
# RedLeaf: Isolation and Communication in a Safe Operating System

Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, and Zhaofeng Li, *University of California, Irvine;* Gerd Zellweger, *VMware Research;* Anton Burtsev, *University of California, Irvine*

https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

# RedLeaf: Isolation and Communication in a Safe Operating System

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

RedLeaf is a new operating system developed from scratch in Rust to explore the impact of language safety on operating system organization. In contrast to commodity systems, RedLeaf does not rely on hardware address spaces for isolation and instead uses only type and memory safety of the Rust language. Departure from costly hardware isolation mechanisms allows us to explore the design space of systems that embrace lightweight fine-grained isolation. We develop a new abstraction of a lightweight language-based isolation domain that provides a unit of information hiding and fault isolation. Domains can be dynamically loaded and cleanly terminated, i.e., errors in one domain do not affect the execution of other domains. Building on RedLeaf isolation mechanisms, we demonstrate the possibility to implement end-to-end zero-copy, fault isolation, and transparent recovery of device drivers. To evaluate the practicality of RedLeaf abstractions, we implement Rv6, a POSIX-subset operating system as a collection of RedLeaf domains. Finally, to demonstrate that Rust and fine-grained isolation are practical—we develop efficient versions of a 10Gbps Intel ixgbe network and NVMe solid-state disk device drivers that match the performance of the fastest DPDK and SPDK equivalents.

## 1 Introduction

Four decades ago, early operating system designs identified the ability to isolate kernel subsystems as a critical mechanism for increasing the reliability and security of the entire system [12, 32]. Unfortunately, despite many attempts to introduce fine-grained isolation to the kernel, modern systems remain monolithic. Historically, software and hardware mechanisms remain prohibitively expensive for isolation of subsystems with tightest performance budgets. Multiple hardware projects explored the ability to implement fine-grained, low-overhead isolation mechanisms in hardware [84, 89, 90]. However, focusing on performance, modern commodity CPUs provide only basic support for coarse-grained isolation of user applications. Similarly, for decades, overheads of safe languages that can provide fine-grained isolation in software remained prohibitive for low-level operating system code. Traditionally, safe languages require a managed runtime, and specifically, garbage collection, to implement safety. Despite many advances in garbage collection, its overhead is high for systems designed to process millions of requests per second per core (the fastest garbage collected languages experience 20-50% slowdown compared to C on a typical device driver workload [28]).

For decades, breaking the design choice of a monolithic kernel remained impractical. As a result, modern kernels suffer from lack of isolation and its benefits: clean modularity, information hiding, fault isolation, transparent subsystem recovery, and fine-grained access control.

The historical balance of isolation and performance is changing with the development of Rust, arguably, the first practical language that achieves safety without garbage collection [45]. Rust combines an old idea of *linear types* [86] with pragmatic language design. Rust enforces type and memory safety through a restricted ownership model allowing only one unique reference to each live object in memory. This allows statically tracking the lifetime of the object and deallocating it without a garbage collector. The runtime overhead of the language is limited to bounds checking, which in many cases can be concealed by modern superscalar out-of-order CPUs that can predict and execute the correct path around the check [28]. To enable practical non-linear data structures, Rust provides a small set of carefully chosen primitives that allow escaping strict limitations of the linear type system.

Rust is quickly gaining popularity as a tool for development of low-level systems that traditionally were done in C [4, 24, 40, 47, 50, 65]. Low-overhead safety brings a range of immediate security benefits—it is expected, that two-thirds of vulnerabilities caused by low-level programming idioms typical for unsafe languages can be eliminated through the use of a safe language alone [20, 22, 67, 69, 77].

Unfortunately, recent projects mostly use Rust as a drop-in replacement for C. We, however, argue that true benefits of language safety lie in the possibility to enable practical, lightweight, fine-grained isolation and a range of mechanisms

that remained in the focus of systems research but remained impractical for decades: fault isolation [79], transparent device driver recovery [78], safe kernel extensions [13, 75], fine-grained capability-based access control [76], and more.

RedLeaf[1] is a new operating system aimed at exploring the impact of language safety on operating system organization, and specifically the ability to utilize fine-grained isolation and its benefits in the kernel. RedLeaf is implemented from scratch in Rust. It does not rely on hardware mechanisms for isolation and instead uses only type and memory safety of the Rust language.

Despite multiple projects exploring isolation in language-based systems [6, 35, 39, 85] articulating principles of isolation and providing a practical implementation in Rust remains challenging. In general, safe languages provide mechanisms to control access to the fields of individual objects (e.g., through *pub* access modifier in Rust) and protect pointers, i.e., restrict access to the state of the program transitively reachable through visible global variables and explicitly passed arguments. Control over references and communication channels allows isolating the state of the program on function and module boundaries enforcing confidentiality and integrity, and, more generally, constructing a broad range of least-privilege systems through a collection of techniques explored by object-capability languages [59].

Unfortunately, built-in language mechanisms alone are not sufficient for implementing a system that isolates mutually distrusting computations, e.g., an operating system kernel that relies on language safety for isolating applications and kernel subsystems. To protect the execution of the entire system, the kernel needs a mechanism that *isolates faults*, i.e., provides a way to terminate a faulting or misbehaving computation in such a way that it leaves the system in a clean state. Specifically, after the subsystem is terminated the isolation mechanisms should provide a way to 1) deallocate all resources that were in use by the subsystem, 2) preserve the objects that were allocated by the subsystem but then were passed to other subsystems through communication channels, and 3) ensure that all future invocations of the interfaces exposed by the terminated subsystem do not violate safety or block the caller, but instead return an error. Fault isolation is challenging in the face of semantically-rich interfaces encouraged by language-based systems—frequent exchange of references all too often implies that a crash of a single component leaves the entire system in a corrupted state [85].

