红叶：安全操作系统中的隔离与通信

Vikram Narayanan
University of California, Irvine

Tianjiao Huang
University of California, Irvine

David Detweiler
University of California, Irvine

Dan Appel
University of California, Irvine

Zhaofeng Li
University of California, Irvine

Gerd Zellweger
VMware Research

Anton Burtsev
University of California, Irvine

***Abstract*— 红叶（RedLeaf）是一个从零开始用 Rust 开发的新操作系统，旨在探索语言安全对操作系统组成的影响。与商业系统相比，红叶不依赖硬件地址空间进行隔离，而只依靠 Rust 语言的类型和内存安全。脱离昂贵的硬件隔离机制让我们得以探索轻量级细粒度隔离系统的设计空间。我们开发了一个新的基于语言的轻量级隔离域的抽象，它提供了一个信息隐藏和故障隔离的单元。域能够被动态地加载并干净地终止，即一个域中的错误不会影响其他域的运行。在红叶隔离机制的基础上，我们展示了实现设备驱动程序的端到端零拷贝、故障隔离和透明恢复的可能性。为了评估红叶抽象的实用性。我们把 Rv6，一个 POSIX 子集操作系统实现成了若干个红叶域的集合。最后，为了证明 Rust 和细粒度隔离的实用性，我们开发了 10Gbps 英特尔 ixgbe 网卡和 NVMe 固态硬盘设备驱动程序的高效版本，其性能可与最快的 DPDK 和 SPDK 相媲美。**

I. 引入

四十年前，早期的操作系统设计认为隔离内核子系统的能力是提高整个系统可靠性和安全性的关键机制[1], [2]。遗憾的是，尽管多次尝试在内核中引入细粒度隔离，但现代系统仍然是庞大且单一（monolithic）的。一直以来，软件和硬件机制对于高性能要求的子系统隔离来说过于昂贵。多个硬件项目探索了在硬件中实现细粒度、低开销隔离机制的能力 [3]–[5]。然而，着眼于性能，现代商品 CPU 只为用户程序的粗粒度隔离提供了基本支持。同样，几十年来，能在软件层面提供细粒度隔离的安全语言对于低级的操作系统代码来说开销仍然过高。传统上，安全语言需要一个可管理的运行时，特别是垃圾回收机制来实现安全。尽管垃圾回收技术取得了许多进步，但对于每个核心每秒处理数百万个请求的系统来说，垃圾回收的开销仍然很高（在典型的设备驱动程序工作情形中，最快的垃圾回收语言比 C 语言慢 20-50%[6]）。

几十年来，打破单一内核的设计选择仍然不切实际。因此，现代内核缺乏隔离性及其优势：简洁的模块化、信息隐藏、故障隔离、透明的子系统恢复和细粒度访问控制。

随着 Rust 的发展，隔离和性能之间的历史平衡正在发生变化。Rust 可以说是第一种在没有垃圾回收的情况下实现安全性的实用语言 [7]。Rust 将旧的线性类型 [8] 思想与实用的语言设计相结合，通过一个受限的所有权模型来实现类型和内存安全，允许内存中的每个生存对象只有唯一一个引用。这样就可以静态跟踪对象的生命周期，并在不使用垃圾回收器的情况下将其删除。该语言的运行时开销仅限于边界检查，且在许多情况下，现代超标量无序 CPU 可以隐藏边界检查，并预测和执行不进行检查时的正确路径 [6]。为了实现实用的非线性数据结构，Rust 提供了一小套精心挑选的基本类型，可以摆脱线性类型系统的严格限制。

Rust 作为一种开发底层系统的工具，迅速受到人们的欢迎，而传统上这些系统都是用 C 语言开发的[9]–[14]。低开销安全带来了一系列立竿见影的安全优势——预计三分之二由典型的不安全低级语言编程习语（idiom）引起的漏洞，仅通过使用安全语言就可以消除 [15]–[19]。

遗憾的是，最近的项目大多把 Rust 作为 C 语言的一种替换。然而，我们认为，语言安全的真正好处在于可以实现实用、轻量级、细粒度的隔离，以及一系列几十年来一直是操作系统研究重点但却无法实用的机制：故障隔离[20]、透明设备驱动恢复[21]、安全内核扩展[22], [23]、基于能力的细粒度访问控制[24] 等等。

红叶[1] 是一个新的操作系统，旨在探索语言安全对操作系统组成的影响，特别是在内核中利用细粒度隔离的能力及其优势。红叶是用 Rust 从零开始实现的。它不依赖硬件机制进行隔离，而只使用 Rust 语言的类型和内存安全。

尽管有多个项目在探索基于语言的系统中的隔离问题 [25]–[28] ，但在 Rust 中阐明隔离原则并提供实用的实现仍然具有挑战性。一般来说，安全语言会提供一些机制来控制对单个对象字段的访问（例如 Rust 中的 pub 访问修饰符），并保护指针，即限制对通过可见全局变量和显式传递参数可获取的程序状态的访问。通过对引用和通信渠道的控制，可以在函数和模块边界上隔离程序的状态，以确保保密性和完整性，更广泛地说，可以通过对象能力语言[29]所探索的一系列技术来构建全面的最小权限系统。

[1]在叶片组织中生成的锈(Rust)菌会使叶片变红

不幸的是，仅靠内置语言机制不足以对互不信任的系统实现隔离，如依赖语言安全隔离应用程序和内核子系统的操作系统内核。为了保护整个系统的执行，内核需要一种隔离故障的机制，即提供一种终止故障或行为不端计算的方法，使系统处于干净的状态。具体来说，在子系统终止后，隔离机制应提供途径来做到：1）取消分配给子系统使用的所有资源；2）如果一个子系统分配的对象已通过通信通道传递给其他子系统，保留它；以及 3）确保今后对已终止子系统所暴露接口的所有调用都不会违反安全规定或阻塞调用者，而是返回错误信息。面对基于语言的系统所鼓励的语义丰富的接口，故障隔离是一项挑战——频繁的引用交换往往意味着单个组件的崩溃会使整个系统处于损坏状态[28]。

从早期的单用户、单语言、单地址空间设计 [30]–[37] 到堆隔离[25], [26] 以及使用线性类型来实现堆隔离 [27]，多年来，在基于语言的系统中隔离计算的目标已经走过了漫长的道路。尽管如此，如今人们对基于语言的隔离原理还不甚了解。在 Sing# 中实现故障隔离的 Singularity [27] 依靠语言和操作系统的紧密协同设计来实现隔离机制。且最近几个提出使用 Rust 实现轻量级隔离的系统，如 Netbricks [38] 和 Splinter[12]，都难以阐明实现隔离的原则，而只是用 Rust 已经提供的信息隐藏来代替故障隔离。类似的，最近推出的 Rust 操作系统 Tock 通过传统的硬件机制和受限的系统调用接口支持用户进程的故障隔离，但却无法为安全 Rust 实现的设备驱动程序（胶囊）提供故障隔离[13]。

我们的工作是开发安全语言的故障隔离原则和机制。我们引入了一种基于语言的隔离域抽象，作为信息隐藏、加载和故障隔离的单元。为了封装域的状态并在域边界实现故障隔离，我们制定了以下原则：

- **堆隔离** 我们将堆隔离作为对所有域的约束（invariant），即各域绝不持有指向其他域私有堆的指针。堆隔离是终止和卸载崩溃域的关键，因为没有其他域持有指向崩溃域私有堆的指针，所以卸载整个私有堆是安全的。为了实现跨域通信，我们引入了一种特殊的共享堆，允许分配可在域之间交换的对象。
- **可交换类型** 为了实现堆隔离，我们引入了可交换类型的概念，即在不泄漏指向私有堆的指针的情况下，可以安全地跨域交换的类型。可交换类型允许我们静态地实现一个约束，即在共享堆上分配的对象不能有指向私有域堆的指针，但可以有指向共享堆上其他对象的引用。
- **所有权追踪** 为了取消崩溃域在共享堆上拥有的资源，我们会跟踪共享堆上所有对象的所有权。当一个对象在域之间传递时，我们会根据它是在域之间移动还是在只读访问中被借用来更新其所有权。我们依靠 Rust 的所有权规范来强制各域在跨域函数调用中传递共享对象引用时失去所有权，也就是说，Rust 会强制要求调用者域中没有被传递对象的别名。
- **接口验证** 为了提供系统的可扩展性，并允许域作者为他们实现的子系统定义自定义接口，同时保留隔离性，我们对所有跨域接口进行了验证，确保接口仅含有可交换类型的约束，从而防止它们破坏堆隔离约束。我们开发了一种接口定义语言（IDL），可静态验证跨域接口的定义并为其生成实现。
- **跨域调用代理** 我们通过代理来调用所有跨域调用——这是在所有域接口上插入的一层可信代码。代理可更新跨域传递对象的所有权，支持从崩溃的域中解绑（unwind）线程的执行，并在域终止后保护未来对域的调用。我们的 IDL 通过接口定义生成代理对象的实现

上述原则使我们能够以实用的方式实现故障隔离：即使面对语义丰富的接口，隔离边界也能将开销降至最低。当域崩溃时，我们会通过解绑当前在域内执行的所有线程的执行来隔离故障，并在不影响系统其他部分的情况下取消分配域的资源。对域接口的后续调用会返回错误类型，但仍然安全，不会引发恐慌。域分配的但在崩溃前返回的所有对象都会保持存活。

为了测试这些原则，我们将红叶实现成了一个微内核操作系统，在这个系统中，一组被隔离的域实现了内核的功能：典型的内核子系统、类 POSIX 接口、设备驱动程序和用户应用程序。红叶提供了现代内核的典型功能：多核支持、内存管理、动态加载内核扩展、类 POSIX 用户进程和快速设备驱动程序。在红叶隔离机制的基础上，我们展示了以透明方式恢复崩溃设备驱动程序的可能性。我们采用了一种与影子驱动程序类似的想法 [21]，即轻量级影子域，它作为访问设备驱动程序的中介，并在崩溃后重新启动它，重新执行其初始化协议。

为了评估红叶抽象的通用性，我们在红叶的基础上实现了 Rv6，这是一个 POSIX 子集操作系统。Rv6 遵循 UNIX V6 规范[39]。尽管 Rv6 是一个相对简单的内核，但它是一个很好的平台，说明了如何将基于语言的细粒度隔离思想应用于以 POSIX 接口为中心的现代内核中。最后，为了证明 Rust 和细粒度隔离带来的开销并不高昂，我们开发了高效版本的 10Gbps 英特尔 Ixgbe 网卡和 PCIe 附加固态硬盘 NVMe 驱动程序。

我们认为，将实用语言安全与所有权规范相结合，可以让我们首次以高效的方式实现操作系统研究中的许多经典理念。红叶速度快，支持内核子系统的细粒度隔离[40]–[42], [20]、故障隔离[21], [20]、实现端到端零拷贝通信[27]、支持用户级设备驱动程序和内核旁路[43]–[46] 等等。

