近，由於其強大的記憶體安全保證和高性能，Rust 語言的應用逐漸增加。本論文着眼於解決將 SeKVM 中基於 C 語言的 KVM（內核虛擬機器）TCB 改寫並遷移到 Rust 的挑戰，為此選擇了最近版本的 Linux 長期支持版本。通過這樣的改寫，我們實作出的虛擬機器監測器 KrustVM 不僅能從最新的 Linux 進展中獲益，而且還能受益於 Rust 提供的安全保障。KrustVM 的設計重點在於最大化其不安全 Rust 程式碼的安全性。我們將不安全程式碼與安全 Rust 隔離，並通過安全抽象將不安全程式碼最小化。此外，利用 Rust 的型別系統，我們確保了受信任 Rust 核心進行的不安全記憶體訪問的安全性。與 KVM 和 SeKVM 相比，KrustVM 的性能損失不大，展示了通過 C 到 Rust 的改寫來保障現有虛擬化管理程式的可行性。

關鍵字：系統安全、虛擬化、KVM

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

Abstract

Commodity hypervisors play a vital role in cloud computing environments by overseeing hardware resources for virtual machines. However, their growing complexity and extensive attack surface pose significant security concerns. 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. Moreover, formal verification on the TCB necessitates significant human effort and is not easily applicable to rapidly evolving codebases. Recently, Rust adoption has been increasing for its strong memory safety guarantees and performance efficiency. This thesis addresses challenges in rewriting and porting the C-based KVM TCB in SeKVM to Rust for a recent Linux long term support version. This allows the resulting hypervisor, KrustVM, to not only benefit from recent Linux advancements, but also be protected by Rust's safety guarantees. KrustVM is designed with a focus on maximizing the safety of

doi:10.6342/NTU202301822

its unsafe Rust usages. We minimized and separated unsafe code from safe Rust by enclosing unsafe code within safe abstractions. Additionally, Rust's type system is utilized to ensure the memory safety of the unsafe memory accesses done by the trusted Rust core. KrustVM incurs modest overhead compared to mainline KVM and SeKVM, and demonstrates the practicality of securing existing hypervisors through a C-to-Rust rewrite.

Keywords: System Security, Virtualization, KVM

doi:10.6342/NTU202301822

Contents

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

List of Figures

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

List of Tables

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

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 [28] or Hyper-V [39], 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.

Previously HypSec [34] 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. SeKVM [35] extended the work of HypSec [34] by formally verifying the smaller TCB and requires significant efforts. Previous systems only verified a specific implementation, and as the codebase evolves to incorporate new features or undergo code refactoring, the existing proof becomes outdated, necessitating a new proof for any code modifications.

doi:10.6342/NTU202301822

Rust is an emerging programming language that ensures strong memory safety guarantees at compile time while offering performance efficiency. Its distinctive ownership and lifetime system effectively addresses potential safety issues that programmers may encounter. Rust prevents various memory safety bugs, for example, null pointer dereferences are eliminated by distinguishing between nullable and non nullable types, nullable types are not allowed by default, array out-of-bound accesses are prevented by runtime checks that are added by the compiler, and Rust's ownership system prevents dangling pointers. Further, Rust allows developers to sacrafice the memory safety guarantees for stronger control over the program, for example to directly manage low-level systems resources such as memory. These low-level operations requires the `unsafe` keyword which marks a region of code for which the compiler does not guarantee memory safety. Due to these attributes, various previous work has adopted Rust to implement systems software with critical security and performance requirements, including operating systems [8, 10, 32, 40], hypervisors [12, 46], web browsers [3], and TEEs [49, 50]. 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.

We have developed KrustVM for Linux 5.15, a Rust-based secure Linux KVM hypervisor. By using Rust, we leverage Rust's automatic safety checks, and most of the responsibility of human auditing is shifted to the compiler. The manual auditing efforts can be focused solely on the portions where unsafe Rust is utilized, rather than the entire program. We first ported the Linux 4.18-based SeKVM to Linux 5.15, a recent version of long-term-support Linux. Then SeKVM's verified TCB is reimplemented with a Rust-based TCB, called Rcore. The kernel version update allows us to take advantage of new

doi:10.6342/NTU202301822

kernel features, including Link-Time-Optimization (LTO) and energy-aware scheduling. Similar to SeKVM, KrustVM incorporates the small Rust TCB Rcore to protect VM confidentiality and integrity against the large and untrusted hypervisor codebase that encompasses KVM's host Linux kernel. Performance evaluation of KrustVM on real Arm64 hardware shows that KrustVM incurs modest performance overhead to application workloads compared to mainline KVM and SeKVM.

During the development of KrustVM, we identified and overcame challenges that arose when trying to rewrite and port SeKVM's TCB to Rust for Linux 5.15. Firstly, the Linux kernel had undergone many changes between version 4.18 and 5.15, such as feature addition and kernel API changes. Therefore, we need to forward port SeKVM to Linux 5.15 prior to initiating the Rust rewrite process. Second, Linux 5.15 does not support Rust as a development language, meaning Rust code can not be linked with the rest of the kernel by the Linux build system. To resolve this challenge, we rolled our own Makefile and integrated the build process of our Rust code with Linux's build system. Third, writing a KVM TCB in a new language like Rust poses many language compatibility issues. For example, C headers are not usable in Rust, and name mangling exists in Rust but not in C, etc. We must address each issue for our implementation to work.

Unsafe Rust is necessary for implementing hypervisors like KrustVM, since operations including low-level memory accesses and running system instructions are not possible in safe Rust. This leads to the fact that Rust's safety guarantee does not apply to Rcore in its entirety. Being aware of this, we implemented Rcore in a way such that the amount of unsafe Rust is minimized. Unsafe code are enclosed within a safe abstraction and a safe API is exposed in order to implement complex functionalities in safe Rust, including CPU, memory, VM boot protection, VM exit, and hypercall handlers. Further,

doi:10.6342/NTU202301822

raw pointer accesses, which are unsafe in Rust, are protected using Rust's type system. In Rcore, raw pointers are used for accessing physical memory. Physical memory is divided into multiple disjoint regions, and the Rcore implementation guarantees that all memory accesses done by Rcore are located in the predefined regions, ensuring that bugs caused by pointers pointing to incorrect memory regions are prevented. This involves transforming raw pointers into references, allowing Rust to automatically insert runtime checks for out-of-bound array indices, and building customized Rust types that enforce bound-checking for raw pointer accesses. These measures make Rcore a more memory safe codebase as it contains a small amount of unsafe code, and raw pointer usages are safe-guarded.