Over the years the goal to isolate computations in language-based systems came a long way from early single-user, single-language, single-address space designs [9, 14, 19, 25, 34, 55, 71, 80] to ideas of heap isolation [6, 35] and use of linear types to enforce it [39]. Nevertheless, today the principles of language-based isolation are not well understood. Singularity [39], which implemented fault isolation in Sing#, relied on a tight co-design of the language and operating system to implement its isolation mechanisms. Nevertheless, several recent systems suggesting the idea of using Rust for lightweight isolation, e.g., Netbricks [68] and Splinter [47], struggled to articulate the principles of implementing isolation, instead falling back to substituting fault isolation for information hiding already provided by Rust. Similar, Tock, a recent operating system in Rust, supports fault isolation of user processes through traditional hardware mechanisms and a restricted system call interface, but fails to provide fault isolation of its device drivers (capsules) implemented in safe Rust [50].

Our work develops principles and mechanisms of *fault isolation* in a safe language. We introduce an abstraction of a language-based isolation domain that serves as a unit of information hiding, loading, and fault isolation. To encapsulate domain's state and implement fault isolation at domain boundary, we develop the following principles:

- **Heap isolation** We enforce *heap isolation* as an invariant across domains, i.e., domains never hold pointers into private heaps of other domains. Heap isolation is key for termination and unloading of crashing domains, since no other domains hold pointers into the private heap of a crashing domain, it's safe to deallocate the entire heap. To enable cross-domain communication, we introduce a special *shared heap* that allows allocation of objects that can be exchanged between domains.
- **Exchangeable types** To enforce heap isolation, we introduce the idea of *exchangeable types*, i.e., types that can be safely exchanged across domains without leaking pointers to private heaps. Exchangeable types allow us to statically enforce the invariant that objects allocated on the shared heap cannot have pointers into private domain heaps, but can have references to other objects on the shared heap.
- **Ownership tracking** To deallocate resources owned by a crashing domain on the shared heap, we track ownership of all objects on the shared heap. When an object is passed between domains we update its ownership depending on whether it's moved between domains or borrowed in a read-only access. We rely on Rust's *ownership discipline* to enforce that domains lose ownership when they pass a reference to a shared object in a cross-domain function call, i.e., Rust enforces that there are no aliases into the passed object left in the caller domain.
- **Interface validation** To provide extensibility of the system and allow domain authors to define custom interfaces for subsystems they implement while retaining isolation, we validate all cross-domain interfaces enforcing the invariant that interfaces are restricted to exchangeable types and hence preventing them from breaking the heap isolation invariants. We develop an interface definition language (IDL) that statically validates definitions of cross-domain interfaces and generates implementations for them.

[1] Forming in the leaf tissue Rust fungi turn it red.

- **Cross-domain call proxying** We mediate all cross-domain invocations with *invocation proxies*—a layer of trusted code that interposes on all domain's interfaces. Proxies update ownership of objects passed across domains, provide support for unwinding execution of threads from a crashed domain, and protect future invocations of the domain after it is terminated. Our IDL generates implementations of the proxy objects from interface definitions.

The above principles allow us to enable fault-isolation in a practical manner: isolation boundaries introduce minimal overhead even in the face of semantically-rich interfaces. When a domain crashes, we isolate the fault by unwinding execution of all threads that currently execute inside the domain, and deallocate domain's resources without affecting the rest of the system. Subsequent invocations of domain's interfaces return errors, but remain safe and do not trigger panics. All objects allocated by the domain, but returned before the crash, remain alive.

To test these principles we implement RedLeaf as a microkernel system in which a collection of isolated domains implement functionality of the kernel: typical kernel subsystems, POSIX-like interface, device drivers, and user applications. RedLeaf provides typical features of a modern kernel: multi-core support, memory management, dynamic loading of kernel extensions, POSIX-like user processes, and fast device drivers. Building on RedLeaf isolation mechanisms, we demonstrate the possibility to transparently recover crashing device drivers. We implement an idea similar to *shadow drivers* [78], i.e., lightweight shadow domains that mediate access to the device driver and restart it replaying its initialization protocol after the crash.

To evaluate the generality of RedLeaf abstractions, we implement Rv6, a POSIX-subset operating system on top of RedLeaf. Rv6 follows the UNIX V6 specification [53]. Despite being a relatively simple kernel, Rv6 is a good platform that illustrates how ideas of fine-grained, language-based isolation can be applied to modern kernels centered around the POSIX interface. Finally, to demonstrate that Rust and fine-grained isolation introduces a non-prohibitive overhead, we develop efficient versions of 10Gbps Intel Ixgbe network and PCIe-attached solid state-disk NVMe drivers.

We argue that a combination of practical language safety and ownership discipline allows us to enable many classical ideas of operating system research for the first time in an efficient way. RedLeaf is fast, supports fine-grained isolation of kernel subsystems [57, 61, 62, 79], fault isolation [78, 79], implements end-to-end zero-copy communication [39], enables user-level device drivers and kernel bypass [11, 21, 42, 70], and more.

## 2 Isolation in Language-Based Systems

Isolation has a long history of research in language-based systems that were exploring tradeoffs of enforcing lightweight isolation boundaries through language safety, fine-grained control of pointers, and type systems. Early operating systems applied safe languages for operating system development [9, 14, 19, 25, 34, 55, 71, 80]. These systems implemented an "open" architecture, i.e., a single-user, single-language, single-address space operating system that blurred the boundary between the operating system and the application itself [48]. These systems relied on language safety to protect against accidental errors but did not provide isolation of subsystems or user-applications (modern unikernels take a similar approach [2, 37, 56]).

SPIN was the first to suggest language safety as a mechanism to implement isolation of dynamic kernel extensions [13]. SPIN utilized Modula-3 pointers as capabilities to enforce confidentiality and integrity, but since pointers were exchanged across isolation boundaries it failed to provide fault isolation—a crashing extension left the system in an inconsistent state.