II. 基于语言的系统隔离

在基于语言的操作系统中，对隔离的研究由来已久，这些研究通过语言安全、指针的细粒度控制和类型系统，探索执行轻量级隔离边界的取舍方法。早期的操作系统采用安全语言开发[30]–[37]。这些系统使用“开放”的架构，即单

用户、单语言、单地址空间的操作系统，模糊了操作系统与应用程序本身的界限[47]。这些系统依靠语言安全来防止意外错误，但不提供子系统或用户应用程序的隔离（现代 unikernel 采用类似方法 [48]–[50]）。

SPIN 是第一个提出将语言安全作为实现动态内核扩展隔离机制的[22]。SPIN 利用 Modula-3 指针来保证执行保密性和完整性，但由于指针是跨隔离边界交换的，因此无法提供故障隔离——一个崩溃的扩展使系统处于不一致状态。

J-Kernel[28] 和 KaffeOS [25] 是最早指出语言安全本身不足以确保隔离故障和终止不受信任的子系统这一问题的内核。为了支持 Java 中对孤立域的终止，J-Kernel 提出了对跨域共享的所有对象的访问进行中转（mediate）的想法[28]。J-Kernel 引入了一种特殊的功能对象，它封装了在孤立子系统中共享的原始对象的接口。为了支持域终止，由崩溃域创建的所有功能都会被撤销，从而丢弃对已被垃圾回收的原始对象的引用，并通过返回异常来阻止未来的访问。J-Kernel 依靠自定义类加载器来验证跨域接口（即在运行时生成远程指定代理，而不是使用静态 IDL 编译器）。为了实现隔离，J-Kernel 采用了一种特殊的调用约定，允许通过引用传递功能引用，但要求对常规的未封装对象进行深拷贝。由于没有共享对象的所有权约束，J-Kernel 提供的故障隔离模型有一定的局限性：当创建对象的域崩溃时，对共享对象的所有引用都会被撤销，从而将故障传播到通过跨域调用获得这些对象的域中。此外，由于缺乏 “移动 “语义，（即当对象被传递给被调用者时，调用者失去对该对象的访问权限），这意味着隔离需要对对象进行深拷贝，而这对于隔离现代高吞吐量设备驱动程序来说是非常困难的。

KaffeOS 不通过功能引用来中转对共享对象的访问，而是采用了 “写屏障”[51] 技术，这种技术会验证整个系统中的所有指针赋值，因此可以强制执行特定的指针规范[25]。KaffeOS 引入了私有域和特殊共享堆的分离，用于跨域共享对象——显式分离对于执行写屏障检查至关重要（如检查指针是否属于特定堆）。写入障碍用于执行以下约束：1）允许私有堆上的对象拥有指向共享堆上对象的指针，但是 2）共享堆上的对象被限制在同一个共享堆上。在跨域调用中，当共享对象的引用被传递到另一个域时，写入屏障被用来验证约束，并创建一对特殊的对象，负责共享对象的引用计数和垃圾回收。KaffeOS 有以下故障隔离模型：当对象的创建者终止时，其他域保留对该对象的访问权（引用计数确保最终对象在所有共享者终止时被回收）。遗憾的是，虽然其他域能够在对象创建者崩溃后访问这些对象，但这并不足以实现完全隔离——共享对象有可能处于不一致的状态（例如，如果对象更新到一半时发生崩溃），从而可能导致其他域停止或崩溃。与 J-Kernel 类似，隔离对象也需要在跨域调用时进行深拷贝。最后，中转所有指针更新的性能开销很大。

奇点操作系统引入了一种新的故障隔离模型，该模型是围绕静态执行的所有权规范建立的[27]。与 KaffeOS 类似，在奇点中，应用程序使用了隔离的私有堆和用于共享对象的特殊“交换堆”。一个开创性的设计是对交换堆上分配的对象强制执行单一所有权，即同一时间只能有一个域对共享堆上的对象进行引用。在跨域传递对象引用时，对象的所有权会在域之间 “移动”（将对象传递到另一个域后，试图访问该对象会被编译器拒绝）。奇点开发了一系列新颖的静态分析和验证技术，在有垃回收圾的 Sing# 语言中静态地确保这一属性。单一所有权是实现简洁实用的故障隔离模型的关键——崩溃的域无法影响系统的其他部分——不仅它们的私有堆被隔离，而且新的所有权规范还允许隔离共享堆，即崩溃的域无法触发其他域中共享引用的撤销，也无法让共享对象处于不一致的状态中。此外，单一所有权允许以零拷贝的方式实现安全隔离，即移动语义保证了对象的发送者将失去对该对象的访问权，因此允许接收者在知道发送者无法访问新状态或改变旧状态的情况下更新对象的状态。

在 J-Kernel、KaffeOS 和奇点的启示基础上，我们的工作制定了在安全语言中执行故障隔离的原则，这些原则可以确保所有权系统。与 J-Kernel 类似，我们采用了用代理封装接口的方法。不过，我们通过静态方式生成代理，以避免运行时开销。我们依赖与 KaffeOS 和奇点类似的堆隔离。我们采用堆隔离的主要原因是，可以在不了解堆内对象语义的情况下，回收域的私有堆。我们为共享堆上的对象借用了移动语义，以提供干净的故障隔离，同时支持来自奇点的零拷贝通信。不过，我们将其扩展为只读借用语义，以支持透明域恢复，同时不放弃零拷贝。由于红叶使用 Rust 实现，因此得益于其所有权规范，我们可以对共享堆上的对象执行移动语义。Rust 基于对线性类型[8]、仿射类型、别名类型[52], [53] 和基于区域的内存管理[54] 的大量研究，并受到 Sing# [55]、Vault [56] 和 Cyclone[57] 等语言的影响，在不影响语言可用性的前提下，静态地实现了所有权。与严重依赖于共同设计 Sing#[55] 及其通信机制的奇点不同，我们在 Rust 语言之外开发了红叶的隔离抽象——可交换类型、接口验证和跨域调用代理。这样，我们就能清楚地阐明提供故障隔离所需的最基本原则，并开发出一套独立于语言之外的机制来实现这些原则，可以说，这样就能使这些原则适应特定的设计取舍。最后，我们针对系统的实用性做出了几项设计取舍。我们针对最常见的“迁移线程”模型[58] 而非消息[27] 设计并实施了隔离机制，以避免在关键的跨域调用路径上出现线程上下文切换，并允许使用更自然的编程习语，例如，在红叶中，域接口只是 Rust trait。

III. 红叶架构

红叶采用微内核系统结构，依靠基于语言的轻量级域实现隔离(Figure 1)。微内核实现了启动执行线程、内存管理、域加载、调度和中断转发所需的功能。隔离域集合实现了设备驱动程序、操作系统个性（即 POSIX 接口）和用户应用程序（第 4.5 节）。由于红叶不依赖硬件隔离原语，因此所有域和微内核都运行在 0 环（ring 0）中。然而，域被

限制使用安全的 Rust（即微内核和可信库是红叶中唯一允许使用不安全 Rust 扩展的部分）。

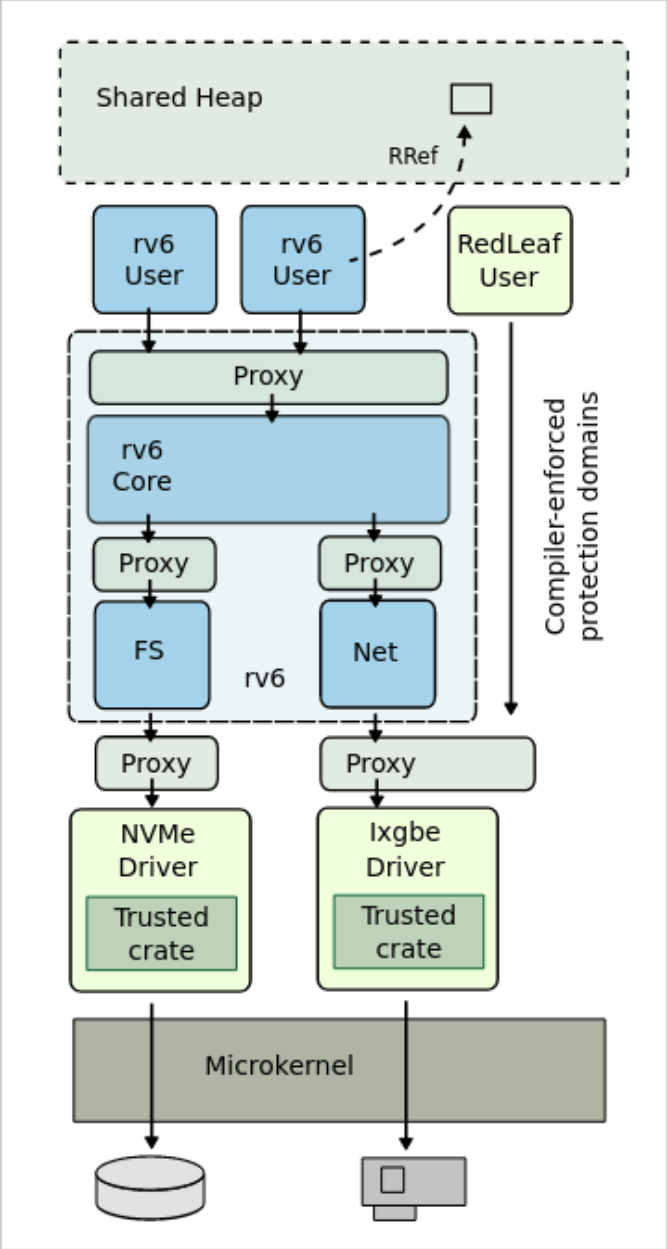

Figure 1: 红叶架构

我们在域之间强制执行堆隔离约束。为了进行通信，域从全局共享堆中分配可共享对象，并交换指向共享堆上分配的对象的特殊指针、远程引用（`RRef<T>`）（第 3.1 节）。所有权规则使我们能够实现跨隔离域的轻量级零拷贝通信（第 3.1 节）

域通过正常的、类型化的 Rust 函数调用进行通信。跨域调用时，线程会在域之间移动，但会在同一堆栈上继续执行。域开发人员为域的入口点及其接口提供接口定义。红叶 IDL 编译器会自动生成创建和初始化域的代码，并检查跨域边界传递的所有类型的有效性（第 3.1.5 节）。

红叶利用受信任的代理对象对所有跨域通信进行调解。代理由 IDL 编译器根据 IDL 定义自动生成（第 3.1.5 节）。在每个域入口处，代理都会检查域是否存活，如果存活，则会创建一个轻量级的继续(continuation)，以便在域崩溃时解绑线程的执行。

在红叶中，对对象和 trait 的引用就是能力。在 Rust 中，trait 声明了一个类型必须实现的一组方法，从而提供了一个接口的抽象。域通过跨域调用来交换对 trait 的引用，从而实现其功能。我们依靠基于功能的访问控制[24] 来执行最小特权原则，并实现灵活的操作系统组织：例如，我们实现了几种应用绕过内核直接与设备驱动程序对话的方案，甚至可以利用 DPDK 式的用户级设备驱动程序访问来链接设备驱动程序库。