The rest of the thesis will be organized as follows. Background will be reviewed in chapter 2. Our threat model and assumptions are listed in chapter 3. The process of implementing a Rust TCB for KVM and the techniques used are described in chapter 4. chapter 5 presents how Rust's safety features are utilized to design and secure Rcore memory accesses. Evaluation of KrustVM and its comparison with mainline KVM and SeKVM is covered in chapter 6. Related work and future work are discussed in chapter 7. At last, we conclude the thesis in chapter 8.

doi:10.6342/NTU202301822

Chapter 2 Background

2.1 Overview of the ARM Architecture

Our work is based on the ARM architecture for its mass adoption in mobile devices, and its rising popularity among major cloud providers [4, 5]. Different from x86, the ARM architecture has a larger general register count, fixed length instructions, and simpler instructions. These properties stem from ARM's original Reduced Instruction Set Computer (RISC) design. CPU privilege levels in ARM are referred as *Exception Levels* (EL), and there are four of them: EL0, EL1, EL2, EL3. The larger the exception level number, the greater the privilege. EL0 is the lowest privilege level designed for userspace software, the `svc` instruction (supervisor call) can be issued is this EL to trap to EL1 for system call service. EL1 is regularly used for running an OS kernel like the Linux kernel. EL1 controls EL0/1 page tables to enable virtual memory for userspace and the kernel space, and sets up the exception vectors to handle EL0 and EL1 exceptions. EL1 can also ask for EL2 service via the `hvc` instruction (hypervisor call). EL2 is designed for running a hypervisor. It is more privileged than EL1, software EL2 is capable of setting various conditions for the hardware to trap to EL2 to intervene the lower EL1 and EL0 execution. For example, it is capable of redirecting all device interrupts to EL2's own exception vector to interpose all interrupts. ARM also provides Nested Page Table (NPT) support in EL2. If EL2 en-

doi:10.6342/NTU202301822

ables NPTs, the physical address that results from an EL0/1 page table walk becomes the *Intermediate Physical Address* (IPA), the IPA must then be translated again by the additional set of page tables set up by the software in EL2 to finally get the physical address used for memory access. The address translation turns into a two stage process, firstly the EL0/1 virtual address is translated into IPA by walking the EL0/1 page table controlled by the kernel in EL1, after that it is translated again by walking the NPT. Thus, when a hypervisor enables NPT, all guest kernels in EL1 only see its own virtual guest physical address space. The hypervisor has full control over the physical memory. Lastly, EL3 is the highest privilege level typically used for running system firmware that initializes the hardware. The *Virtualization Host Extensions* (VHE) is an ARM architecture extension added to support running an unmodified OS kernel designed for an EL1 environment directly in EL2. The extension is needed because originally, EL2 differs from EL1 in a few ways. First, EL1 has two *Translation Table Base Registers* (TTBRs), while EL2 only has one. It was designed like this because OS kernels running in EL1 need the extra base register to separate user process address space and kernel address space, and hypervisors normally do not host applications. Second, there is no *Address Space Identifiers* (ASIDs) support in EL2 for the same reason. Third, the bit layout of some system registers and page table format in EL2 are different from their EL1 counterparts. VHE addresses the problems above by adding another TTBR, ASID support, and synchronized the bit layout of EL2 and EL1 system registers and the page table formats. On hardware that support VHE, the Linux kernel can thus boot in both EL1 and EL2.

doi:10.6342/NTU202301822

2.2 KVM ARM

KVM ARM was merged into the mainline Linux kernel version 3.9 [15]. It was designed to support unmodified guest VMs by utilizing hardware virtualization support introduced in section 2.1. The authors proposed *split-mode virtualization* [16], allowing the KVM ARM hypervisor to split its execution across CPU modes and be integrated into the Linux kernel. Split-mode virtualization installs a small amount of code in EL2 called the *lowvisor* when Linux initializes. The lowvisor is only responsible for hypervisor tasks that can only be done in the more privileged EL2, including running EL2 exception vectors and installing the base addresses of NPTs in the `VTTBR_EL2` register, which holds the NPT root pointer. Split-mode virtualization has various advantages. Kernel features including memory allocation, CPU scheduling can still be done in EL1, thus simplifying the lowvisor, also the small lowvisor makes the addition of KVM ARM a less intrusive change to the Linux codebase, increasing the possibility of it being merged into the mainline kernel for its maintainability and ease of review.

Split-mode virtualization was proposed before the introduction of ARM VHE. With VHE, the entire Linux kernel can be run in EL2, removing the need for KVM to split its execution across CPU privilege levels. Before with split-mode virtualization, the lowvisor must multiplex the EL1 context, or context switch EL1 system registers when entering or exiting VMs, which leads to overhead. By running Linux entirely in EL2, guest EL1 states do not have to be saved or restored each time a VM enter or exit happens, reducing the overhead. KVM ARM was then further developed to support both the new VHE feature (VHE mode), while keeping the option for the original split-mode virtualization, or Non-VHE (NVHE) mode.

doi:10.6342/NTU202301822

2.3 HypSec

HypSec [34] is a new hypervisor design which uses microkernel principles to reduce the trusted computing base of the hypervisor while protecting the confidentiality and integrity of VM data. It is motivated by the fact that as hypervisors become more complex, their ever-growing large codebases expose a huge attack surface for adversaries. HypSec restructures the large monolithic hypervisor into a minimized trusted core, the *corevisor* and the remaining large untrusted host, the *hostvisor*. The corevisor is reduced by separating access control from resource allocation. The corevisor has full access to hardware resources to perform access control to protect VM data. On the other hand, I/O, interrupt virtualization and resource management such as CPU scheduling, memory management, and device management are delegated to the hostvisor, which can leverage a host OS. The corevisor executes at a higher CPU privilege level than the hostvisor, it deprivileges the hostvisor at a lower privileged level, ensuring the untrusted host cannot disable or control privileged hardware features. NPTs are enabled by the corevisor when running the hostvisor and VMs so that they do not have direct access to physical memory. The corevisor unmaps its own private memory pages from the respective NPTs, making them inaccessible to VMs and the hostvisor. The corevisor unmaps a given VM's memory pages from the hostvisor or other VMs' NPTs to isolate these pages. NPTs for the hostvisor and VMs are allocated from the corevisor's memory pool, to which the host and VMs have no access. Since VM and corevisor memory is unmapped from the host NPT, a compromised hostvisor that accesses these memory pages causes an NPT fault that traps to the corevisor. NPT faults are routed to the corevisor itself, allowing it to reject invalid hostvisor memory accesses. The work also used HypSec to retrofit KVM ARM's NVHE mode. NVHE

doi:10.6342/NTU202301822

mode is chosen over VHE mode for the retrofit, as VHE mode runs both the host kernel and KVM entirely in EL2, which prevents the corevisor from restricting the host kernel's full hardware access.

2.4 SeKVM

SeKVM [35] extended the work of HypSec and presented a secure and formally verified Linux KVM hypervisor. While HypSec reduced the trusted computing base of the hypervisor, potential bugs in the TCB can still nullify the guarantee of VM data confidentiality and integrity. SeKVM builds on the design of HypSec and further formally verified the hypervisor TCB. The work proposed *microverification*, where a large codebase such as KVM ARM, is restructured into a small core and a set of untrusted services such that the security of the entire hypervisor can be proven by verifying the small core alone. SeKVM retrofitted KVM ARM's NVHE mode into the trusted *KCore* and the set of untrusted services *KServ*. To verify Kcore, *security-preserving layers* are introduced to modularize the verification process. KCore's detailed C and assembly implementations are abstracted into higher-level specifications with the help of the Coq proof assistant, the specifications are then used to prove security properties that would be intractable to verify directly on the implementation.