J-Kernel [85] and KaffeOS [6] were the first kernels to point out the problem that language safety alone is not sufficient for enforcing fault isolation and termination of untrusted subsystems. To support termination of isolated domains in Java, J-Kernel developed the idea of mediating accesses to all objects that are shared across domains [85]. J-Kernel introduces a special capability object that wraps the interface of the original object shared across isolated subsystems. To support domain termination, all capabilities created by a crashing domain were revoked hence dropping the reference to the original object that was garbage collected and preventing the future accesses by returning an exception. J-Kernel relied on a custom class loader to validate cross-domain interfaces (i.e., generate remote-invocation proxies at run-time instead of using a static IDL compiler). To enforce isolation, J-Kernel utilized a special calling convention that allowed passing capability references by reference, but required a deep copy for regular unwrapped objects. Without ownership discipline for shared objects, J-Kernel provided a somewhat limited fault isolation model: the moment the domain that created the object crashed all references to the shared objects were revoked, propagating faults into domains that acquired these objects through cross-domain invocations. Moreover, lack of "move" semantics, i.e., the ability to enforce that the caller lost access to the object when it was passed to the callee, implied that isolation required a deep copy of objects which is prohibitive for isolation of modern, high-throughput device drivers.

Instead of mediating accesses to shared objects through capability references, KaffeOS adopts the technique of "write barriers" [88] that validate all pointer assignments throughout the system and hence can enforce a specific pointer discipline [6]. KaffeOS introduced separation of private domain and special shared heaps designated for sharing of objects across domains—explicit separation was critical to perform the write barrier check, i.e., if assigned pointer belonged to a specific heap. Write barriers were used to enforce the follow-

ing invariants: 1) objects on the private heap were allowed to have pointers into objects on the shared heap, but 2) objects on the shared heap were constrained to the same shared heap. On cross-domain invocations, when a reference to a shared object was passed to another domain, the write barrier was used to validate the invariants, and also to create a special pair of objects responsible for reference counting and garbage collecting shared objects. KaffeOS had the following fault isolation model: when the creator of the object terminated, other domains retained access to the object (reference counting ensured that eventually objects were deallocated when all sharers terminated). Unfortunately, while other domains were able to access the objects after their creator crashed, it was not sufficient for clean isolation—shared objects were potentially left in an inconsistent state (e.g., if the crash happened halfway through an object update), thus potentially halting or crashing other domains. Similar to J-Kernel, isolation of objects required a deep copy on a cross-domain invocation. Finally, performance overhead of mediating all pointer updates was high.

Singularity OS introduced a new fault isolation model built around a statically enforced ownership discipline [39]. Similar to KaffeOS, in Singularity applications used isolated private heaps and a special "exchange heap" for shared objects. A pioneering design decision was to enforce *single ownership* of objects allocated on the exchange heap, i.e., only one domain could have a reference to an object on the shared heap at a time. When a reference to an object was passed across domains the ownership of the object was "moved" between domains (an attempt to access the object after passing it to another domain was rejected by the compiler). Singularity developed a collection of novel static analysis and verification techniques enforcing this property statically in a garbage collected Sing# language. Single ownership was key for a clean and practical fault isolation model—crashing domains were not able to affect the rest of the system—not only their private heaps were isolated, but a novel ownership discipline allowed for isolation of the shared heap, i.e., there was no way for a crashing domain to trigger revocation of shared references in other domains, or leave shared objects in an inconsistent state. Moreover, single ownership allowed secure isolation in a zero-copy manner, i.e., the move semantics guaranteed that the sender of an object was losing access to it and hence allowed the receiver to update the object's state knowing that the sender was not able to access new state or alter the old state underneath.

Building on the insights from J-Kernel, KaffeOS, and Singularity, our work develops principles for enforcing fault isolation in a safe language that enforces ownership. Similar to J-Kernel, we adopt wrapping of interfaces with proxies. We, however, generate proxies statically to avoid the run-time overhead. We rely on heap isolation similar to KaffeOS and Singularity. Our main reason for heap isolation is to be able to deallocate the domain's private heap without any semantic knowledge of objects inside. We borrow move semantics for the objects on the shared heap to provide clean fault isolation and at the same time support zero-copy communication from Singularity. We, however, extend it with the read-only borrow semantics which we need to support transparent domain recovery without giving up zero-copy. Since we implement RedLeaf in Rust, we benefit from its ownership discipline that allows us to enforce the move semantics for objects on the shared heap. Building on a body of research on linear types [86], affine types, alias types [18, 87], and region-based memory management [81], and being influenced by languages like Sing# [29], Vault [30], and Cyclone [43], Rust enforces ownership statically and without compromising usability of the language. In contrast to Singularity that heavily relies on the co-design of Sing# [29] and its communication mechanisms, we develop RedLeaf's isolation abstractions—exchangeable types, interface validation, and cross-domain call proxying—outside of the Rust language. This allows us to clearly articulate the minimal set of principles required to provide fault isolation, and develop a set of mechanisms implementing them independently from the language, that, arguably, allows adapting them to specific design tradeoffs. Finally, we make several design choices aimed at practicality of our system. We design and implement our isolation mechanisms for the most common, "migrating threads" model [31] instead of messages [39] to avoid a thread context switch on the critical cross-domain call path and allow a more natural programming idiom, e.g., in RedLeaf domain interfaces are just Rust traits.

## 3 RedLeaf Architecture

RedLeaf is structured as a microkernel system that relies on lightweight language-based domains for isolation (Figure 1). The microkernel implements functionality required to start threads of execution, memory management, domain loading, scheduling, and interrupt forwarding. A collection of isolated domains implement device drivers, personality of an operating system, i.e., the POSIX interface, and user applications (Section 4.5). As RedLeaf does not rely on hardware isolation primitives, all domains and the microkernel run in ring 0. Domains, however, are restricted to safe Rust (i.e., microkernel and trusted libraries are the only parts of RedLeaf that are allowed to use unsafe Rust extensions).

We enforce the heap isolation invariant between domains. To communicate, domains allocate shareable objects from a global *shared heap* and exchange special pointers, remote references (`RRef<T>`), to objects allocated on the shared heap (Section 3.1). The ownership discipline allows us to implement lightweight zero-copy communication across isolated domains (Section 3.1).