保护模型 红叶背后的核心假设是，我们相信 (1) Rust 编译器能正确实现语言安全，以及 (2) 信任使用了不安全代码的 Rust 核心库，例如实现内部可变性的类型等。红叶的 TCB 包括微内核、实现硬件接口和底层抽象所需的少量可信红叶包、提供硬件资源安全接口的设备板块（如访问 DMA 缓冲区等）、红叶 IDL 编译器和红叶受信编译环境。目前，我们并不处理不安全 Rust 扩展中的漏洞，但我们再次推测，最终所有不安全代码都将通过功能正确性验证[59]–[61]。具体来说，RustBelt 项目提供了一个指南，以确保不安全代码被封装在安全接口中[62]。

我们相信设备是非恶意的。未来可以通过使用 IOMMU 来保护物理内存，从而放宽这一要求。最后，我们没有防范侧信道攻击；虽然这些攻击很重要，但解决它们超出了当前工作的范围。我们推测，未来的 CPU 将采用硬件对策来减少信息泄露[63]。

A. 域和错误隔离

在红叶中，域是信息隐藏、故障隔离和组合的单元。设备驱动程序、内核子系统（如文件系统、网络栈等）和用户程序都作为域加载。每个域的参数之一是微内核系统调用接口的引用。该接口允许每个域创建执行线程、分配内存、创建同步对象等。默认情况下，微内核系统调用接口是域的唯一权限，即域可以影响系统其他部分的唯一接口。不过，域可以为入口函数定义自定义类型，要求在创建时传递对象和接口的附加引用。默认情况下，我们不会为域创建新的执行线程。

不过，每个域都可以在加载域时通过微内核调用的 `init` 函数创建线程。在内部，微内核会跟踪代表每个域创建的所有资源：分配的内存、注册的中断线程等。线程的寿命可能超过创建它们的域，因为它们会进入其他域，并在那里无限期运行。这些线程会继续运行，直到它们返回到崩溃的域，并且该域是它们延续链中的最后一个域。

错误隔离 红叶域支持故障隔离。我们按以下方式定义故障隔离。我们说，当进入域的线程之一出现恐慌时，域就会崩溃并需要终止。恐慌可能会使域内可访问的对象处于不一致状态，从而使域内任何线程的进一步运行变得不切实

际（也就是说，即使线程没有发生死锁或恐慌，计算结果也是未定义的）。那么，如果以下条件成立，我们就可以说**故障被隔离**了。首先，我们可以将运行在崩溃域内的所有线程解绑到域入口点，并向调用者返回错误信息。其次，后续调用域的尝试会返回错误，但不会违反安全保证或导致恐慌。第三，崩溃域的所有资源都可以安全地回收，也就是说，其他域不会持有对崩溃域堆的引用（堆隔离约束），我们可以回收该域拥有的所有资源，而不会发生泄漏。第四，其他域中的线程会继续执行，并可继续访问由崩溃的域分配、但在崩溃前已转移到其他域的对象。

实施故障隔离具有挑战性。在红叶中，隔离的子系统输出复杂、语义丰富的接口，也就是说，域可以自由交换对接口和对象层次结构的引用。我们做出了几种设计选择，使我们能够干净利落地封装域的状态，同时支持语义丰富的接口和零拷贝通信。

1) 堆隔离与共享：

私有和共享堆：为了提供跨域的故障隔离并确保域的安全终止，我们要保证跨域堆隔离，即域的私有堆、堆栈或全局数据部分上分配的对象不能从域外访问。这个约束允许我们在执行的任何时刻安全地终止任何域。由于没有其他域持有指向已终止域的私有堆的指针，因此删除整个堆是安全的。

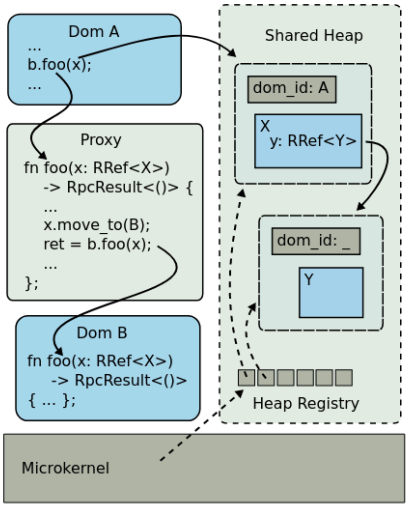

Figure 2: Inter-domain communication. Domain `A` invokes method `foo()` of domain `B`. The proxy that interposes on the invocation moves the ownership of the object pointed by `x` between domains.

Figure 2:

为了支持高效的跨域通信，我们为可跨域发送的对象提供了一个特殊的全局共享堆。域在共享堆上分配对象的方式与 Rust `Box<T>` 类型在普通堆上分配 `T` 类型的值类似。我们构建了一种特殊类型，即远程引用或 `RRef<T>` ，它可以在共享堆上分配 `T` 类型的值（Figure 2）。`RRef<T>` 由两部分组成：一个小的元数据和值本身。 `RRef<T>` 元数据包含当前拥有引用的域的标识符、借用计数器和值的类型信息。`RRef<T>` 元数据和值在共享堆上分配，允许 `RRef<T>` 在最初分配它的域之后继续存在。

私有堆中的内存分配：为了对域的私有堆进行封装，我们采用了两级内存分配方案。在底部，微内核为域提供了一个接口，用于分配无类型的粗粒度内存区域（大于一个页面）。每次粗粒度分配都会记录在**堆注册表**中。要在域的私有堆上进行精细类型分配，每个域都要与提供 Rust 内存分配接口 `Box<T>` 的可信包进行链接。域堆分配遵循 Rust 的所有权规则，即对象超出作用域时会被重新分配。两级方案有以下优点：只分配大内存区域，微内核记录域分配的所有内存，而不会产生显著的性能开销。如果该域发生恐慌，微内核会遍历分配给该域的分配器所分配的所有未类型化内存区域的注册表，并在不调用任何析构函数的情况下将其注销。这种无类型、粗粒度的回收是安全的，因为我们确保了堆隔离约束：其他域对回收的堆没有引用。

2) 可交换类型：

在共享堆上分配的对象必须遵守以下规则：它们只能由可交换类型组成。可交换类型确保以下约束：共享堆上的对象不能有指向私有堆或共享堆的指针，但可以有指向共享堆上分配的其他对象的 `RRef` 。红叶的 IDL 编译器在生成域的接口时会验证这一不变式（第 3.1.5 节）。我们将可交换类型定义为以下集合： 1） RRef 本身，2） Rust 原始 `Copy` 类型的子集，如 `u32`、`u64` ，但不包括一般情况下的引用 ，也不包括指针 ，3） 由可交换类型构建的匿名（元组、数组）和命名（枚举、结构体）复合类型 ，4） 对具有接收可交换类型方法的 `trait` 的引用。此外，所有 trait 方法都必须遵循以下调用约定，要求这些方法返回 `RpcResult<T>` 类型，以支持从崩溃域返回的线程的简洁中止语义（第 3.1 节）。IDL 会检查接口定义，并验证所有类型是否格式正确（第 3.1.5 节）。

3) 所有权追踪：

在红叶中，`RRef<T>` 可以在域之间自由传递。我们允许 `RRef<T>` 被移动或不可变借用。不过，我们对 `RRef<T>` 实施了所有权约束，在跨域调用时保证。通过所有权跟踪，我们可以安全地删除崩溃域所拥有的共享堆上的对象。`RRef<T>` 的元数据部分会记录所有者域和跨域调用时的借用次数。

最初，`RRef<T>` 由分配引用的域所有。如果在跨域调用中将引用转移到另一个域，我们就会更改 `RRef<T>` 中的所有者标识符，将所有权从一个域转移到另一个域。所有跨域通信都由可信代理进行中转，因此我们可以通过代理安全地更新所有者标识符。Rust 的所有权规范确保域内对象始终只有一个远程引用，因此当引用在跨域调用时在域之间移动时，调用者将失去对传递给被调用者的对象的访问权限。如果引用在跨域调用中被不可变借用，我们不会更改 `RRef<T>` 中的所有者标识符，而是递增跟踪 `RRef<T>` 被借用次数的计数器。

递归引用 `RRef<T>` 可以形成对象的层次结构。为了避免在跨域调用时递归移动层次结构中的所有 `RRef<T>` ，只有对象层次结构的根节点才有有效的所有者标识符 (Figure 2

中只有对象 X 有有效的域标识符 A，对象 Y 没有）。在跨域调用时，根 RRef<T> 会被代理更新，从而更改域标识符，在域之间移动 RRef<T> 的所有权。这就需要一个特殊的方案，以便在发生崩溃时取消 RRef<T> 的分配：我们扫描整个 RRef<T> 注册表，清理崩溃域所拥有的资源。为了防止删除层次结构中的子对象，我们的做法是，这些子对象没有有效的 RRef<T> 标识符（我们在扫描过程中跳过它们）。根 RRef<T> 对象的 drop 方法会遍历整个层次结构，并重新分配所有子对象（RRef<T> 不能形成循环）。请注意，我们应该小心处理 RRef<T> 从层次结构中移出的情况。为了正确地重新分配 RRef<T>，我们需要给它分配一个有效的域标识符，也就是说，当 Y 从 X 移出时，它就会得到一个正确的域标识符。我们用可信的访问器方法对 RRef<T> 字段分配进行中转。我们生成的访问器方法提供了从对象字段中取出 RRef<T> 的唯一方法。这样，我们就可以对移动操作进行中转，并更新被移动 RRef<T> 的域标识符。请注意，对于未命名的复合类型（如数组和元组），无法强制执行访问器。对于这些类型，我们会在跨越域边界时更新所有复合元素的所有权。

回收共享堆 通过所有权跟踪，我们可以回收当前由崩溃域拥有的对象的内存。我们对所有已分配的 RRef<T> 进行全局注册（Figure 2）。当域发生恐慌时，我们会浏览注册表，并删除崩溃域拥有的所有引用。如果 RRef<T> 被借用，我们会推迟回收，直到借用次数降为零。要取消每个 RRef<T> 的分配，我们必须为每种 RRef<T> 类型提供一个 drop 方法，并能动态识别引用的类型。每个 RRef<T> 都有一个由 IDL 编译器生成的唯一类型标识符（IDL 知道系统中的所有 RRef<T> 类型，因为它生成了所有跨域接口）。我们将类型标识符与 RRef<T> 一起存储，并调用适当的 drop 方法来正确地删除共享堆上任何可能的分层数据结构。

4) 跨域调用代理*:*

为了实现故障隔离，红叶依靠调用代理对所有跨域调用进行干预（Figure 2）。代理对象暴露的接口与其所中转的接口完全相同。因此，代理干预对接口用户来说是透明的。为确保隔离性和安全性，代理在每个封装函数中实现了以下功能：1）在执行调用之前，代理会检查域是否存活。如果域还存活，代理会通过更新微内核中的线程状态来记录线程在域之间移动的事实。当域崩溃时，我们会利用这一信息来解绑所有恰好在域内执行的线程。2）对于每次调用，代理都会创建一个轻量级的继续，以捕获跨域调用前的线程状态。该继续允许我们解绑线程的执行，并向调用者返回错误信息。3）代理在域之间移动作为参数传递的所有 RRef<T> 的所有权，或更新所有借用引用的借用计数，使其保持不变。4）最后，代理会封装所有作为参数传递的 trait 引用：代理会为每个 trait 创建一个新的代理，并传递该代理实现的 trait 的引用。