2.5 The Rust Programming Language

Rust is a relatively young programming language compared to C that aims to be safe and fast. It enables programs to be memory-safe without requiring programmers to manually manage memory as in traditional languages (e.g. C/C++). Different from other

doi:10.6342/NTU202301822

memory-safe languages such as Python or Go, Rust does not employ garbage collection for managing memory. Instead, the concepts of lifetimes, ownership, and borrowing rules are introduced to mandate the programmer to follow specific rules. Statically enforcing programming rules empowers Rust to perform comparably to C. The rules are checked at compile time, eliminating the need for runtime checks that incur overhead. Furthermore, these checks ensure adherence to the specified rules, and if no violations are found, the code is directly translated into machine instructions without any additional overhead or alteration in behavior. 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.

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.

doi:10.6342/NTU202301822

```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

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.

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

doi:10.6342/NTU202301822

```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*

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* [41]. 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 nec-

doi:10.6342/NTU202301822

```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

essary 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.

Interior Mutability. Mutating referenced data via an immutable reference is forbidden in Rust. However, this is sometimes too restrictive for implementing efficient algorithms or data structures. For instance, a cache might be desirable for a read-only search data structure to optimize lookup time. Nevertheless, updating the cache state requires mutability for the cache, violating the read-only constraint. Hence, a mechanism is needed for mutating data under a read-only variable. The Rust standard library provides some special types that allow the user to modify data even with read-only access, to address this issue. This design pattern is known as *Interior Mutability*. `unsafe` operations are used to implement these types to bend Rust's usual rules that govern mutation and borrowing. These types ensure the borrowing rules are followed, i.e. one mutable borrower at one time, and no mutable borrowers when read-only borrowers exist, at runtime. A panic occurs whenever the runtime checks fail, stopping the program to avoid safety issues. For example, `Mutex` in Rust provides interior mutability. A lock is used to ensure that only one borrower of the inner data exists at one time. More precisely, when attempt-

doi:10.6342/NTU202301822

ing to borrow data that has already been borrowed, the `Mutex` enforces a busy wait until the data is released, 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.

Generics and Traits. In addition to the safety mechanisms, Rust also provides features for writing code that operates on values of many different types. `Generic` allows code to work with type parameters, reducing similar code that work with different types. For example, the vector type in Rust's standard library `std::vec::Vec` is capable of holding an array of an 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.

Error Handling. Rust offers enum types `Result<T, E>` and `Option<T>` that have variants to explicitly represent the state of error. A `Result` type can be the enum variant `Ok(T)`, which denotes a proper result with type `T`, or `Err(E)`, which represents an error with reason of type `E`. To simplify error handling, Rust provides a convenient syntactic sugar, the `?` operator. When used on a `Result`, it retrieves the `T` from `Ok(T)`. However, if the `Result` is `Err(E)`, the `Err` variant is immediately returned from the enclosing function, propagating the error to the caller. When handling enum types, the program must

doi:10.6342/NTU202301822

handle all variants of the enum, and not doing so results in a compilation error, this enforces the programmer to handle all possible cases, including errors. Similarly, `Option` can have the `Some(T)` variant, or the `None` variant, which represents the state of not having a value. These types prevent unexpected errors when accessing a potentially non-existing value, or a potential error 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:10.6342/NTU202301822

doi:10.6342/NTU202301822

Chapter 3 Assumptions and Threat Model

We assume a remote attacker or a curious administrator that aims to compromise the integrity and confidentiality of VM data. An attacker can exploit bugs in the host kernel integrated with KVM. A remote attacker cannot access the hardware, so physical attacks such as cold boot attacks [22] and memory bus snooping are out of scope. On-site security measure [19] is assumed to be in place to prohibit unauthorized physical access to the hardware. Side-channel attacks [7, 24, 36, 43, 52, 53] are also excluded from our threat model.

We assume a VM does not voluntarily reveal its sensitive data, intentionally or by accident. A VM can be compromised by a remote attacker that exploits vulnerabilities in the VM. We do not provide security features to prevent or detect VM vulnerabilities, so a compromised VM that involuntarily reveals its data is out of scope. However, attackers may try to attack other hosted VMs from a compromised VM for which we provide protection.

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

Chapter 4 Implementing a Linux KVM TCB in Rust

In the process of developing KrustVM, 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. Once the forward port of SeKVM to Linux 5.15 is done, we then rewrote the SeKVM TCB Kcore in Rust to create Rcore, KrustVM's TCB. This chapter describes the challenges that arose when implementing a Rust-based KVM TCB for Linux 5.15, and the techniques employed to solve them.

4.1 Forward Porting SeKVM from Linux 4.18 to Linux 5.15

The Linux kernel gained many new features between version 4.18 and 5.15, 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. By forward porting SeKVM from its original Linux kernel version 4.18 to 5.15, the codebase

doi:10.6342/NTU202301822

can benefit from these advancements.

SeKVM is based on the mainline KVM ARM in NVHE mode, therefore, to forward port it to a newer kernel version, Kernel functions called by SeKVM must be updated. For example, the data cache flushing function `__flush_dcache_area` is changed to `dcache_clean_inval_poc` in Linux 5.15. All outdated functions and macros in the SeKVM codebase are updated. Moreover, a new KVM mode pkvm [25] is added to mainline KVM ARM in Linux 5.11, we made sure the logic of SeKVM and pkvm is separated such that the two modes of operation can coexist in the codebase. This is done by checking for the kernel configuration at KVM initialization, if the configuration option for SeKVM is set, pkvm will not be initialized. Mainline KVM had also made the code that runs in ARM's hypervisor mode (EL2) more self-contained. Symbols belonging to EL2 are isolated from kernel mode symbols name-wise, a prefix `__kvm_nvhe_` is prepended to all symbols in EL2. Parts of SeKVM that references symbols in the original NVHE KVM EL2 code then must adjust how it references those symbols. The predefined helper macro `CHOOSE_NVHE_SYM()` is used, it prepends the prefix (`__kvm_nvhe_`) for referencing NVHE symbols so that it is not required to write `__kvm_nvhe_` every time the code references a NVHE symbol. This makes our code cleaner and easier to maintain. For SeKVM symbols that need to be referenced by the original NVHE KVM EL2 code, in this case, the helper macro `KVM_NVHE_ALIAS()` is used, which creates an additional symbol referring to the same piece of data as the input symbol whose name is prepended by the NVHE prefix, enabling the NVHE KVM EL2 code to reference it. Furthermore, to resolve the issue that the compiler optimizing struct zeroing operations with `memset` calls, which are not mapped in EL2, the C compiler flag `-ffreestanding` is included during the compilation of SeKVM.

doi:10.6342/NTU202301822

4.2 Integrating Rust and Linux

In order to write a KVM TCB in Rust, Rust code must be compiled and linked with the rest of the Linux kernel. However, 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. As a result, incorporating our Rust code into the kernel requires the developer to manually invoke the Rust compiler to build the Rust crate, and then link it against the kernel, which can be both laborious and susceptible to errors. To overcome this challenge, we integrated the Rust toolchain with the Linux kernel build system. A new subdirectory in Linux's source path `arch/arm64/krustvm` is created, and it contains the Rcore crate and the `Makefile` for this directory. Rcore is implemented in a single crate on the `no_std` environment and compiled into a single static library `libkrustvm.a`. To support building `libkrustvm.a` and linking it with the rest of the kernel with `make`, the following is added to the Makefile:

1. append `libkrustvm.a` to Kbuild built-in object goals `obj-y` by adding the line

   `obj-y += libkrustvm.a`

2. define Makefile target to instruct `make` to use `cargo` to generate `libkrustvm.a`.