Domains communicate via normal, typed Rust function invocations. Upon cross-domain invocation, the thread moves between domains but continues execution on the same stack. Domain developers provide an interface definition for the

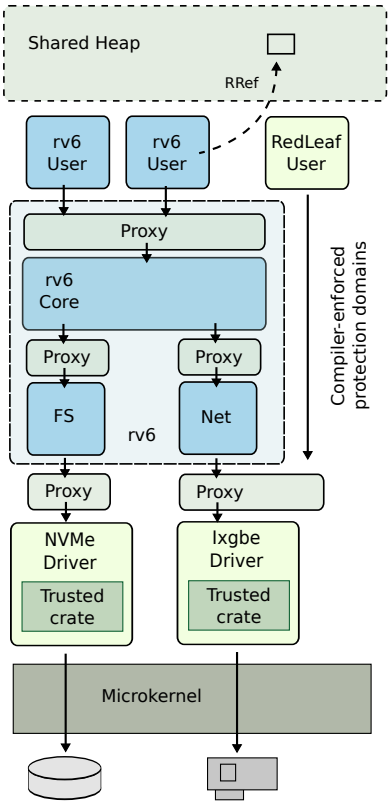


**Figure 1:** RedLeaf architecture

domain's entry point and its interfaces. The RedLeaf IDL compiler automatically generates code for creating and initializing domains and checks the validity of all types passed across domain boundaries (Section 3.1.5).

RedLeaf mediates all cross-domain communication with trusted proxy objects. Proxies are automatically generated from the IDL definitions by the IDL compiler (Section 3.1.5). On every domain entry, the proxy checks if a domain is alive and if so, it creates a lightweight continuation that allows us to unwind execution of the thread if the domain crashes.

In RedLeaf references to objects and traits are capabilities. In Rust, a trait declares a set of methods that a type must implement hence providing an abstraction of an interface. To expose their functionality, domains exchange references to traits via cross-domain calls. We rely on capability-based access control [76] to enforce the principle of least privilege and enable flexible operating system organizations: e.g., we implement several scenarios in which applications talk to the device driver directly bypassing the kernel, and even can link against device driver libraries leveraging DPDK-style user-level device driver access.

**Protection model** The core assumptions behind RedLeaf are that we trust (1) the Rust compiler to implement language safety correctly, and (2) Rust core libraries that use unsafe code, e.g., types that implement interior mutability, etc. RedLeaf's TCB includes the microkernel, a small set of trusted RedLeaf crates required to implement hardware interfaces and low-level abstractions, device crates that provide a safe interface to hardware resources, e.g., access to DMA buffers, etc., the RedLeaf IDL compiler, and the RedLeaf trusted compilation environment. At the moment, we do not address vulnerabilities in unsafe Rust extensions, but again speculate that eventually all unsafe code will be verified for functional correctness [5, 8, 82]. Specifically, the RustBelt project provides a guide for ensuring that unsafe code is encapsulated within a safe interface [44].

We trust devices to be non-malicious. This requirement can be relaxed in the future by using IOMMUs to protect physical memory. Finally, we do not protect against side-channel attacks; while these are important, addressing them is simply beyond the scope of the current work. We speculate that hardware counter-measures to alleviate the information leakage will find their way in the future CPUs [41].

## 3.1 Domains and Fault Isolation

In RedLeaf domains are units of information hiding, fault isolation, and composition. Device drivers, kernel subsystems, e.g., file system, network stack, etc., and user programs are loaded as domains. Each domain starts with a reference to a microkernel system-call interface as one of its arguments. This interface allows every domain to create threads of execution, allocate memory, create synchronization objects, etc. By default, the microkernel system call interface is the only authority of the domain, i.e., the only interface through which the domain can affect the rest of the system. Domains however can define a custom type for an entry function requesting additional references to objects and interfaces to be passed when it is created. By default, we do not create a new thread of execution for the domain.

Every domain, however, can create threads from the `init` function called by the microkernel when the domain is loaded. Internally, the microkernel keeps track of all resources created on behalf of each domain: allocated memory, registered interrupt threads, etc. Threads can outlive the domain creating them as they enter other domains where they can run indefinitely. Those threads continue running until they return to the crashed domain and it is the last domain in their continuation chain.

**Fault isolation** RedLeaf domains provide support for *fault-isolation*. We define fault isolation in the following manner. We say that a domain *crashes* and needs to be terminated when one of the threads that enters the domain panics. Panic potentially leaves objects reachable from inside the domain in an inconsistent state, making further progress of any of the threads inside the domain impractical (i.e., even if threads do not deadlock or panic, the results of the computation are undefined). Then, we say that the *fault is isolated* if the following conditions hold. First, we can unwind all threads running inside the crashing domain to the domain entry point and return an error to the caller. Second, subsequent attempts to

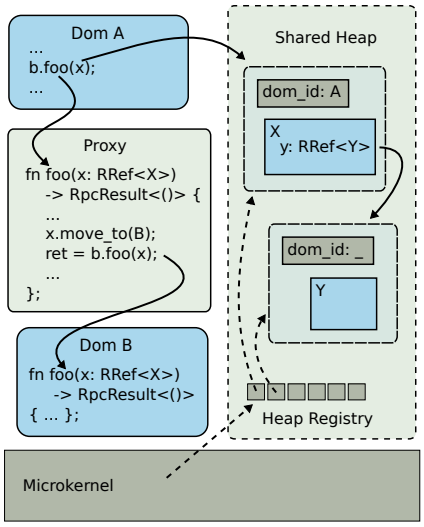


**Figure 2:** Inter-domain communication. Domain `A` invokes method `foo()` of domain `B`. The proxy that interposes on the invocation moves the ownership of the object pointed by `x` between domains.

invoke the domain return errors but do not violate safety guarantees or result in panics. Third, all resources of the crashed domain can be safely deallocated, i.e., other domains do not hold references into the heap of the crashed domain (heap isolation invariant), and we can reclaim all resources owned by the domain without leaks. Fourth, threads in other domains continue execution, and can continue accessing objects that were allocated by the crashed domain, but were moved to other domains before the crash.

Enforcing fault isolation is challenging. In RedLeaf isolated subsystems export complex, semantically rich interfaces, i.e., domains are free to exchange references to interfaces and hierarchies of objects. We make several design choices that allow us to cleanly encapsulate domain's state and yet support semantically rich interfaces and zero-copy communication.