线程解绑 要从崩溃域解绑线程的执行，我们需要捕捉线程进入被调用者域之前的状态。对于代理所介导的每个 trait 函数，我们都会使用一个汇编蹦床（trampolime），将所有通用寄存器保存到一个**继续**中。微内核为每个线程维护一个继续栈。每个继续都包含所有通用寄存器的状态和一个指向错误处理函数的指针，该函数的签名与域接口导出的函数相同。如果必须解绑线程，我们会将堆栈恢复到继续所捕获的状态，并在相同的堆栈上以相同的通用寄存器值调用错误处理函数。错误处理函数会向调用者返回错误信息。

```
pub trait BDev {
  fn read(&self, block: u32, data: RRef<[u8; BSIZE]>)
        -> RpcResult<RRef<[u8; BSIZE]>>;
  fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>)
        -> RpcResult<()>;
}

#[create]
pub trait CreateBDev {
  fn create(&self, pci: Box<dyn PCI>)
    -> RpcResult<(Box<dyn Domain>, Box<dyn BDev>)>
}
```

Figure 3:

为了在崩溃时干净地返回错误，我们对所有跨域调用执行了以下调用约定：每个跨域函数都必须返回 RpcResult<T>，这是一种枚举类型，其中要么包含返回值，要么包含错误（Figure 3）。这样，我们就可以实现以下约束：从崩溃域中解绑的函数永远不会返回损坏的数据，而是返回一个 RpcResult<T> 错误。

5) 接口验证*:*

红叶的 IDL 编译器负责验证域接口，并生成在共享堆上执行所有权规范所需的代理代码。红叶 IDL 是 Rust 的一个子集，扩展了多个属性以控制代码的生成（Figure 3）。这种设计选择使我们能够为开发人员提供熟悉的 Rust 语法，并重新使用 Rust 的解析基础架构。

要实现接口的抽象，我们需要依赖 Rust 的 *trait*。trait 提供了一种定义方法集合的方法，类型必须实现这些方法才能满足 trait 的要求，从而定义了特定的行为。例如，BDev trait 要求任何提供该 trait 的类型实现两个方法：read() 和 write()（Figure 3）。通过交换对 trait 对象的引用，域可连接到系统的其他部分，并与其他域建立通信。

每个域都为创建 trait 提供了 IDL 定义，允许任何可访问该 trait 的域创建该类型的域（Figure 3）。标有 *#[create]* 属性的 create trait 既定义了域入口函数的类型，也定义了可用于创建域的 trait。具体来说，BDev 域的入口函数将 PCI trait 作为参数，并返回一个指向 BDev 接口的指针。请注意，当 BDev 域与 BDev 接口一起创建时，微内核也会返回域 trait，允许域的创建者以后对其进行控制。IDL 会生成用于创建该类型域的 create trait 和微内核代码的 Rust 实现。

接口检验 我们在 IDL 编译器的静态分析过程中进行接口验证。编译器首先会解析所有依赖的 IDL 文件，创建统一的抽象语法树 (AST)，然后将其传递到验证和生成阶段。在接口验证过程中，我们使用 AST 来提取每个验证类型的相关信息。从本质上讲，我们创建了一个图，其中编码了所

有类型的信息以及它们之间的关系。然后，我们使用该图验证每个类型是否可交换，以及是否满足所有隔离约束：跨域接口的方法是否返回 RpcResult<T> ，等等。

B. 零复制通信

结合 Rust 的所有权规范和共享堆上执行的单一所有权，我们可以提供隔离，而不会牺牲整个系统的端到端零拷贝。为了利用零拷贝通信，域使用 RRef<T> 类型在共享堆上分配对象。每次跨域调用时，域之间都会移动一个可变引用（可对对象进行可写访问的引用），或者借用一个不可变引用。如果调用成功，即被调用者域没有发生恐慌，那么被调用者可能会返回一组 RRef<T> ，将所有权转移给调用者。与 Rust 本身不同的是，我们不允许借用可变引用。借用可变引用可能会导致域崩溃时出现不一致的状态，因为受损对象会在线程解绑后返回给调用者。因此，我们要求移动和返回所有可变引用。显式返回。如果域崩溃，返回的不是引用，而是 RpcResult<T> 错误。

面对崩溃的域和提供透明恢复的要求，零拷贝具有挑战性。典型的恢复协议会重新启动崩溃的域，并重新发出失败的域调用，试图向调用者隐瞒崩溃情况。这通常要求在重新启动调用时作为参数传递的对象在恢复域内可用。可以在每次调用前为每个对象创建一个副本，但这样会带来很大的开销。为了在不增加副本的情况下恢复域，我们依赖于对跨域调用 RRef<T> 不可变借用的支持。例如，BDev 接口的 write() 方法借用了写入块设备的数据的不可变引用（Figure 3）。如果域借用了不可变引用，Rust 的类型系统会保证域不能修改借用的对象。因此，即使域崩溃，将未修改的只读对象返回给调用者也是安全的。作为恢复协议的一部分，调用者可以再次将不可变引用作为参数重新调用。这样就可以实现透明恢复，而无需在每次调用时创建可能会崩溃的参数备份。

IV. 实现

在引入一系列新颖抽象的同时，我们以实用性和性能为原则来指导红叶的设计。在某种程度上，红叶被设计为全功能商品内核（如 Linux）的替代品。

A. 微内核

红叶微内核提供了一个最小接口，用于创建和加载隔离域、执行线程、调度、底层中断调度和内存管理。红叶实现了与 Linux 类似的内存管理机制——buddy[64] 和 slab[65] 分配器的组合为微内核内部的堆分配提供了一个接口（Box<T> 机制）。每个域都在内部运行自己的分配器，并直接向内核 buddy 分配器请求内存区域。

我们用汇编实现底层中断进入和退出代码。虽然 Rust 提供了对 x86 中断函数 ABI 的支持（这是一种将 x86 中断堆栈帧作为参数来编写 Rust 函数的方法），但在实践中并没有什么用处，因为我们需要对中断的进入和退出进行干预，例如保存所有 CPU 寄存器。

在红叶中，设备驱动程序是在用户域中实现的（微内核本身不处理除定时器和 NMI 之外的任何设备中断）。域将线程注册为中断处理程序，以处理设备产生的中断。对于每个外部中断，微内核会维护一个等待中断的线程列表。当收到中断时，这些线程会被放回调度程序运行队列。

B. 动态加载域

在红叶中，域的编译与内核无关，而是动态加载的。动态加载。Rust 本身并不支持动态扩展（除 Splinter[12] 外，现有的 Rust 系统会静态链接其执行的所有代码[66], [13], [38]）。从概念上讲，动态扩展的安全性依赖于以下约束：跨越域边界的所有数据结构的类型 ,包括入口函数的类型，以及通过入口函数可到达的任何接口传递的所有类型，在整个系统中都是相同的 ,即具有相同的含义和实现。这样，即使系统的各个部分是单独编译的，也能确保跨域边界的类型安全保证得以保留。

为了确保类型在系统的所有组件中具有相同的含义，红叶依赖于可信的编译环境。该环境允许微内核检查域是否根据相同版本的 IDL 接口定义、相同的编译器版本和标志进行编译。在编译域时，受信任环境会签署包含所有 IDL 文件的指纹和一串编译器标志。微内核会在加载域时验证域的完整性。此外，我们还强制要求域只能使用安全的 Rust 代码，并与白名单上的 Rust 库进行链接。

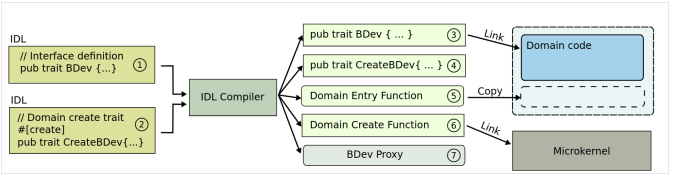

Figure 4:

代码生成 域的创建和加载依赖于 IDL 编译器生成的代码（Figure 4）。IDL 可确保域边界的安全性，并支持用户定义的域接口。根据域接口定义（Figure 4, 1）及其创建函数（2），IDL 生成以下代码：1）所有接口的 Rust 实现（3）和创建（4）特性，2）可信入口点函数（5）被置于域的构建树中，与域的其他部分一起编译，以确保域的入口函数与域的创建代码相匹配，从而维护域边界的安全，3）微内核域创建函数，用于创建具有入口点函数特定类型签名的域（6）；以及 4）该接口的代理实现（7）。通过控制入口点的生成，我们确保微内核内部和域内部的入口函数类型相匹配。如果域试图通过改变入口函数的类型来违反安全性，编译就会失败。

C. 安全设备驱动

在红叶中，设备驱动程序是作为普通域实现的，没有额外的权限。与其他域一样，它们也仅限于 Rust 的安全子集。为了访问硬件，我们为设备驱动程序提供了一系列可信包，这些包实现了设备硬件接口的安全接口，例如访问设备寄存器及其 DMA 引擎。例如，ixgbe 设备板块提供了对设备 BAR 区域的访问，并将其提交队列和接收队列抽象为用于从缓冲区添加和删除请求的方法集合。

设备驱动域由 `init` 域在系统启动时创建。每个 PCI 设备都会引用 `pci` 域内实现的 PCI trait。与其他驱动域类似，PCI 驱动程序依赖于可信包来枚举总线上的所有硬件设备。可信设备包构建的 `BARAddr` 对象包含 PCI BAR 区域的地址。我们使用自定义类型保护每个 `BARAddr` 对象，因此它只能在实现访问特定 BAR 区域的可信设备板块内使用。`pci` 域会探测具有匹配设备标识符的设备驱动程序。驱动程序接收到 `BARAddr` 对象的引用，并开始通过其可信包访问设备。

D. 设备驱动恢复

轻量级隔离机制和简洁的域接口使我们能够利用影子驱动程序实现透明的设备驱动程序恢复[21]。我们将影子驱动程序开发为普通的无特权红叶域。与代理对象类似，影子驱动程序封装了设备驱动程序的接口，并公开一个相同的接口。相比之下，代理相对简单，可以从 IDL 定义中生成，而影子驱动程序则更需要才智，因为它实现了特定于驱动程序的恢复协议。影子驱动程序会中断与驱动程序的所有通信。在正常运行期间，影子驱动程序会将所有调用传递给真正的设备驱动程序。不过，它会保存恢复驱动程序所需的所有信息（例如，对 PCI 接口的引用，以及设备初始化协议的其他部分）。如果驱动程序崩溃，影子驱动程序会收到来自代理域的错误信息。当线程通过继续机制解绑时，代理本身也会收到错误信息。影子不会向调用者返回错误，而是触发域恢复协议。它会创建一个新的驱动程序域，并通过中断驱动程序的所有外部通信，重新执行其初始化协议。