```
1 $(obj)/libkrustvm.a: $(src)/krustvm/src/*.rs
2         cargo build --release --target=aarch64-unknown-linux-gnu
```

3. convert `libkrustvm.a` into `krustvm.o` by calling `ld`

The Makefile in `arch/arm64/krustvm` generates `krustvm.o`, and the kernel build system will then link this file with all other object files in the kernel and produce the final kernel image.

doi:10.6342/NTU202301822

4.3 Rewriting C-based Kcore into Rust-based Rcore

4.3.1 The Rewrite Process

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 in the TCB 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, such as using Rust's pattern-matching `match` syntax instead of the C-like `if` statements, and using references instead of raw pointers. Furthermore, it may result in a lot of `unsafe` blocks. These issues are solved 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.

4.3.2 Rust Code Organization

Rust packages code into *modules*, modules are containers for functions, types, constants, traits, etc. Rust programs or libraries are made up of one or multiple modules. Rcore consists of multiple modules, including the typical utility functions module, and modules that contain functions that implement different hypervisor tasks, for example mapping a page in the host kernel's NPT. Moreover, each of the Rcore metadata types (Table 4.1) used for storing NPT information, physical memory page ownership, VM in-

doi:10.6342/NTU202301822

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

formation, SMMU page table metadata, etc., is implemented as its own module that defines the type and its associated type methods. One of the modules is `VMInfo`, it includes the definition of the type `VMInfo`, which stores information of a VM including its VMID, state, and an array of VCPU states. The module also contains methods for reading the VMID, setting the state of the VM, etc. Another module aggregates the Rcore metadata structures into a single big structure `RcoreMetadata` (line 1 in Listing 4) to simplify the memory used by these metadata. Some fields in `RcoreMetadata` are shared by all CPU cores, while others are per CPU. We leverage the custom mutex type `KMutex` from [13] to protect concurrent accesses to fields shared by all CPU cores in `RcoreMetadata`. Shared fields 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. Different from Rust, C does not support methods for structs, it therefore lacks the ability to present an API that provides type-specific functionalities while hiding how the method's implementation manipulates the structure's data. For instance in Listing 5, users of `VMInfo` is forbidden from accessing the `vmid` field of `VMInfo` directly, but must call the `get_vmid` method. The user thus can not arbitrarily modify data inside the structure. This Rust feature helps eliminate bugs such as writing to read-only fields, and accessing fields that are not intended to be exposed to the users of the type.

doi:10.6342/NTU202301822

```rust
struct RcoreMetadata {
  [...] // other fields omitted
  pub pmem_info: KMutex<PMemInfo>,
  [...] // other fields omitted
}

const RCORE_METADATA_PTR: *mut RcoreMetadata = /* Rcore's memory address
  ↪  */;

```

Listing 4: Rcore metadata

```rust
// in the VM module:
pub struct VMInfo {
    vmid: u32,
    [...] // other fields omitted
}