### 3.1.1 Heap Isolation and Sharing

**Private and shared heaps** To provide fault isolation across domains and ensure safe termination of domains, we enforce *heap isolation* across domains, i.e., objects allocated on the private heap, stack, or global data section of the domain can not be reached from outside of the domain. This invariant allows us to safely terminate any domain at any moment of execution. Since no other domain holds pointers into the private heap of a terminated domain, it is safe to deallocate the entire heap.

To support efficient cross-domain communication, we provide a special, global *shared heap* for objects that can be sent across domains. Domains allocate objects on the shared heap in a way similar to the normal heap allocation with the Rust `Box<T>` type that allocates a value of type `T` on the heap. We construct a special type, *remote reference* or `RRef<T>`, that allocates a value of type `T` on the shared heap (Figure 2). `RRef<T>` consists of two parts: a small metadata and the value itself. The `RRef<T>` metadata contains an identifier of the domain currently owning the reference, borrow counter, and type information for the value. The `RRef<T>` metadata along with the value are allocated on the shared heap that allows `RRef<T>` to outlive the domain that originally allocates it.

**Memory allocation on the domain heap** To provide encapsulation of domain's private heap, we implement a two-level memory allocation scheme. At the bottom, the microkernel provides domains with an interface for allocating untyped coarse-grained memory regions (larger than one page). Each coarse-grained allocation is recorded in the *heap registry*. To serve fine-grained typed allocations on the domain's private heap, each domain links against a trusted crate that provides the Rust memory allocation interface, `Box<T>`. Domain heap allocations follow the rules of the Rust's ownership discipline, i.e., objects are deallocated when they go out of scope. The two-level scheme has the following benefit: allocating only large memory regions, the microkernel records all memory allocated by the domain without significant performance overheads. If the domain panics, the microkernel walks the registry of all untyped memory regions allocated by the allocator assigned to the domain and deallocates them without calling any destructors. Such untyped, coarse-grained deallocation is safe as we ensure the heap isolation invariant: other domains have no references into the deallocated heap.

### 3.1.2 Exchangeable Types

Objects allocated on the shared heap are subject to the following rule: they can be composed of only of *exchangeable* types. Exchangeable types enforce the invariant that objects on the shared heap cannot have pointers into private or shared heaps, but can have `RRef<T>`s to other objects allocated on the shared heap. RedLeaf's IDL compiler validates this invariant when generating interfaces of the domain (Section 3.1.5). We define exchangeable types as the following set: 1) `RRef<T>` itself, 2) a subset of Rust primitive `Copy` types, e.g., `u32`, `u64`, but not references in the general case, nor pointers, 3) anonymous (tuples, arrays) and named (enums, structs) composite types constructed out of exchangeable types, 4) references to traits with methods that receive exchangeable types. Also, all trait methods are required to follow the following calling convention that requires them to return the `RpcResult<T>` type to support clean abort semantics for threads returning from crashing domains (Section 3.1). The IDL checks interface definition and validates that all types are well-formed (Section 3.1.5).

### 3.1.3 Ownership Tracking

In RedLeaf `RRef<T>`s can be freely passed between domains. We allow `RRef<T>`s to be moved or borrowed immutably. However, we implement an ownership discipline for `RRef<T>`s that is enforced on cross-domain invocations. Ownership tracking allows us to safely deallocate objects on the shared heap owned by a crashing domain. The metadata section of the

RRef<T> keeps track of the owner domain and the number of times it was borrowed immutably on cross-domain invocations.

Initially, RRef<T> is owned by the domain that allocates the reference. If the reference is *moved* to another domain in a cross-domain call, we change the owner identifier inside RRef<T> moving ownership from one domain to another. All cross-domain communication is mediated by trusted proxies, so we can securely update the owner identifier from the proxy. Rust's ownership discipline ensures that there is always only one remote reference to the object inside the domain, hence when the reference is moved between domains on a cross-domain call, the caller loses access to the object passing it to the callee. If the reference is *borrowed immutably* in a cross-domain call, we do not change the owner identifier inside RRef<T>, but instead increment the counter that tracks the number of times RRef<T> was borrowed.

**Recursive references** RRef<T>s can form hierarchies of objects. To avoid moving all RRef<T>s in the hierarchy recursively on a cross-domain invocation, only the root of the object hierarchy has a valid owner identifier (in Figure 2 only object X has a valid domain identifier A, object Y does not). Upon a cross-domain call, the root RRef<T> is updated by the proxy which changes the domain identifier to move ownership of the RRef<T> between domains. This requires a special scheme for deallocating RRef<T>s in case of a crash: we scan the entire RRef<T> registry to clean up resources owned by a crashing domain. To prevent deallocation of children objects of the hierarchy, we rely on the fact that they do not have a valid RRef<T> identifier (we skip them during the scan). The drop method of the root RRef<T> object walks the entire hierarchy and deallocates all children objects (RRef<T>s cannot form cycles). Note, we should carefully handle the case when an RRef<T> is taken out of the hierarchy. To deallocate this RRef<T> correctly we need to assign it a valid domain identifier, i.e., Y gets a proper domain identifier when it is moved out from X. We mediate RRef<T> field assignments with trusted accessor methods. We generate accessor methods that provide the only way to take out an RRef<T> from an object field. This allows us to mediate the move operation and update the domain identifier for the moved RRef<T>. Note that accessors cannot be enforced for the unnamed composite types, e.g., arrays and tuples. For these types we update ownership of all composite elements upon crossing the domain boundary.

**Reclaiming shared heap** Ownership tracking allows us to deallocate objects that are currently owned by the crashing domain. We maintain a global registry of all allocated RRef<T>s (Figure 2). When a domain panics, we walk through the registry and deallocate all references that are owned by the crashing domain. We defer deallocation if RRef<T> was borrowed until the borrow count drops to zero. Deallocation of each RRef<T> requires that we have a drop method for each RRef<T> type and can identify the type of the reference dynamically. Each RRef<T> has a unique type identifier generated by the IDL compiler (the IDL knows all RRef<T> types in the system as it generates all cross-domain interfaces). We store the type identifier along with the RRef<T> and invoke the appropriate drop method to correctly deallocate any, possibly, hierarchical data structure on the shared heap.