E. *Rv6* 操作系统特性

为了评估红叶抽象的通用性，我们在红叶的基础上实现了 Rv6，一个 POSIX 子集操作系统。在高层次上，Rv6 遵循 xv6 操作系统的实现方法[67]，但它是作为一个孤立的红叶域集合来实现的。具体来说，我们将 Rv6 实现为以下域：核心内核、文件系统、网络栈子系统、网络和磁盘设备驱动程序以及用户域集合。用户域通过 Rv6 系统调用接口与核心内核通信。核心内核会将系统调用分配给文件系统或网络堆栈。文件系统本身会与其中一个红叶块设备驱动程序通信，以获取磁盘访问权限。我们实现了三种块设备驱动程序：内存、AHCI 和 NVMe。文件系统实现了日志、缓冲缓存、inode 和命名层。网络子系统实现了 TCP/IP 协议栈并连接到网络设备驱动程序（我们目前只实现了一个支持 10Gbps 英特尔 Ixgbe 设备的驱动程序）。我们不支持 fork() 系统调用的全部语义，因为我们不依赖地址空间，因此无法虚拟和克隆域的地址空间。相反，我们提供了创建系统调用的组合，允许用户应用程序加载和启动新域[68]。Rv6 启动后会进入一个支持管道和 I/O 重定向的 shell，并能启动与典型 UNIX 系统类似的其他应用程序。

V. 评估

我们在公开的 CloudLab 网络测试平台上进行所有实验[69]。[2]对于基于网络的实验，我们使用了两台 CloudLab c220g2 服务器，配置了两个主频为 2.6 GHz 的英特尔 E5-2660 v3 10 核 Haswell CPU、160 GB 内存和一个双端口英特尔 X520 10Gb 网卡。我们在 CloudLab d430 节点上运行 NVMe 性能测试，该节点配置了两个 2.4 GHz 64 位 8 核 E5-2630 Haswell CPU 和一个 PCIe 连接的 400GB 英特尔 P3700 系列固态硬盘。Linux 机器运行 64 位 Ubuntu 18.04，内核配置为 4.8.4，没有任何投机执行攻击（speculative execution attack）缓解措施，因为最近的英特尔 CPU 在硬件上解决了一系列投机执行攻击问题。所有红叶实验均在裸机硬件上进行。在所有实验中，我们禁用了超线程、涡轮增压、CPU 空闲状态和频率缩放，以减少性能测试中的差异。

A. 域隔离的开销

Operation	Cycles
seL4	834
VMFUNC	169
VMFUNC-based call/reply invocation	396
RedLeaf cross-domain invocation	124
RedLeaf cross-domain invocation (passing an `RRef<T>`)	141
RedLeaf cross-domain invocation via shadow	279
RedLeaf cross-domain via shadow (passing an `RRef<T>`)	297

Figure 5:

基于语言的隔离对比硬件机制 为了了解基于语言的隔离与传统硬件机制相比有哪些优势，我们将红叶的跨域调用与 seL4 微内核实现的同步 IPC 机制[70]，以及最近利用基于 VMFUNC 的扩展页表（EPT）切换的内核隔离框架[42]进行了比较。我们选择 seL4，因为它在多个现代微内核中实现了最快的同步 IPC [71]。我们在配置 seL4 时没有使用熔断缓解措施。在 c220g2 上，服务器 seL4 的跨域调用延迟为 834 个周期（Figure 5）。

最近，英特尔 CPU 引入了两个新的硬件隔离原语——内存保护密钥（MPK）和 EPT 与虚拟机功能的切换——为内存隔离提供支持，开销与系统调用相当[72] (MPK 为 99-105 个周期[73], [72]，VMFUNC 为 268-396 个周期[73], [71], [42], [72])。遗憾的是，这两种基元(primitives)都需要复杂的机制来实施隔离，例如二进制重写 [71], [72]、硬件断点保护[73]、在管理程序的控制下执行[74], [71], [42] 等。此外，由于 MPK 和 EPT 交换在设计上都不支持特权 0 环代码的隔离，因此需要额外的技术来确保内核子系统的隔离[42]。

为了比较基于 EPT 的隔离技术与红叶中使用的基于语言的技术的性能，我们配置了 LVDs（一种最新的基于 EPT 的内核隔离框架）[42]，以执行 1000 万次跨域调用，并使用 RDTSC 指令测量以周期为单位的延迟。在 LVD 中，跨域调用依靠 VMFUNC 指令切换 EPT 的根，并在被调用域中选择新的栈。然而，LVD 无需额外切换权限级别或页表。

[2]红叶链接：https://mars-research.github.io/redleaf

在 c220g2 服务器上，一条 VMFUNC 指令需要 169 个周期，而一次完整的调用/回复调用需要 396 个周期（Figure 5）。

在红叶中，跨域调用是通过调用代理域提供的 trait 对象来启动的。代理域使用微内核系统调用将线程从被调用者域转移到调用者域，创建继续以在调用失败时将线程解绑到入口点，并调用被调用者域的 trait。在返回路径上，类似的序列会将线程从被调用者域移回调用者域。在红叶中，通过代理对象进行的空跨域调用（Figure 5）会带来 124 个周期的开销。保存线程状态（即创建继续）需要 86 个周期，因为它需要保存所有普通寄存器。传递一个 `RRef<T>` 会增加 17 个周期的开销，因为 `RRef<T>` 会在域之间移动。为了了解透明恢复的底层开销，我们测量了通过影子域执行相同调用的延迟。在影子域的情况下，调用要跨越两个代理和一个用户自建的影子域，由于代理域和影子域的额外跨越，需要 286 个周期。

大多数最新的英特尔 CPU 都支持内存保护密钥的 0 环执行，即保护密钥监督器（PKS）[75]，最终实现了特权内核代码的低开销隔离机制。不过，即使有了低开销的硬件隔离机制，零拷贝故障隔离方案也需要对共享对象的所有权进行规范，而这需要编程语言的支持，即静态分析[27] 或可强制执行单一所有权的类型系统。

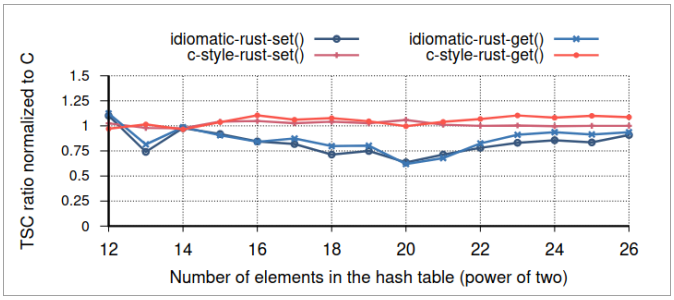

Figure 6:

Rust 的开销 Rust 的内存安全保证是有代价的。除了需要在运行时进行检查以确保安全外，一些 Rust 抽象还需要付出非零的运行时代价，例如实现内部可变性的类型、选项类型等。为了测量 Rust 语言本身带来的开销，我们开发了一个简单的哈希表，该表使用开放式寻址方案，并依靠带有线性探测功能的 Fowler-Noll-Vo (FNV) 哈希函数来存储 8 个字节的键和值。使用相同的散列逻辑，我们开发了三种实现：1）纯 C 语言；2）规范 Rust 语言（Rust 编程手册鼓励的风格）；3）C 风格 Rust 语言，本质上使用 C 语言编程习惯，但使用 Rust 语言。具体来说，在 C 风格 Rust 中，我们避免了：1）使用高阶函数；2）使用 `Option<T>` 类型（我们在规范 Rust 代码中使用该类型来区分表中已占用和未占用的条目）。如果没有 `Option<T>` 类型为键值对增加至少一个额外的字节，我们就可以在内存中对键值对进行紧密的缓存对齐表示，从而避免额外的缓存缺失。我们令哈希表中的条目数从 2^12^ 到 2^26^ 不等，并保持哈希表 75% 满。在大多数哈希表大小的情况下，我们的规范 Rust 实现仍然比纯 C 语言慢 25%，而 C 风格 Rust 的性能与纯 C 语言相当，甚至更好，尽管只是 3-10 个周期（Figure 6）。我们将此归功于 Rust 编译器生成的代码更加紧凑（C-style Rust 中关键 get/set 路径上有 47 条指令，而 C 中有 50 条指令）。

B. 设备驱动

红叶背后的一个关键假设是，Rust 的安全性适用于开发现代操作系统内核中速度最快的子系统。如今，每个 I/O 请求的延迟时间低至数百个周期，在所有内核组件中，提供访问高吞吐量 I/O 接口、网络适配器和低延迟非易失性 PCIe 附加存储的设备驱动程序的性能预算最为紧张。为了了解 Rust 零成本抽象的开销是否允许开发此类低开销子系统，我们开发了两个设备驱动程序：1）英特尔 82599 10Gbps 以太网驱动程序（Ixgbe）；2）用于 PCIe 附加固态硬盘的 NVMe 驱动程序。

1) *Ixgbe* 网卡驱动:

我们将 RedLeaf Ixgbe 驱动程序的性能与 在 Linux 上高度优化的 DPDK 用户空间数据包处理框架[44] 驱动程序的性能进行了比较。DPDK 和我们的驱动程序都在轮询模式下工作，从而使它们达到最高性能。我们令红叶运行几种不同配置：1）`redleaf-driver`：基准应用程序与驱动程序静态链接（这种配置最接近 DPDK 等用户级数据包框架；同样，我们将 Ixgbe 接口直接传递给红叶）；2）`redleaf-domain` :基准应用程序在单独的域中运行，但通过代理直接访问驱动程序域（这种配置代表了多个孤立应用程序共享网络设备驱动程序的情况[73]）；3）`rv6-domain`：基准应用程序作为 Rv6 程序运行，它首先通过系统调用进入 Rv6，然后调用驱动程序（这种配置类似于商品操作系统内核的设置，其中用户应用程序通过内核网络堆栈访问 I/O 接口）。此外，我们还运行了有影子驱动程序和无影子驱动程序的后两种配置（`redleaf-shadow` 和 `rv6-shadow`），这两种配置会引入额外的跨域来使用影子域（这两种配置会评估透明驱动程序恢复的开销）。在所有测试中，我们都将应用线程固定在单个 CPU 内核上。

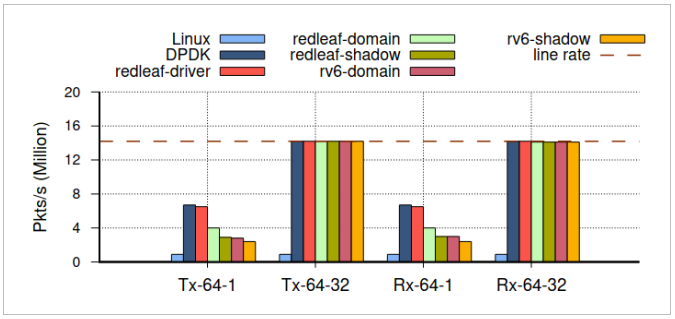

Figure 7:

我们发送 64 字节数据包，并测量两种批量大小的性能：1 个和 32 个数据包（Figure 7）。在数据包接收测试中，我们使用 DPDK 框架中的快速数据包生成器以线速(line-rate)生成数据包。在数据包发送和接收测试中，由于 Linux 的网络协议栈和同步套接字接口过于通用，其速度仅为 0.89

Mpps（Figure 7）。在一个批次上，DPDK 达到了 6.7 Mpps，在 RX 和 TX 路径上都比红叶（6.5 Mpps）快 7%（Figure 7）。在批量处理 32 个数据包时，两个驱动程序都达到了万兆以太网接口的线速性能（14.2 Mpps）。为了了解跨域调用的影响，我们将基准应用程序作为单独的域（`redleaf-domain`）和 Rv6 程序（`rv6-domain`）运行。跨域的开销在批量为 1 的情况下非常明显，红叶在进行一次跨域（`redleaf-domain`）时，每个内核收发数据包的速度为 4 Mpps，如果调用涉及影子域（redleaf-shadow），则为 2.9 Mpps。在两次跨域的情况下，性能下降到 2.8 Mpps（`rv6-domain`），如果通过影子（`rv6-shadow`）访问驱动程序，性能下降到 2.4 Mpps。在 32 个数据包的批次中，由于所有配置都会使设备达到饱和，因此跨域的开销就会消失。

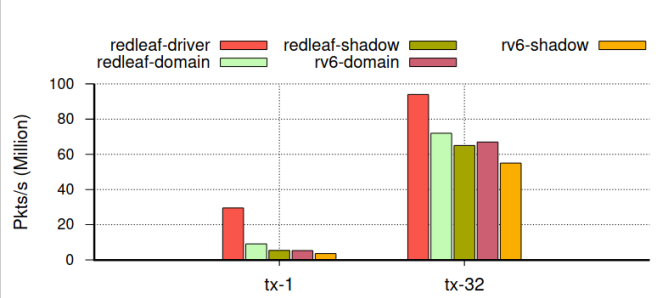

Figure 8:

Nullnet 为了进一步研究不受设备本身限制的隔离开销，我们开发了一个纯软件 `nullnet` 驱动程序，它只需将数据包返回给调用者，而不是将其队列到设备上（Figure 8）。在 1 个数据包的测试中，多次跨域的开销限制了 nullnet 驱动程序的理论性能。如果应用程序与驱动程序静态链接（`redleaf-driver`），则 `nullnet` 驱动程序的理论性能将从每核 29.5 Mpps 降至从 Rv6 应用程序访问 `nullnet` 时的 5.3 Mpps（`rv6-domain`）。添加阴影驱动程序后，这一数字将降至 3.6 Mpps（`rv6-shadow`）。同样，如果在与驱动程序相同的域中运行应用程序，在 32 个数据包的情况下，`nullnet` 可达到 94 Mpps。当基准代码作为 Rv6 应用程序（`rv6-domain`）运行时，性能下降到 67 Mpps，而当 Rv6 应用程序涉及影子驱动程序（`rv6-shadow`）时，性能下降到 55 Mpps。

2) *NVMe* 驱动:

为了解红叶 NVMe 驱动程序的性能，我们将其与 Linux 内核中的多队列块驱动程序和 SPDK 存储框架中经过优化的 NVMe 驱动程序进行了比较[45]。SPDK 和红叶驱动程序都在轮询模式下工作。与 Ixgbe 类似，我们评估了几种配置：1）静态链接（`redleaf-driver`）；2）需要一次跨域（`redleaf-domain`）；和 3）作为 Rv6 用户程序（`rv6-domain`）运行。我们在有影子驱动程序（`redleaf-shadow` 和 `rv6-shadow`）和没有影子驱动程序的情况下运行后两种配置。所有测试都仅限于单 CPU 内核。

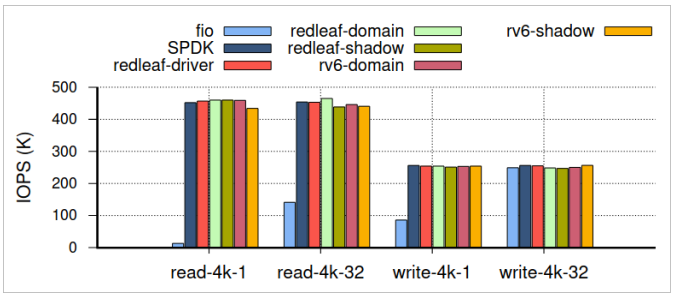

Figure 9:

我们在 1 个和 32 个请求的批量大小上执行块大小为 4KB 的连续读写测试（Figure 9）。在 Linux 上，我们使用快速 I/O 生成器 `fio`；在 SPDK 和红叶上，我们开发了类似的基准应用程序，一次性提交一组请求，然后轮询已完成的请求。为了给我们的评估设定一个最佳基线，我们选择了能让我们以最快速度访问设备的配置参数。具体来说，在 Linux 上，我们将 `fio` 配置为使用异步 `libaio` 库来重叠 I/O 提交，并使用直接 I/O 标志绕过页面缓存。

在顺序读取测试中，Linux 上的 `fio` 在批量大小为 1 和 32 的情况下，每个内核分别达到 13K IOPS 和 141K IOPS（Figure 9）。在批量为 1 的情况下，红叶驱动程序（每核 457K IOPS）比 SPDK（每核 452K IOPS）快 1%。两种驱动程序都能实现最高的设备读取性能。SDPK 的速度较慢，因为它会执行额外的处理，目的是收集每个请求的性能统计数据。在批量大小为 32 的情况下，红叶驱动程序的速度慢了不到 1%（453K IOPS 对比 454K IOPS SPDK）。在批量大小为 32 的连续写入测试中，Linux 与设备的最大吞吐量（约 256K IOPS）相差不到 3%。红叶的速度慢不到百分之一（255K IOPS）。由于 NVMe 与 Ixgbe 相比是一种速度较慢的设备，因此对于两种批量大小的数据而言，跨域的开销都很小。在跨越一个域时，性能甚至提高了 0.7%（我们将其归因于访问设备门铃寄存器的模式不同，设备和 CPU 之间会发生冲突）。

C. 应用性能测试

为了解安全和隔离对应用工作负载的性能影响，我们开发了几款传统上依赖于操作系统内核快速数据传输的应用软件：1) Magleb 负载平衡器（`maglev`）[76]；2) 带有网络功能的键值存储（`kv-store`）；3）最小网络服务器（`httpd`）。

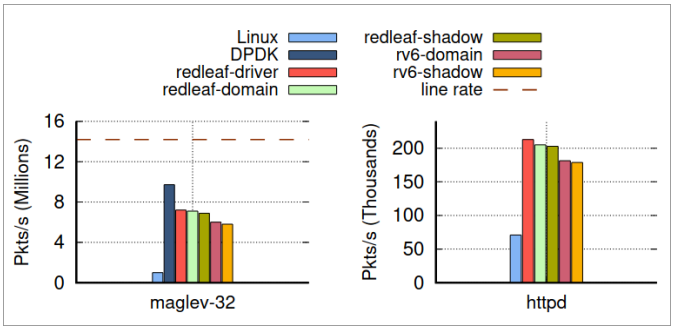

Figure 10:

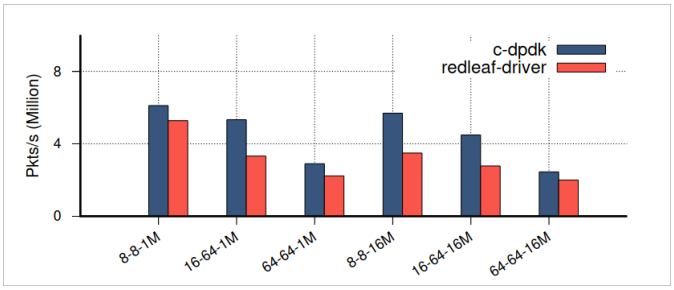

Figure 11:

Maglev 负载平衡器 Maglev 是谷歌开发的一种负载平衡器，用于在一组后端服务器之间平均分配进入的客户端流量[76]。对于每个新流量，Maglev 会在哈希表中进行查找，从可用的后端服务器中选择一个，哈希表的大小与后端服务器的数量成正比（在我们的实验中为 65537）。一致的散列允许流量在所有服务器上均匀分布。然后，Maglev 将所选的后端服务器记录在哈希表（即流量跟踪表）中，用于将同一流量的数据包重定向到同一后端服务器。流量跟踪表的大小与流量的数量成正比（我们在实验中选择了 1M 流量）。处理数据包时，如果数据包是现有流量，则需要在流量跟踪表中查找，或者查找后端服务器并插入流量跟踪表以记录新流量。为了比较红叶的性能，我们开发了 C 语言和 Rust 语言版本的 Meglev 核心逻辑。此外，我们还评估了两个 C 语言版本：一个是作为使用套接字接口的普通 Linux 程序运行，另一个是作为 DPDK 网络处理框架[44] 的网络功能开发。在所有版本中，我们都使用相同代码逻辑，并尽可能采用相同的优化。同样，在所有设置中，我们都限制在一个 CPU 内核上执行。作为 Linux 程序运行时，由于 Linux 内核的同步套接字接口和通用网络协议栈，`maglev` 的单核速度限制在 1 Mpps（Figure 10）。`maglev` DPDK 功能在 32 个数据包的批次上运行，由于网络设备驱动程序优化良好，每个内核能够达到 9.7 Mpps 的速度。通过与驱动程序静态链接，红叶应用程序（`redleaf-driver`）实现了每内核 7.2 Mpps 的速度。随着跨域次数的增加，性能有所下降。作为 Rv6 应用程序运行时，Maglev 在不使用影子域的情况下，每个内核的转发速度为 5.3 Mpps，使用影子域时为 5.1 Mpps。

键值储存 从社交网络[77] 到键值数据库[78]，键值存储是一系列数据中心系统事实上的标准构件。为了评估红叶支持高效数据中心应用开发的能力，我们开发了网络附加键值存储 `kv-store` 的原型。我们的原型在设计上采用了一系列与 Mica[79] 类似的现代优化技术，例如，像 DPDK 这样的用户级设备驱动程序、旨在避免跨核缓存一致性流量的分区设计、保证请求直接指向存储密钥的特定 CPU 内核的数据包流转向、请求处理路径上的无锁和无分配等。我们的实现依赖于一个哈希表，它使用线性探测的开放寻址方案和 FNV 哈希函数。在实验中，我们比较了两种实现的性能：一种是为 DPDK 开发的 C 语言版本，另一种是与驱动程序（`redleaf-driver`）在同一域中执行的 Rust 版本，即最接近 DPDK 的配置。我们评估了两种哈希表大小： 1M 和 16M 条目，三组键值对（`<8B,8B>`, `<16B,64B>`, `<64B,64B>`）。红叶版本是用 C 风格的 Rust 代码实现的，也就是说，我们避免了有运行时开销的 Rust 抽象（如 `Option<T>` 和 `RefCell<T>` 类型）。这确保我们可以控制键值对的内存布局，避免额外的缓存缺失。尽管我们进行了优化，但红叶的性能仅为 C DPDK 版本的 61-86%。性能下降的主要原因是我们的代码使用安全 Rust 中的变长数组（`Vec<T>`）来表示数据包数据。要创建一个响应，我们需要通过调用 `extend_from_slice()` 函数将响应头、键和值复制到响应数据包中，从而对该变长数组进行三次扩展。该函数会检查变长数组是否需要增长，并执行复制。相比之下，C 语言的实现则得益于对 `memcpy()` 的不安全调用。作为练习，我们使用不安全的 Rust 类型转换实现了数据包序列化逻辑，使我们的性能达到了 C 语言的 85-94%。不过，我们不允许在红叶域内使用不安全的 Rust。