impl VMInfo {
    #[inline(always)]
    pub fn get_vmid(&self) -> u32 {
        self.vmid
    }
}
```

Listing 5: type method example

4.3.3 Rust-Rewrite Challenges

Enforcing Linking Section. 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.

Matching Linux Types and Constants. Our implementation is compatible with the Linux kernel codebase. For example, the page size definition is identical in Rcore and KVM. For types shared between Linux and Rcore like `kvm_vcpu`, the type definitions are generated automatically with the tool `bindgen` [9]. `Bindgen` can generate Rust type definitions by parsing C's struct definitions, saving developers' time that would otherwise

doi:10.6342/NTU202301822

be spent defining the same type. For types that are shared by both C and Rust, the Rust attribute `#[repr(C)]` is used to ensure their alignment, field layout order, and padding are the same in both languages, to prevent data corruption. This happens for example when the field layout of a structure is different for the two languages, resulting in C and Rust accessing distinct offsets within the structure when reading or writing to the same field. And for constants that are used by both Linux and Rcore, including the page size mentioned above, they are copied from C to Rust manually. Due to the limited support of macro in `bindgen` and its complex usages by Linux, `bindgen` is not used to generate constants.

Entry Point Binding. Whenever an exception gets taken to EL2, the CPU switches its exception level to EL2, saves the program status and exception syndrome, and jumps to the preassigned exception vector. We modify the exception vectors, which are written in assembly, to call Rcore's entry point functions instead of the original C handlers to transfer the control flow to our Rust code. Rcore's entry point functions must be annotated with the Rust attribute `#[no_mangle]`. This attribute informs the Rust compiler that the function name should not be mangled, in order for the linker to resolve the symbol reference in the exception vectors.

4.3.4 Unsafe Rust Usages

A small part of Rcore's implementation is coded in unsafe Rust. Unsafe Rust exists because the underlying computer hardware is inherently unsafe. Certain tasks are impossible without unsafe operations, such as directly accessing a specific address to configure the interrupt controller, or issuing a memory barrier instruction in the middle of a function. Overall, the source of unsafe Rust includes inline assembly, the Foreign Function

doi:10.6342/NTU202301822

Interface (FFI), KVM ARM Per-CPU variables, and raw pointer accesses. The first three categories are discussed in this section, and for raw pointer usages, chapter 5 shows how each raw pointer usage scenario is checked to guarantee their memory safety.

Inline Assembly. Inline assembly are used for system instructions (e.g. TLB invalidation instructions) and system register accesses. ARM architecture uses the `mrs` instruction to read a system register's value to a general purpose register, and the `msr` instruction to write a system register with the content of a general purpose register. Inline assembly can be inserted in Rust code with the help of Rust's built-in `core::arch::asm` module. It can be used to embed handwritten assembly in the assembly output generated by the compiler. For system register accesses, the `aarch64-cpu` crate [1] is imported into our Rcore crate, it provides a clean API for reading and writing AArch64 system registers. The actual inline assembly usages of `mrs` and `msr` are abstracted behind `aarch64-cpu`'s safe APIs.

FFI. FFIs are used for calling longer assembly routines, below is a list showing the FFI routines used in Rcore.

- `__guest_enter`: context switching general purpose registers and entering guest VMs.
- `dcache_clean_inval`: invalidate cache of the input memory range.
- `acquire_lock` and `release_lock`: Rcore's spinlock primitive that spin on an address using an assembly loop.
- `tlb_flush_ipa`: flushes the input intermediate physical address range of the vmid given.

doi:10.6342/NTU202301822

KVM ARM Per-CPU Variables in Rust. Using KVM ARM Per-CPU variables is a special case for unsafe Rust. Mainline KVM has its own EL2 per CPU variable mechanism (Figure 4.1); it is implemented by first allocating enough space for all cores to have a copy of the per CPU variables, then, for each CPU core, it records the offset from its copy of the variables to the base copy. This per-core offset is then stored in each core's `TPIDR_EL2` system register. When there is a requirement to access a per CPU variable, the address of the base copy is first acquired, then `TPIDR_EL2`'s value is added to the base copy's address to calculate the per-core address. KrustVM continues to use this mechanism by declaring the symbol which corresponds to the base address as a Rust extern static variable, take its raw address, then add the value in `TPIDR_EL2` to it. This approach requires three `unsafe` statements, first from reading the address of the extern static variable, then reading `TPIDR_EL2` via inline assembly, and lastly, another `unsafe` to dereference the calculated address. Concurrent accesses will not pose a problem since each core accesses a different address.

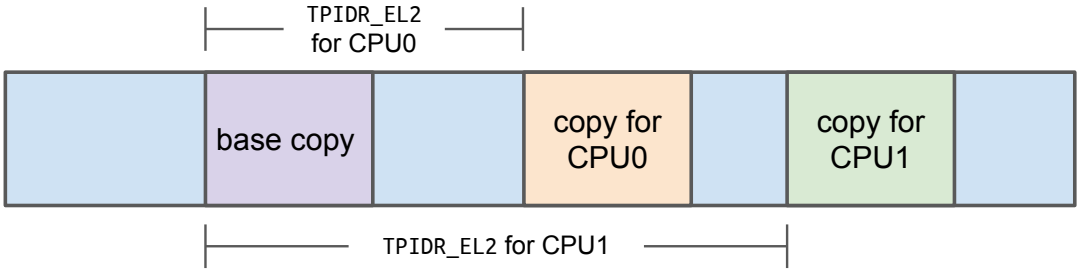

Figure 4.1: KVM ARM Per-CPU Variables Mechanism

4.4 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-

doi:10.6342/NTU202301822

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 is an unusable hole reserved for the GPU in the physical memory address space of Rpi-4B, spanning from 948MB to 1GB. The proprietary bootloader of Rpi-4B restricts the placement of the kernel image within the range of 0 to 948MB. Considering that SeKVM requires more than 1GB of private memory, an overlap between Kcore's private memory and the unusable hole becomes unavoidable (Figure 4.2). This makes SeKVM unable to initialize on Rpi-4B.

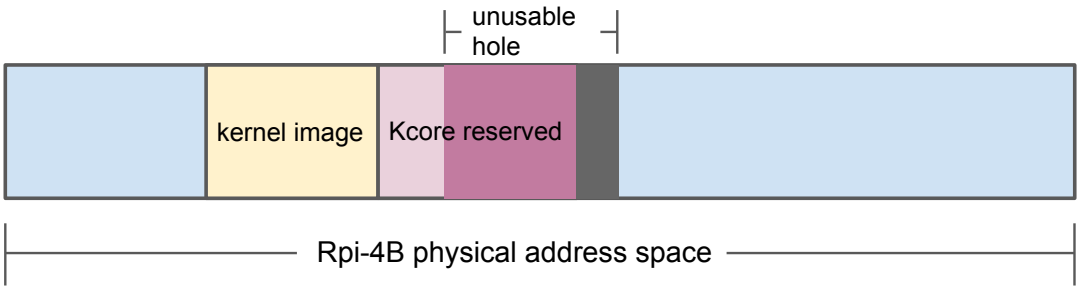

Figure 4.2: Kcore overlaps the unusable hole on Rpi-4B

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 4.3). Malicious accesses to Rcore private memory from the host kernel are prevented by unmapping the private memory from the host kernel's NPT.

doi:10.6342/NTU202301822

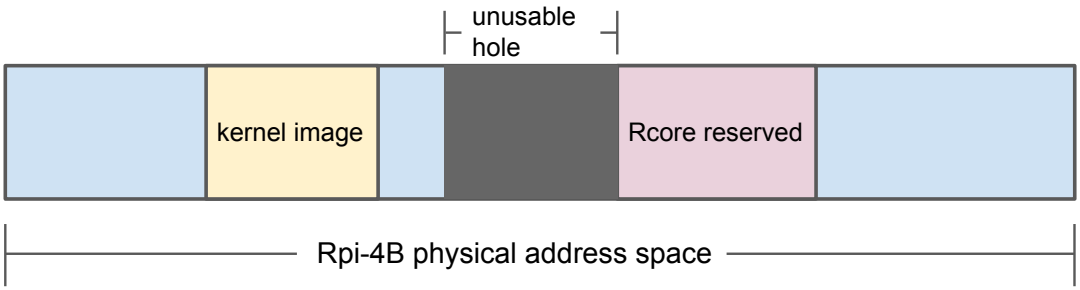

Figure 4.3: Overlap prevention

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

Chapter 5 Securing Rcore Memory Accesses

Numerous software bugs arise from the disparity between the address a pointer points to and the intended address it should reference. For example, NPT walking code must calculate addresses of each level's page table entry. If the address calculation is erroneous, the NPT walking code may dereference the pointer which points to an unrelated region of memory, and write to guest memory, or other hypervisor metadata, leading to crashes or vulnerabilities. Since Rcore contain raw pointer accesses, which are not protected by Rust, pointers referring to unintended areas are also possible in our codebase. To tackle these kinds of bugs, Rcore's memory accesses are categorized into disjoint regions, and raw pointer accesses to a specific region are guaranteed to not access any of the other regions. With the memory region isolation, the NPT address miscalculation above will be eliminated, as raw pointers specified for NPT accesses will be guaranteed to not point to any other region. section 5.1 describes each of the memory region defined for Rcore's memory accesses, and section 5.2 presents how raw pointers are guaranteed to access the intended region.

doi:10.6342/NTU202301822

5.1 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 intermediate physical address to physical address 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 described in subsection 4.3.2.

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 SMMU area. This area is used for host OS and guest VMs operation, initially all memory belongs to the host, and memory are allocated for the guest VMs by the host OS as the VMs get created and start to consume memory. 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.

doi:10.6342/NTU202301822

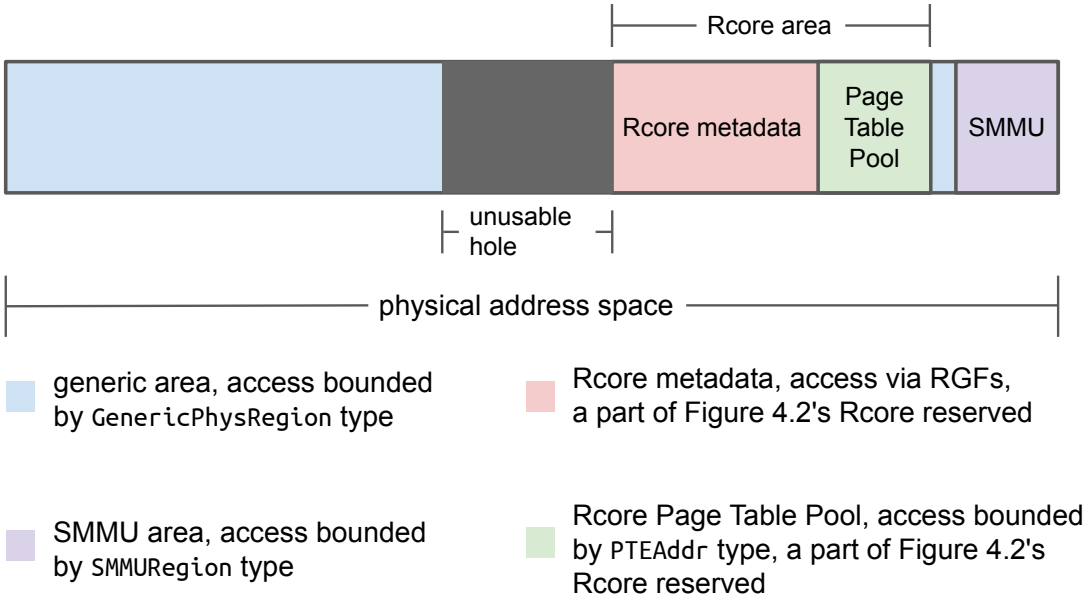

Figure 5.1: Memory Regions

5.2 Memory Region Isolation

Raw pointers are types in Rust that are not checked by Rust's aliasing xor mutability rule, meaning there can be multiple raw pointers pointing to the same piece of data. Further, raw pointers are also nullable. The relaxation of these safety rules opens up the potential for memory safety bugs including null pointer dereferences or use-after-free when raw pointers are accessed. Hence, raw pointer accesses are prohibited in safe Rust. As detailed in the upcoming paragraphs, we examine the need for raw pointers for accessing the four regions described in section 5.1 and the measures taken to guarantee their isolation, even when employing unsafe Rust in their implementation. The amount of unsafe code that contains raw pointer accesses is also deliberately made small (∼50 LOC).

doi:10.6342/NTU202301822

5.2.1 Raw Pointer Access: 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, the address to the instance of `RcoreMetadata` is pre-defined, and the memory region is initialized at boot time, so it can be used safely thereby. Using that raw address, several functions are implemented that transform the raw pointer to `RcoreMetadata` into mutable references to each of its fields and return them to the caller.

We leverage the design proposed in [13] and implemented 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 2 of Listing 6 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.

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

doi:10.6342/NTU202301822