### 3.1.4 Cross-Domain Call Proxying

To enforce fault isolation, RedLeaf relies on *invocation proxies* to interpose on all cross-domain invocations (Figure 2). A proxy object exposes an interface identical to the interface it mediates. Hence the proxy interposition is transparent to the user of the interface. To ensure isolation and safety, the proxy implements the following inside each wrapped function: 1) The proxy checks if the domain is alive before performing the invocation. If the domain is alive, the proxy records the fact that the thread moves between domains by updating its state in the microkernel. We use this information to unwind all threads that happen to execute inside the domain when it crashes. 2) For each invocation, the proxy creates a lightweight continuation that captures the state of the thread right before the cross-domain invocation. The continuation allows us to unwind execution of the thread, and return an error to the caller. 3) The proxy moves ownership of all RRef<T>s passed as arguments between domains, or updates the borrow count for all references borrowed immutably. 4) Finally, the proxy wraps all trait references passed as arguments: the proxy creates a new proxy for each trait and passes the reference to the trait implemented by that proxy.

**Thread unwinding** To unwind execution of a thread from a crashing domain, we capture the state of the thread right before it enters the callee domain. For each function of the trait mediated by the proxy, we utilize an assembly trampoline that saves all general registers into a *continuation*. The microkernel maintains a stack of continuations for each thread. Each continuation contains the state of all general registers and a pointer to an error handling function that has the signature identical to the function exported by the domain's interface. If we have to unwind the thread, we restore the stack to the state captured by the continuation, and invoke the error handling function on the same stack and with the same values of general registers. The error handling function returns an error to the caller.

To cleanly return an error in case of a crash, we enforce the following calling convention for all cross-domain invocations: every cross-domain function must return RpcResult<T>, an enumerated type that either holds the returned value or an error (Figure 3). This allows us to implement the following invariant: functions unwound from the crashed domain never return corrupted data, but instead return an RpcResult<T> error.