网络服务器 网页加载的延迟对用户体验和搜索引擎分配的网页排名都起着至关重要的作用[80], [81]。我们开发了一个可提供静态 HTTP 内容的网络服务器 `httpd` 原型。我们的原型使用一种简单的“运行——完成”执行模式，以轮询方式从所有打开的连接中轮询传入的请求。对于每个请求，它都会执行请求解析，并回复所请求的静态网页。我们 我们将我们的实现与事实上的行业标准网络服务器之一 Nginx[82] 进行了比较。在测试中，我们使用了 `wrk` HTTP 负载生成器[83]，并将其配置为以一个线程和 20 个开放连接运行。在 Linux 上，Nginx 每秒可处理 70.9 K 个请求，而在应用程序与驱动程序（`redleaf-driver`）和网络协议栈运行于同一域的配置下，我们的 `httpd` 实现每秒可处理 212 K 个请求（Figure 10）。具体来说 我们受益于对网络协议栈和网络设备驱动程序的低延迟访问。作为 Rv6 域运行时，`httpd` 实现了每秒 181.4 K 个数据包的速率（如果使用影子域，则为 178.9 K）。

D. 设备驱动恢复

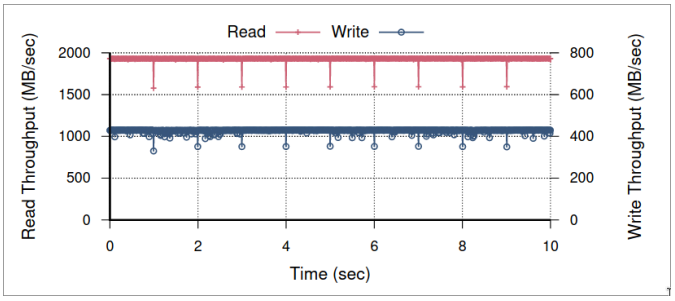

Figure 12:

为了评估透明设备驱动程序恢复带来的开销，我们开发了一个测试，其中 Rv6 程序访问由内存块设备支持的 Rv6 文件系统。作为 Rv6 程序运行时，基准应用程序使用 4K 块连续读写 Rv6 文件系统中的文件。Rv6 文件系统通过影子驱动程序访问块设备，该驱动程序可在块设备崩溃时执行

恢复。测试期间，我们每秒触发一次块设备驱动程序崩溃（Figure 12）。自动恢复会导致性能略有下降。对于读取时，重启和不重启的平均吞吐量分别为 2062 MB/s 和 2164 MB/s (性能下降 5%)。写入方面 ,重启后的总吞吐量平均为 356 MB/s，未重启时为 423 MB/s （性能下降 16%）。

VI. 相关工作

近期有些项目使用 Rust 构建底层高性能系统，包括数据存储[84], [12], [85]、网络功能虚拟化[38]、网络引擎 [86] 以及多个操作系统[87], [10], [13], [88]、unikernel[89] 和 hypervisor[9], [90], [11]。Firecracker [9]、英特尔 Cloud Hypervisor[11] 和谷歌 Chrome OS Virtual Machine Monitor[90] 用基于 Rust 的实现取代了 Qemu 硬件仿真器。Redox[10] 利用 Rust 开发了基于微内核的操作系统（微内核和用户级设备驱动程序都用 Rust 实现，但可自由使用不安全的 Rust）。设备驱动程序在 3 环（ring 3）中运行，使用传统硬件机制进行隔离，并使用系统调用与微内核通信。总的来说，所有这些系统都利用 Rust 作为 C 语言的安全替代品，但并没有探索 Rust 在类型和内存安全之外的功能。

Tock 开发了许多原则，以尽量减少在面向硬件的内核代码中使用不安全的 Rust[13]。Tock 的结构是一个最小的核心内核和一系列设备驱动程序（胶囊）。Tock 依靠 Rust 的语言安全来隔离胶囊（在 Tock 中，用户应用程序与商品硬件机制隔离）。为确保隔离，Tock 禁止在胶囊中进行不安全的扩展，但不限制在胶囊和主内核之间共享指针（这与使用指针作为能力的语言系统类似，如 SPIN[22]）。因此 ，任何一个模块出现故障 整个系统都会停止运行。我们的工作建立在许多设计原则的基础上 ，这些原则旨在最大限度地减少 Tock 开发的不安全 Rust 代码量，并通过支持故障隔离和扩展的动态加载对这些原则进行了扩展。与 Tock 类似，Netbricks[38] 和 Splinter [12] 也依赖 Rust 来隔离网络功能和用户定义的数据库扩展。这些系统都不支持去分配崩溃子系统的资源、恢复或接口和对象引用的通用交换。

VII. 结论

“是一段旅程，而不是一个终点”[27] ,奇点操作系统为影响 Rust 设计的许多概念奠定了基础。反过来，通过在 Rust 本身中应用故障隔离原则，我们的工作完成了这一旅程的循环。然而，红叶只是向前迈出的一步，而不是最终的设计——在实用性和性能原则的指导下，我们的工作首先是一系列机制和一个实验平台，以便未来的系统架构能够利用语言安全。Rust 为系统开发人员提供了我们期待已久的机制：实用、零成本的安全性，以及可强制执行所有权的类型系统。可以说，我们实现的隔离是最关键的机制，因为它为在有故障和不可信组件的系统中确保一系列抽象提供了基础。通过阐明隔离原则，我们的工作将开启未来对隔离和安全抽象的探索：安全动态扩展、细粒度访问控制、最小特权、计算和数据的搭配、透明恢复、和除此之外的很多。

VIII. 致谢

我们要感谢 USENIX ATC 2020 和 OSDI 2020 的评审人员以及我们的指导老师 Michael Swift，他的许多真知灼见帮助我们改进了这项工作。此外，我们还要感谢 Utah CloudLab 团队，特别是 Mike Hibler，感谢他在我们的硬件需求方面给予的持续支持。我们还要感谢 Abhiram Balasubramanian 在红叶设备驱动程序方面提供的帮助，以及 Nivedha Krishnakumar 在底层性能分析方面提供的帮助。本研究部分得到了美国国家科学基金会（拨款号：1837051 和 1840197）、英特尔和 VMWare 的支持。

References

[1] D. Bell, L. L. S. computer system: Unified exposition, and M. 1. Multics interpretation. Technical Report ESD-TR-75-306 MITRE Corp.,.

[2] J. 1. Lester J Fraim. Scomp: A Solution to the Multilevel Security Problem. Computer 16(07):26–34,.

[3] N. N. Y. E. L. Vilanova M. Ben-Yehuda and J. 2. M. Valero. CODOMs: Protecting Software with Codecentric Memory Domains. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA) pages 469–480,.

[4] Emmett Witchel Junghwan Rhee and 2. Krste Asanovi´c. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP ’05) page 31–44,.

[5] D. C. S. W. M. J. A. B. D. B. L. P. G. N. R. N. J. Woodruff R. N. M. Watson and 2. M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA) pages 457–468,.

[6] F. B. A. E. E. G. S.-T. T. G. S. D. L. A. O. M. S. S. V. e. a. T. C. f. W. N. D. i. H.-L. P. L. I. P. o. t. 2. A. S. o. A. f. N. Paul Emmerich Simon Ellmann and 2. Communications Systems (ANCS) pages 1–13. IEEE,.

[7] S. Klabnik and 2. Carol Nichols. The Rust Programming Language. No Starch Press,.

[8] P. W. L. T. C. C. the World! In IFIP TC 2 Working Conference on Programming Concepts and 1. Methods pages 347–359,.

[9] A. I. A. L. R. N. P. P. Alexandru Agache Marc Brooker, D.-M. P. F. L. V. for Serverless Applications. In Proceedings of the 17th USENIX Symposium on Networked Systems Design, and 2. Implementation (NSDI ’20) pages 419–434,.

[10] R. P. D. R. - Your Next(Gen) OS. http://www.redox-os.org/,.

[11] I. C. H. V. https://github.com/cloud-hypervisor/cloud-hypervisor,.

[12] M. N. T. Z. R. R. Chinmay Kulkarni Sara Moore, R. S. S. B.-M. E. for Multi-Tenant Low-Latency Storage. In Proceedings of the 13th USENIX Symposium on Operating Systems Design, and O. 2. Implementation (OSDI ’18) pages 627–643,.

[13] B. G. D. B. G. P. P. P. D. Amit Levy Bradford Campbell, P. L. M. a 64kB Computer Safely, and 2. Efficiently. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17) page 234–251,.

[14] O. developers. Oreboot. https://github.com/oreboot/oreboot,.

[15] X. W. D. Z. N. Z. Haogang Chen Yandong Mao, M. F. K. L. K. V. S.-o.-t.-A. Defenses, and 2. Open Problems. In Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys '11) pages 5:1–5:5,.

[16] Cody Cutler M Frans Kaashoek, R. T. M. T. benefits, costs of writing a POSIX kernel in a high-level language. In Proceedings of the 13th USENIX Symposium on Operating Systems Design, and 2. Implementation (OSDI '18) pages 89–105,.

[17] S. S. C. C. J. L. Nicolas Palix Gaël Thomas, G. M. F. in Linux: Ten Years Later. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages, and 2. Operating Systems (ASPLOS XVI) page 305–318,.

[18] M. P. D. S. by Design: Security and 2. I. D. S. b. D. C. D. W. Legacy at Microsoft. https://vimeo.com/376180843,.

[19] 2. Jeff Vander Stoep. Android: protecting the kernel. Linux Security Summit,.

[20] H. M. L. Michael M Swift Steven Martin and 2. Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In Proceedings of the 10th workshop on ACM SIGOPS European workshop pages 102–107,.

[21] B. N. B. Michael M Swift Muthukaruppan Annamalai and 2. Henry M Levy. Recovering Device Drivers. ACM Transactions on Computer Systems (TOCS) 24(4):333–360,.

[22] P. P. E. G. S. M. E. F. D. B. C. C. B. N. Bershad S. Savage, S. E. E. Safety, and 1. Performance in the SPIN Operating System. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95) page 267–283,.

[23] C. Small, M. I. S. V. A. I. P. for Operating System, D. o. E. Database Research. Technical Report TR 30-94 Harvard University, and 1. Applied Sciences,.

[24] 2. h. Marc Stiegler. The E Language in a Walnut,.

[25] G. Back, W. C. H. T. K. J. R. S. A. T. on Programming Languages, and 2. Systems (TOPLAS) 27(4):583–630,.