```rust
// the RGF of pmem_info
pub fn get_pmem_info<T: CanGetPMemInfo>(_: &mut T) -> &mut
  ↪ KMutex<PMemInfo> {
  // SAFETY: The pointer points to an initialized memory.
  // The data is properly wrapped in a KMutex
  // and the caller have the permission to get PMemInfo
  unsafe {
    &mut (*RCORE_METADATA_PTR).pmem_info
  }
}
```

Listing 6: Rcore Reference Getter Function

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. The Rcore metadata region is unmapped from the host Linux kernel, and we check its address to ensure that it does not overlap with the page table pool area or the SMMU area. Hence, it is impossible for Rcore metadata accesses to access the other three regions accidentally.

5.2.2 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 7) 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 7. 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 spec-

doi:10.6342/NTU202301822

ified range overlaps with the Rcore area or the SMMU area, the constructor returns a `None` variant, indicating that the construction has failed. Listing 8 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, `Option` is transformed 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.

5.2.3 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 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

doi:10.6342/NTU202301822

```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 7: `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 8: Example usage of `GenericPhysRegion`

doi:10.6342/NTU202301822

in the Rcore page table pool area by using `PTEAddr`.

5.2.4 Raw Pointer Access: SMMU

In a manner analogous to the generic area and page table pool, the type `SMMURegion` for accessing SMMU is created. `SMMURegion`'s `new` method takes the MMIO address and verifies its inclusion within the SMMU region. `SMMURegion` is the only type that contain raw pointers pointing to the SMMU area, as other types' constructors reject addresses pointing to the SMMU area. Therefore, Rcore must use `SMMURegion` whenever it reads or writes SMMU registers. By utilizing this type for SMMU accesses, SMMU accesses are guaranteed to access the correct address region.

doi:10.6342/NTU202301822

Chapter 6 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. The 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, SeKVM, and mainline KVM are all 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 [34] 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 mainline KVM. 2 physical CPUs and 1 GB of RAM is configured for the bare metal setup, and each VM is equipped with 2 virtual CPUs , and 1 GB of RAM.

We ran the benchmarks listed in Table 6.1 in VMs on KrustVM, SeKVM, and mainline KVM. Figure 6.1 shows the normalized results. In Figure 6.1 we normalized the results to bare-metal performance. 1.00 refers to no virtualization overhead, and a higher

doi:10.6342/NTU202301822

value means higher overhead. The performance on real application workloads show modest overhead overall for KrustVM compared to SeKVM and mainline KVM.

In the `TCP_MAERTS` benchmark, it can be observed that mainline KVM, SeKVM, and KrustVM all outperformed the bare-metal setup. This is caused by the Virtio driver batching multiple packet sends before submitting data to the NIC. In contrast, the bare-metal driver submits data to the NIC for each individual transmission, leading to a higher overhead. `TCP_STREAM`, which measures bulk data receive performance, does not demonstrate the advantages of packet batching because receiving packets causes additional VM exits as the hypervisor injects virtual interrupts to notify the VM of incoming packets. The overhead caused by these extra VM exits dwarfs the benefits provided by packet batching.

For `TCP_RR` and the YCSB-Redis benchmarks, KrustVM experienced higher overhead difference compared to mainline KVM at around 8% and 14%, respectively. Performance of the bare-metal setup of these two benchmarks are roughly twice as good as the VMs, amplifying the difference between mainline KVM and KrustVM when plotting Figure 6.1. In fact, when the performance is normalized against mainline KVM (Figure 6.2), all benchmarks executed on KrustVM demonstrate an overhead of less than 10% compared to mainline KVM. This shows that KrustVM is only slightly less performant than mainline KVM.

The overhead difference of KrustVM and mainline KVM is more significant in the networking benchmarks (i.e. Netperf, Apache, Memchached, YCSB-Redis). The rationale for this can be illustrated by comparing how network data is exchanged between the guest and the host. We configured a Virtio network device for VMs. For mainline KVM, the memory buffers in the Virtio rings are shared between the host kernel and VMs, such

doi:10.6342/NTU202301822

Name	Description
Kernbench	Compilation of the Linux 6.0 kernel using `tinyconfig` for Arm with GCC 9.4.0.
Hackbench	`hackbench` [44] using Unix domain sockets and 50 process groups running in 50 loops.
Netperf	`netperf` [27] 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` [48] 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` [42] benchmark v1.2.3 with its default parameters.
YCSB-Redis	`redis` v7.0.11 using the `YCSB` [11, 14] benchmark v0.17.0 with its default parameters.