### 3.1.5 Interface Validation

RedLeaf's IDL compiler is responsible for validation of domain interfaces and generation of proxy code required for enforcing the ownership discipline on the shared heap. RedLeaf

```
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:** `BDev` domain IDL interface definitions.

IDL is a subset of Rust extended with several attributes to control generation of the code (Figure 3). This design choice allows us to provide developers with the familiar Rust syntax and also re-use Rust's parsing infrastructure.

To implement an abstraction of an interface, we rely on Rust's *traits*. Traits provide a way to define a collection of methods that a type has to implement to satisfy the trait, hence defining a specific behavior. For example, the `BDev` trait requires any type that provides it to implement two methods: `read()` and `write()` (Figure 3). By exchanging references to trait objects domains connect to the rest of the system and establish communication with other domains.

Each domain provides an IDL definition for the *create* trait that allows any domain that has access to this trait to create domains of this type (Figure 3). Marked with the `#[create]` attribute, the create trait both defines the type of the domain entry function, and the trait that can be used to create the domain. Specifically, the entry function of the `BDev` domain takes the `PCI` trait as an argument and returns a pointer to the `BDev` interface. Note that when the `BDev` domain is created along with the `BDev` interface, the microkernel also returns the `Domain` trait that allows creator of the domain to control it later. The IDL generates Rust implementations of both the create trait and the microkernel code used to create the domain of this type.

**Interface validation** We perform interface validation as a static analysis pass of the IDL compiler. The compiler starts by parsing all dependent IDL files creating a unified abstract syntax tree (AST), which is then passed to validation and generation stages. During the interface validation pass, we use the AST to extract relevant information for each type that we validate. Essentially, we create a graph that encodes information about all types and relationships between them. We then use this graph to verify that each type is exchangeable and that all isolation constraints are satisfied: methods of cross-domain interfaces return `RpcResult<T>`, etc.

## 3.2 Zero-copy Communication

A combination of the Rust's ownership discipline and the single-ownership enforced on the shared heap allows us to provide isolation without sacrificing end-to-end zero-copy across the system. To utilize zero-copy communication, domains allocate objects on the shared heap with using the `RRef<T>` type. On every cross-domain invocation a mutable reference (a reference that provides writable access to the object) is moved between domains, or an immutable reference can be borrowed. If the invocation succeeds, i.e., the callee domain does not panic, a set of `RRef<T>`s might be returned by the callee moving the ownership to the caller. In contrast to Rust itself, we do not allow borrowing of mutable references. Borrowing of mutable references may result in an inconsistent state in the face of a domain crash when damaged objects are returned to the caller after the thread is unwound. Hence, we require all mutable references to be moved and returned explicitly. If a domain crashes, instead of a reference an `RpcResult<T>` error is returned.

Zero-copy is challenging in the face of crashing domains and the requirement to provide transparent recovery. A typical recovery protocol re-starts the crashing domain and re-issues the failing domain call, trying to conceal the crash from the caller. This often requires that objects passed as arguments in the re-started invocation are available inside the recovery domain. It is possible to create a copy of each object before each invocation, but this introduces significant overhead. To recover domains without additional copies, we rely on support for immutable borrowing of `RRef<T>`s on cross-domain invocations. For example, the `write()` method of the `BDev` interface borrows an immutable reference to the data written to the block device (Figure 3). If an immutable reference is borrowed by the domain, Rust's type system guarantees that the domain cannot modify the borrowed object. Hence, even if the domain crashes, it is safe to return the unmodified read-only object to the caller. The caller can re-issue the invocation as part of the recovery protocol providing the immutable reference as an argument again. This allows implementing transparent recovery without creating backup copies of arguments on each invocation that can potentially crash.

# 4 Implementation

While introducing a range of novel abstractions, we guide the design of RedLeaf by principles of practicality and performance. To a degree, RedLeaf is designed as a replacement for full-featured, commodity kernels like Linux.

## 4.1 Microkernel

The RedLeaf microkernel provides a minimal interface for creating and loading isolated domains, threads of execution, scheduling, low-level interrupt dispatch, and memory management. RedLeaf implements memory management mechanisms similar to Linux—a combination of buddy [46] and slab [16] allocators provides an interface for heap allocation inside the microkernel (the `Box<T>` mechanism). Each domain runs its own allocator internally and requests regions of memory directly from the kernel buddy allocator.

We implement the low-level interrupt entry and exit code in assembly. While Rust provides support for the *x86-interrupt* function ABI (a way to write a Rust function that takes the

x86 interrupt stack frame as an argument), in practice, it is not useful as we need the ability to interpose on the entry and exit from the interrupt, for example, to save all CPU registers.

In RedLeaf device drivers are implemented in user domains (the microkernel itself does not handle any device interrupts besides timer and NMI). Domains register threads as interrupt handlers for device-generated interrupts. For each external interrupt, the microkernel maintains a list of threads waiting for an interrupt. The threads are put back on the scheduler run queue when the interrupt is received.

## 4.2 Dynamic Domain Loading

In RedLeaf domains are compiled independently from the kernel and are loaded dynamically. Rust itself provides no support for dynamic extensions (except Splinter [47], existing Rust systems statically link all the code they execute [7, 50, 68]). Conceptually, the safety of dynamic extensions relies on the following invariant: types of all data structures that cross a domain boundary, including the type of the entry point function, and all types passed through any interfaces reachable through the entry function are the same, i.e., have identical meaning and implementation, across the entire system. This ensures that even though parts of the system are compiled separately type safety guarantees are preserved across domain boundaries.

To ensure that types have the same meaning across all components of the system, RedLeaf relies on a *trusted compilation environment*. This environment allows the microkernel to check that domains are compiled against the same versions of IDL interface definitions, and with the same compiler version, and flags. When a domain is compiled, the trusted environment signs the fingerprint that captures all IDL files, and a string of compiler flags. The microkernel verifies the integrity of the domain when it is loaded. Additionally, we enforce that domains are restricted to only safe Rust, and link against a white-listed set of Rust libraries.

**Code generation** Domain creation and loading rely on the code generated by the IDL compiler (Figure 4). IDL ensures safety at domain boundaries and allows support for user-defined domain interfaces. From the definitions of domain interfaces (Figure 4, ①) and its create function (②) the IDL generates the following code: 1) Rust implementations of all interfaces (③) and the create (④) trait, 2) a trusted entry point function (⑤) that is placed in the domain's build tree and compiled along with the rest of the domain to ensure that domain's entry function matches the domain create code, hence preserving safety on the domain boundary, 3) a microkernel domain create function that creates domains with a specific type signature of the entry point function (⑥), and 4) implementation of the proxy for this interface (⑦). By controlling the generation of the entry point, we ensure that the types of the entry function inside the microkernel and inside the domain match. If a domain tries to violate safety by changing the type of its entry function the compilation fails.

## 4.3 Safe Device Drivers

In RedLeaf device drivers are implemented as regular domains with no additional privileges. Like other domains they are restricted to the safe subset of Rust. To access the hardware, we provide device drivers with a collection of trusted crates that implement a safe interface to the hardware interface of the device, e.g., access to device registers and its DMA engines. For example, the ixgbe device crate provides access to the BAR region of the device, and abstracts its submit and receive queues with the collection of methods for adding and removing requests from the buffers.