[26] C. W. Michael Golm Meik Felser and 2. Jürgen Kleinöder. The JX Operating System. In Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATC '02) page 45–58,.

[27] G. C. Hunt and A. 2. James R. Larus. Singularity: Rethinking the Software Stack. ACM SIGOPS Operating Systems Review 41(2):37–49,.

[28] G. C. C. H. D. H. Thorsten von Eicken Chi-Chao Chang, D. S. J.-K. A. C.-B. O. S. for Java. In Secure Internet Programming: Security Issues for Mobile, and p. 3. 1. Distributed Objects,.

[29] M. S. M. R. C. T. a Unified Approach to Access Control and M. 2. Concurrency Control. PhD thesis Johns Hopkins University,.

[30] Fred Barnes Christian Jacobsen and p. 1. S. 2. Brian Vinter. RMoX: A Raw-Metal occam Experiment. In Communicating Process Architectures 2003 volume 61 of Concurrent Systems Engineering Series,.

[31] E. J. Andrew P. Black Norman C. Hutchinson and 2. Henry M. Levy. The Development of the Emerald Programming Language. In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III) page 11–1–11–51,.

[32] H. Bromley and 2. Richard Lamson. LISP Lore: A Guide to Programming the Lisp Machine. Springer Science & Business Media,.

[33] D. L. P. D. M. R. H. W. T. Sean M Dorward Rob Pike and 1. Philip Winterbottom. The Inferno operating system. Bell Labs Technical Journal 2(1):5–18,.

[34] A. Goldberg, D. R. S.-8. T. Language, and 1. its Implementation. Addison-Wesley Longman Publishing Co. Inc.,.

[35] D. K. Peter W Madany Susan Keohan and M. V. C. 1. Tom Saulpaugh. JavaOS: A Standalone Java Environment. White Paper Sun Microsystems,.

[36] T. R. H. H. C. L. W. C. L. P. R. M. H. G. M. David D Redell Yogen K Dalal and 1. Stephen C Purcell. Pilot: An Operating System for a Personal Computer. Communications of the ACM 23(2):81–92,.

[37] R. J. B. Daniel C Swinehart Polle T Zellweger, R. B. H. A. S. V. of the Cedar Programming Environment. ACM Transactions on Programming Languages, and 1. Systems (TOPLAS) 8(4):419–490,.

[38] K. J. M. W. S. R. Aurojit Panda Sangjin Han, S. S. N. T. the V out of NFV. In Proceedings of the 12th USENIX Symposium on Operating Systems Design, and N. 2. Implementation (OSDI '16) pages 203–216,.

[39] 1. John Lions. Lions' commentary on UNIX 6th edition with source code. Peer-to-Peer Communications Inc.,.

[40] D. Z. X. W. N. Z. Yandong Mao Haogang Chen, M. F. K. S. F. I. with API Integrity, and 2. Multi-Principal Modules. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11) page 115–128,.

[41] C. J. S. S. S. B. M. Q. A. H. A. Y. J. S. M. B. Vikram Narayanan Abhiram Balasubramanian and J. 2. Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19) pages 269–284,.

[42] G. T. T. J. Vikram Narayanan Yongzhe Huang, A. B. L. K. I. with Virtualization, and 2. VM Functions. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20) page 157–171,.

[43] A. K. S. G. C. K. Adam Belay George Prekas, E. B. I. A. P. D. O. S. for High Throughput, L. L. I. P. of the 11th USENIX Symposium on Operating Systems Design, and O. 2. Implementation (OSDI '14) pages 49–65,.

[44] I. C. D. D. P. D. K. http://dpdk.org/,.

[45] I. C. S. P. D. K. (SPDK). https://spdk.io,.

[46] I. Z. D. R. K. P. D. W. A. K. T. A. Simon Peter Jialin Li, T. R. A. T. O. S. is the Control Plane. In Proceedings of the 11th USENIX Symposium on Operating Systems Design, and O. 2. Implementation (OSDI '14) pages 1–16,.

[47] B. W. Lampson and p. 9. 1. Robert F. Sproull. An Open Operating System for a Single-User Machine. In Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP '79),.

[48] 2. Erlang on Xen. http://erlangonxen.org/,.

[49] H. L. V. M. (HaLVM). http://corp.galois.com/halvm,.

[50] C. R. D. S. B. S. T. G. S. S. S. H. Anil Madhavapeddy Richard Mortier and M. 2. Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. ACM SIGARCH Computer Architecture News 41(1):461–472,.

[51] 1. Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Memory Management pages 1–42,.

[52] J. B. A. burying: Unique variables without destructive reads. Software: Practice and 2. Experience 31(6):533–553,.

[53] D. Walker and M. 2. Greg Morrisett. Alias Types for Recursive Data Structures (Extended Version). Technical Report TR2000-1787 Cornell University,.

[54] M. Tofte, J.-P. T. R.-B. M. M. Information, and 1. Computation 132(2):109–176,.

[55] C. H. O. H. G. H. J. R. L. Manuel Fähndrich Mark Aiken, S. L. L. S. for Fast, and 2. Reliable Message-Based Communication in Singularity

OS. In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06) page 177–190,.

[56] M. Fahndrich, R. D. Adoption, F. P. L. T. for Imperative Programming. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design, and 2. Implementation (PLDI '02) pages 13–24,.

[57] D. G. M. W. H. J. C. Trevor Jim J. Greg Morrisett and J. 2. Yanling Wang. Cyclone: A safe dialect of C. In Proceedings of the General Track: 2002 USENIX Annual Technical Conference (ATC '02) pages 275–288,.

[58] B. Ford and 1. Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC '94) pages 97–114,.

[59] F. P. Vytautas Astrauskas Peter Müller, A. J. S. L. R. T. for Modular Specification, and p. 1. Verification. In Proceedings of the ACM on Programming Languages (OOPSLA) volume 3,.

[60] Marek Baranowski Shaobo He, Z. R. V. R. P. with SMACK. In Proceedings of the 16th International Symposium on Automated Technology for Verification, and p. 5. S. 2. Analysis (ATVA) volume 11138 of Lecture Notes in Computer Science,.

[61] J. Toman S. Pernsteiner and N. 2. E. Torlak. Crust: A Bounded Verifier for Rust (N). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) pages 75–80,.

[62] R. K. Ralf Jung Jacques-Henri Jourdan and p. 1. 2. Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In Proceedings of the ACM on Programming Languages (POPL) volume 2,.

[63] I. S. C. M. by Product CPU Model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html,.

[64] O. 1. Kenneth C. Knowlton. A Fast Storage Allocator. Communications of the ACM 8(10):623–624,.

[65] 1. Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In Proceedings of the USENIX Summer 1994 Technical Conference (USTC'94) page 6,.

[66] A. B. A. P. Z. R. Abhiram Balasubramanian Marek S. Baranowski and 2. Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17) pages 156–161,.

[67] a. s. U.-l. t. o. s. h. 2. Robert Morris Russ Cox Frans Kaashoek. Xv6,.

[68] O. K. Andrew Baumann Jonathan Appavoo and 2. Timothy Roscoe. A Fork() in the Road. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19) page 14–22,.

[69] Robert Ricci Eric Eide, C. T. I. C. S. I. for Advancing Cloud Architectures, and 2. Applications. ; login:: the magazine of USENIX & SAGE 39(6):36–38,.

[70] K. Elphinstone and 2. Gernot Heiser. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13) page 133–150,.

[71] Z. Y. X. W. Zeyu Mi Dingji Li, H. C. S. Fast, and 2. Secure Inter-Process Communication for Microkernels. In Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19),.

[72] N. O. D. M. S. P. D. Anjo Vahldiek-Oberwagner Eslam Elnikety and p. 1. A. 2. Deepak Garg. ERIM: Secure Efficient In-process Isolation with Protection Keys (MPK). In Proceedings of the 28th USENIX Security Symposium (USENIX Security '19),.

[73] E. J. J. C. M. L. S. K. S. Mohammad Hedayati Spyridoula Gravani and J. 2. Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19) pages 489–504,.

[74] K. C. H. C. Yutao Liu Tianyu Zhou, Y. X. T. M. D. with Efficient Hypervisor-Enforced Intra-Domain Isolation. In Proceedings of the 22nd ACM SIGSAC Conference on Computer, and 2. Communications Security (CCS '15) page 1607–1619,.

[75] I. 64 and 2. h.-6.-a.-i.-3.-a.-s.-c.-v.-1.-2.-2.-2.-3.-3.-3.-3.-a.-4. IA-32 Architectures Software Developer's Manual,.

[76] C. C. C. S. R. K. E. M.-H. A. C. B. C. W. S. Daniel E. Eisenbud Cheng Yi, J. D. H. M. A. Fast, R. S. N. L. B. I. P. of the 13th USENIX Symposium on Networked Systems Design, and M. 2. Implementation (NSDI '16) pages 523–535,.

[77] S. G. M. K. H. L. H. C. L. R. M. M. P. D. P. P. S. D. S. T. T. Rajesh Nishtala Hans Fugal, V. V. S. M. at Facebook. In Proceedings of the 10th USENIX Symposium on Networked Systems Design, and A. 2. Implementation (NSDI '13) pages 385–398,.

[78] M. J. G. K. A. L. A. P. S. S. P. V. Giuseppe DeCandia Deniz Hastorun and 2. Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07) page 205–220,.

[79] D. G. A. Hyeontaek Lim Dongsu Han, M. K. M. A. H. A. to Fast In-Memory Key-Value Storage. In Proceedings of the 11th USENIX Symposium on Networked Systems Design, and A. 2. Implementation (NSDI '14) pages 429–444,.

[80] G. W. C. B. U. site speed in web search ranking. https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html,.

[81] A. Osmani, I. G. S. is now a landing page factor for Google Search, and A. https://developers.google.com/web/updates/2018/07/search-ads-speed,.

[82] Nginx. Nginx: High Performance Load Balancer Web Server and R. P. https://www.nginx.com/,.

[83] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk,.

[84] J. B. L. T. A. M. E. E. K. M. F. K. Jon Gjengset Malte Schwarzkopf, P.-S. D.-F. f. H.-P. W. A. I. P. o. t. 1. U. C. o. O. S. D. Robert Morris. Noria: Dynamic, and 2. Implementation (OSDI '18) page 213–231,.

[85] M. I. R. I. P. B. Derek G. Murray Frank McSherry and 5. S. 2. Martin Abadi. Incremental Iterative Data Processing with Timely Dataflow. Communications of the ACM,.

[86] t. P. B. E. P. h. Servo,.

[87] K. Boos, L. Z. T. A. S. S.-F. O. S. I. P. of the 9th Workshop on Programming Languages, and 2. Operating Systems (PLOS'17) page 29–35,.

[88] 2. Alex Light. Reenix: Implementing a Unix-like operating system in Rust. Undergraduate Honors Theses Brown University,.

[89] Stefan Lankes Jens Breitbart, S. P. E. R. for Unikernel Development. In Proceedings of the 10th Workshop on Programming Languages, and 2. Operating Systems (PLOS '19) page 8–15,.

[90] G. G. C. O. V. M. M. https://chromium.googlesource.com/chromiumos/platform/crosvm,.