Table 6.1: Application Benchmarks

that the network device can DMA data directly into a guest-visible buffer. In contrast, KrustVM does not allow the host kernel to share memory with VMs by default. However, to support Virtio, KrustVM provides the `GRANT_MEM` and `REVOKE_MEM` hypercalls which can be explicitly called by the guest kernel to share memory with the host kernel. Therefore, KrustVM VMs must invoke additional hypercalls while running network workloads, inducing extra overhead. SeKVM also uses the same set of hypercalls for the network workloads, thus incurring the same type of overhead.

doi:10.6342/NTU202301822

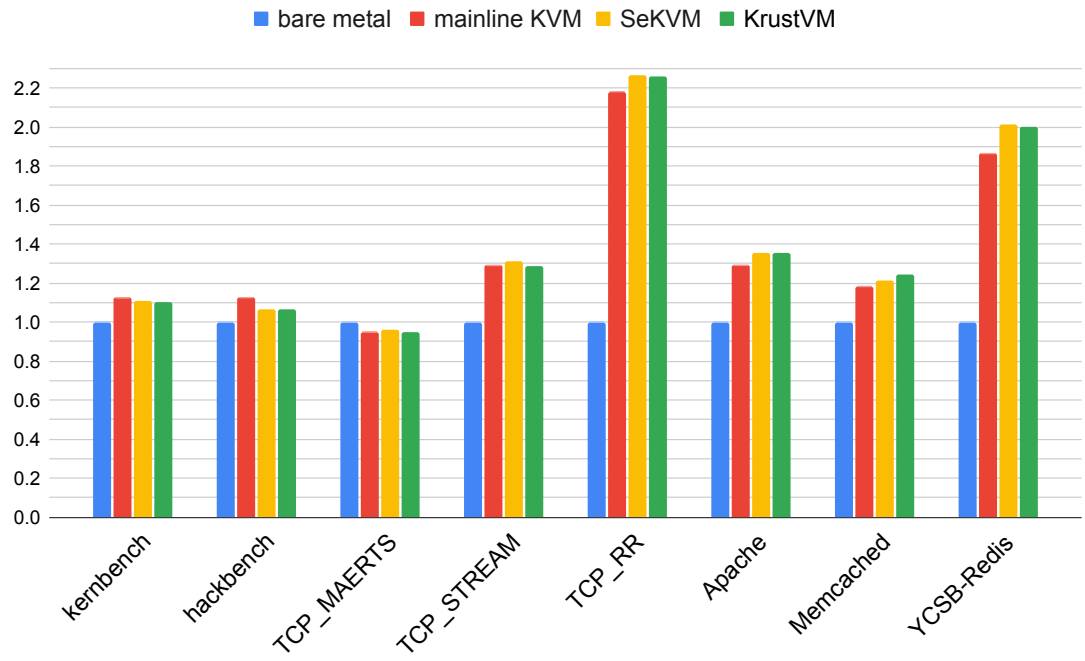

Figure 6.1: Application Benchmark Performance: Overhead normalized to the bare-metal setup

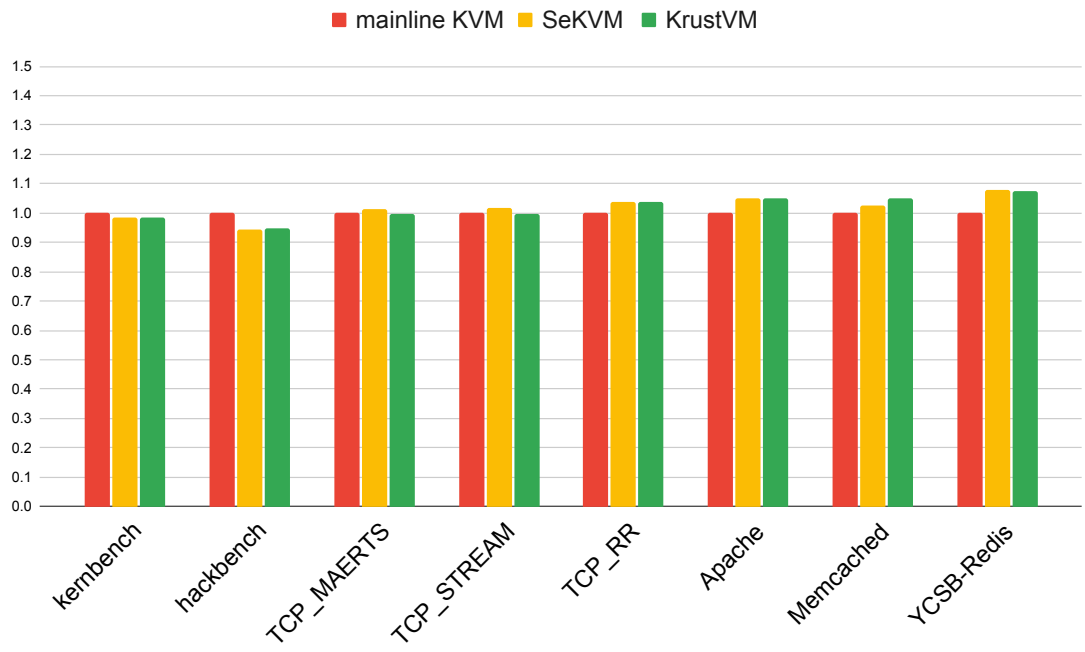

Figure 6.2: Application Benchmark Performance: Overhead normalized to mainline KVM

doi:10.6342/NTU202301822

Chapter 7 Related Work and Future Work

7.1 Related Work

7.1.1 VM Protection

Various previous work redesigned the hypervisor to protect VMs. Cloudvisor [38, 51] introduced a tiny security monitor underneath the commodity hypervisor to protect the hosted VMs. Twinvisor [33] supports regular VMs and confidential VMs by running hypervisors within both of ARM TrustZone's normal world and secure world. HypSec and pkvm [25, 34] reduced the hypervisor's resource access control component into to a small core to reduce the attack surface. Unlike our work, none of them used Rust to secure their hypervisor implementation. KrustVM and SeKVM [35] both leveraged HypSec's design [34] 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 imple-

doi:10.6342/NTU202301822

mentation took less than one person-year. Different from formal verification, because the Rust compiler automatically ensures memory safety properties, our hypervisor codebase is flexible to frequent updates.

7.1.2 Rust-based Systems

Recent work extended existing C/C++ systems with a Rust binding to enable a Rust-based programming environment. Rust-SGX [50] and RusTEE [49] 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 [18] 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 [45] rewrote a subset of QEMU's functionalities and separated them into libraries in Rust crates. Firecracker [2], crosvm [20], Cloud Hypervisor [37], and VMSH [47] 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 [26] 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.

7.1.3 Verification and Formal Methods

seL4 [29] presented the first machine-checked verification of an OS kernel. The total effort including code implementation, proof construction, and the related research took over 20 person years. CertiKOS [21] presented an extensible architecture for building cer-

doi:10.6342/NTU202301822

tified concurrent OS kernels. Their mC2 kernel is verified to be functionally correct and contains no bugs. Better than seL4, the mC2 kernel is capable of running on stock x86 machines with mutliple cores. Further, the assembly code in the kernel are also verified, and the CompCertX verified compiler is used for compilation, these combined make the mC2 kernel certified down to the assembly level. Overall the proof effort for certifing the mC2 kernel is 2 person years, and it consists of 6500 lines of C and x86 assembly. SeKVM [35] presented a secure and formally verified Linux KVM hypervisor. Different from seL4 and CertiKOS, which both verified less commonly used kernels, SeKVM verified the commodity Linux KVM hypervisor, which is widely deployed commercially. To achieve this, KVM is retrofitted into a small trusted core Kcore and the remaining untrusted services Kserv, and Kcore is proved to guarantee VM confidentiality and integrity. All of these work verified their code implementation by writing proofs to show that the code refines, or satisfies the specifications given. The specifications captures all the bahavior of the implementation. However, the high level specifications themselves may be potentially insecure. To mitigate this issue, SeKVM takes a step further and proves Kcore's specification upholds VM confidentiality and integrity by showing there can not exist any information leakage between VMs and Kserv, regardless of how they interact with Kcore's interface. In other words, the specifications of Kcore are guaranteed to exhibit the desired security properties. On the other hand, seL4 and CertiKOS does not address this issue.

These formally verified software systems are guaranteed to be free of programming errors, but achieving this comes at the cost of significant human effort and reduced flexibility for making changes. Although we hope that Rust can help us with reducing the effort needed to secure software without sacraficing the ability to update the codebase easily, it

doi:10.6342/NTU202301822

is challenging to solely rely on Rust to provide safety guarantees comparable to formal verification. Aside from the fact that there are still portions of unsafe Rust code within Rcore that the compiler cannot ensure memory safety for, logical errors such as setting the wrong permission bits for an NPT entry are possible in Rust. Rust verification tools [6, 17, 23, 30, 31] allows developers to annotate Rust code with specifications, invariants and assertions and then verify them formally, mathematically proving that the code satisfies the specifications. We can utilize these tools, which offer functional correctness guarantees, to eliminate logical errors and enhance the safety of Rcore.

7.2 Future Work

This thesis demonstrated how unsafe raw pointer accesses are protected through isolating memory regions, leveraging language features like automatic bound checks for array types, and type constructor that checks for their arguments. One aspect of this approach is that relying solely on memory region isolation is not enough to prevent bugs caused by raw pointers pointing to unintended addresses. Erroneous raw pointers are not detectable by memory region isolation if they reference an incorrect addresses within the correct region. Basically, the granularity of the protection is limited to the size of the memory regions. This limitation is similar to the hardware MMU, which cannot detect violations of sub-page access permissions because its protection granularity is at the page level.

In spite of the fact that memory region isolation alone can not guarantee the absence of raw pointers pointing to unintended addresses, a possible way of enhancing the protection given by memory region isolation is to shrink the sizes of the memory regions. The smaller granularity can increase the chances of detecting a raw pointer pointing to an

doi:10.6342/NTU202301822

unintended address. This can be done by by using the previously mentioned features even more extensively to enforce a separation of memory regions that are smaller in size, such as separating the NPT pools into the first level NPT pool, second level NPT pool, and so on, or splitting the Rcore metadata region into multiple isolated regions, such as `VMInfo` region, `PMemInfo` region, etc.

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

Chapter 8 Conclusions

We have presented KrustVM, a Rust-based secure KVM hypervisor that is based on Linux 5.15 and rewritten from the C-based Linux 4.18 SeKVM. Similar to SeKVM, KrustVM delivers VM confidentiality and integrity protection against an untrusted Linux host kernel integrated with KVM. We overcame challenges in forward porting SeKVM to Linux 5.15, and rewriting SeKVM's TCB in Rust. Furthermore, unsafe raw pointer accesses in KrustVM's TCB Rcore are protected by memory region isolation, which leveraged Rust's compile-time checks and its strong type system. These measures, combined with Rust's high-performance nature, enable KrustVM to ensure robust memory safety, accomodate frequent code modifications, and maintain overall efficiency. In our experiments, KrustVM preserves the performance efficiency of KVM, demonstrating the practicality for deployments.

doi:10.6342/NTU202301822

doi:10.6342/NTU202301822

References

[1] A. R. Adam Greig. aarch64-cpu rust crate. https://crates.io/crates/aarch64-cpu, 2023.

[2] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[3] 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.

[4] Arm. Arm and aws: Working together to " re:invent" the cloud. https://www.arm.com/company/news/2018/11/arm-and-aws-working-together-to-reinvent-the-cloud, 2018.

[5] Arm. Arm neoverse adopted by google cloud. https://www.arm.com/company/news/2022/07/arm-neoverse-adopted-by-google-cloud, 2022.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for

doi:10.6342/NTU202301822

modular specification and verification. Proc. ACM Program. Lang., 3(OOPSLA), oct 2019.

[7] M. Backes, G. Doychev, and B. Kopf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In 20th Annual Network and Distributed System Security Symposium (NDSS 2013), San Diego, CA, Feb. 2013.

[8] 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.

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

[10] 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.

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

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

[13] Y.-H. Chiang, W.-L. Chang, J.-T. Du, and S.-W. Li. Krustvm: a rust-based secure kvm hypervisor. https://github.com/ntu-ssl/linux-sekvm-rust, 2023.

[14] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. pages 143–154, 09 2010.

doi:10.6342/NTU202301822

[15] C. Dall and J. Nieh. Supporting kvm on the arm architecture. https://lwn.net/Articles/557132/, 2013.

[16] C. Dall and J. Nieh. 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, ASPLOS '14, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.

[17] X. Denis, J.-H. Jourdan, and C. Marché. Creusot: A foundry for the deductive verification of rust programs. In Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings, page 90–105, Berlin, Heidelberg, 2022. Springer-Verlag.

[18] R. for Linux Team. Rust for linux. https://rust-for-linux.com/, 2023.

[19] Google. Google Cloud Security and Compliance Whitepaper - How Google protects your data. https://static.googleusercontent.com/media/gsuite.google.com/en//files/google-apps-security-and-compliance-whitepaper.pdf, Sept. 2017.

[20] Google. Chromiumos virtual machine monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm/, 2023.

[21] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 653–669, Savannah, GA, Nov. 2016. USENIX Association.

doi:10.6342/NTU202301822

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008), pages 45–60, San Jose, CA, July 2008.

[23] S. Ho and J. Protzenko. Aeneas: Rust verification by functional translation. Proc. ACM Program. Lang., 6(ICFP), aug 2022.

[24] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015), pages 591–604, San Jose, CA, May 2015.

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

[26] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He. HyperEnclave: An open and cross-platform trusted execution environment. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 437–454, Carlsbad, CA, July 2022. USENIX Association.

[27] R. Jones. Netperf. https://github.com/HewlettPackard/netperf, June 2018.

[28] 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.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In Proceedings of the ACM SIGOPS 22nd

doi:10.6342/NTU202301822

Symposium on Operating Systems Principles, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[30] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang., 7(OOPSLA1), apr 2023.

[31] N. Lehmann, A. Geller, N. Vazou, and R. Jhala. Flux: Liquid types for rust, 2022.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015), pages 605–622, San Jose, CA, May 2015.

doi:10.6342/NTU202301822

[37] C. H. maintainers. Cloud hypervisor - run cloud virtual machines securely and efficiently. https://www.cloudhypervisor.org/, 2023.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In

doi:10.6342/NTU202301822

Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), pages 199–212, Chicago, IL, Nov. 2009.

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

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

[46] 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.

[47] J. Thalheim, P. Okelmann, H. Unnibhavi, R. Gouicem, and P. Bhatotia. Vmsh: Hypervisor-agnostic guest overlays for vms. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 678 – 696, New York, NY, USA, 2022. Association for Computing Machinery.

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

[49] 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.

[50] 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,

doi:10.6342/NTU202301822

CCS '19, page 2333–2350, New York, NY, USA, 2019. Association for Computing Machinery.

[51] 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.

[52] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012), pages 305–316, Raleigh, NC, Oct. 2012.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in Paas Clouds. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014), pages 990–1003, Nov. 2014.

doi:10.6342/NTU202301822