Device driver domains are created by the `init` domain when the system boots. Each PCI device takes a reference to the PCI trait that is implemented inside the `pci` domain. Similar to other driver domains, the PCI driver relies on a trusted crate to enumerate all hardware devices on the bus. The trusted crate constructs `BARAddr` objects that contain addresses of PCI BAR regions. We protect each `BARAddr` object with a custom type, so it can only be used inside the trusted device crate that implements access to this specific BAR region. The `pci` domain probes device drivers with matching device identifiers. The driver receives a reference to the `BARAddr` object and starts accessing the device via its trusted crate.

## 4.4 Device Driver Recovery

Lightweight isolation mechanisms and clean domain interfaces allow us to implement transparent device driver recovery with shadow drivers [78]. We develop shadow drivers as normal unprivileged RedLeaf domains. Similar to proxy objects, the shadow driver wraps the interface of the device driver and exposes an identical interface. In contrast to the proxy which is relatively simple and can be generated from the IDL definition, the shadow driver is intelligent as it implements a driver-specific recovery protocol. The shadow driver interposes on all communication with the driver. During normal operation, the shadow passes all calls to the real device driver. However, it saves all information required for the recovery of the driver (e.g., references to PCI trait, and other parts of the device initialization protocol). If the driver crashes, the shadow driver receives an error from the proxy domain. The proxy itself receives the error when the thread is unwound through the continuation mechanism. Instead of returning an error to its caller, the shadow triggers the domain recovery protocol. It creates a new driver domain and replays its initialization protocol, by interposing on all external communication of the driver.

## 4.5 Rv6 Operating System Personality

To evaluate the generality of RedLeaf's abstractions, we implemented Rv6, a POSIX-subset operating system on top of RedLeaf. At a high-level, Rv6 follows the implementation of the xv6 operating system [73], but is implemented as a collection of isolated RedLeaf domains. Specifically, we implement Rv6 as the following domains: the core kernel, file system,

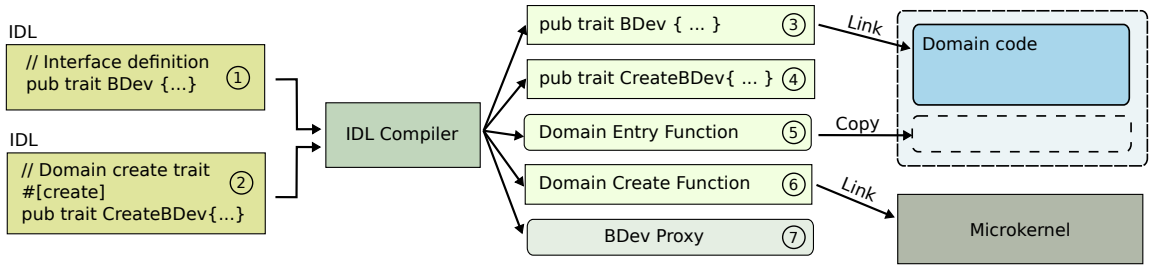


**Figure 4:** IDL code generation

network stack subsystem, network and disk device drivers, and collection of user domains. User domains communicate with the core kernel through the Rv6 system call interface. The core kernel dispatches the system call to either the file system or a network stack. The file system itself communicates with one of the RedLeaf block device drivers to get access to disk. We implemented three block device drivers: in-memory, AHCI, and NVMe. The file system implements journaling, buffer cache, inode, and naming layers. The network subsystem implements the TCP/IP stack and connects to the network device driver (we currently implement only one driver that supports a 10Gbps Intel Ixgbe device). We do not support the full semantics of the `fork()` system call as we do not rely on address spaces and hence cannot virtualize and clone the address space of the domain. Instead, we provide a combination of `create` system calls that allow user applications to load and start new domains [10]. Rv6 boots into a shell that supports pipes and I/O redirection and can start other applications similar to a typical UNIX system.

## 5 Evaluation

We conduct all experiments in the openly-available CloudLab network testbed [72].[2] For network-based experiments, we utilize two CloudLab c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.6 GHz, 160 GB RAM, and a dual-port Intel X520 10Gb NIC. We run our NVMe benchmarks on a CloudLab d430 node that is configured with two 2.4 GHz 64-bit 8-Core E5-2630 Haswell CPUs, and a PCIe-attached 400GB Intel P3700 Series SSD. Linux machines run 64-bit Ubuntu 18.04 with a 4.8.4 kernel configured without any speculative execution attack mitigations as recent Intel CPUs address a range of speculative execution attacks in hardware. All RedLeaf experiments are performed on bare-metal hardware. In all the experiments, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling to reduce the variance in benchmarking.

### 5.1 Overheads of Domain Isolation

**Language based isolation versus hardware mechanisms**

To understand the benefits of language-based isolation over traditional hardware mechanisms, we compare RedLeaf's cross-domain calls with the synchronous IPC mechanism implemented by the seL4 microkernel [27], and a recent kernel-isolation framework that utilizes VMFUNC-based extended page table (EPT) switching [62]. We choose seL4 as it implements the fastest synchronous IPC across several modern microkernels [58]. We configure seL4 without meltdown mitigations. On the c220g2, server seL4 achieves the cross-domain invocation latency of 834 cycles (Table 1).

| 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 |

**Table 1:** Language-based cross-domain invocation vs hardware isolation mechanisms.

Recent Intel CPU introduces two new hardware isolation primitives—memory protection keys (MPK) and EPT switching with VM functions—provide support for memory isolation with overheads comparable to system calls [83] (99-105 cycles for MPK [38, 83] and 268-396 cycles for VMFUNC [38, 58, 62, 83]). Unfortunately, both primitives require complex mechanisms to enforce isolation, e.g., binary rewriting [58, 83], protection with hardware breakpoints [38], execution under control of a hypervisor [54, 58, 62]. Moreover, since neither MPK nor EPT switching are designed to support isolation of privileged ring 0 code, additional techniques are required to ensure isolation of kernel subsystems [62].

To compare the performance of EPT-based isolation with language-based techniques we use in RedLeaf, we configure LVDs, a recent EPT-based kernel isolation framework [62] to perform ten million cross-domain invocations and measure the latency in cycles with the `RDTSC` instruction. In LVDs, the cross-domain call relies on the VMFUNC instruction to switch the root of the EPT and selects a new stack in the callee domain. LVDs, however, require no additional switches of a privilege level or a page-table. A single VMFUNC instruction takes 169 cycles, while a complete call/reply invocation takes 396 cycles on the c220g2 server (Table 1).

In RedLeaf, a cross-domain call is initiated by invoking

[2] RedLeaf is available at `https://mars-research.github.io/redleaf`.

the trait object provided by the proxy domain. The proxy domain uses a microkernel system call to move the thread from the callee to the caller domain, creates continuation to unwind the thread to the entry point in case the invocation fails, and invokes the trait of the callee domain. On the return path, a similar sequence moves the thread from the callee domain back into the caller. In RedLeaf, a null cross-domain invocation via a proxy object (Table 1) introduces an overhead of 124 cycles. Saving the state of the thread, i.e., creating continuation, takes 86 cycles as it requires saving all general registers. Passing one `RRef<T>` adds an overhead of 17 cycles as `RRef<T>` is moved between domains. To understand the low-level overhead of transparent recovery, we measure the latency of performing the same invocation via a shadow domain. In case of a shadow the invocation crosses two proxies and a user-built shadow domain and takes 286 cycles due to additional crossing of proxy and shadow domains.

Most recent Intel CPUs implement support for ring 0 enforcement of memory protection keys, protection keys supervisor (PKS) [3], finally enabling low-overhead isolation mechanism for the privileged kernel code. Nevertheless, even with low-overhead hardware isolation mechanisms, a zero-copy fault-isolation scheme requires ownership discipline for shared objects that arguably requires support from the programming language, i.e., either a static analysis [39] or a type system that can enforce single-ownership.

**Overheads of Rust** Memory safety guarantees of Rust come at a cost. In addition to the checks required to ensure safety at runtime, some Rust abstractions have a non-zero runtime cost, e.g., types that implement interior mutability, option types, etc. To measure the overheads introduced by Rust language itself, we develop a simple hash table that uses an open-addressing scheme and relies on the Fowler–Noll–Vo (FNV) hashing function with linear probing to store eight byte keys and values. Using the same hashing logic, we develop three implementations: 1) in plain C, 2) in idiomatic Rust (the style encouraged by the Rust programming manual), and 3) in C-style Rust that essentially uses C programming idioms but in Rust. Specifically, in C-style Rust, we avoid 1) using higher-order functions and 2) the `Option<T>` type that we utilize in the idiomatic code to distinguish between the occupied and unoccupied entries in the table. Without the `Option<T>` type that adds at least one additional byte to the key-value pair, we benefit from a tight, cache-aligned representation of key-value pairs in memory to avoid additional cache misses. We vary the number of entries in the hash table from $2^{12}$ to $2^{26}$ and keep the hash-table 75% full. On most hash table sizes, our implementation in idiomatic Rust remains 25% slower than the one in plain C, whereas C-style Rust performs equal to or even better than plain C, although by only 3-10 cycles (Figure 5). We attribute this to a more compact code generated by the Rust compiler (47 instructions on the critical get/set path in C-style Rust versus 50 instructions in C).

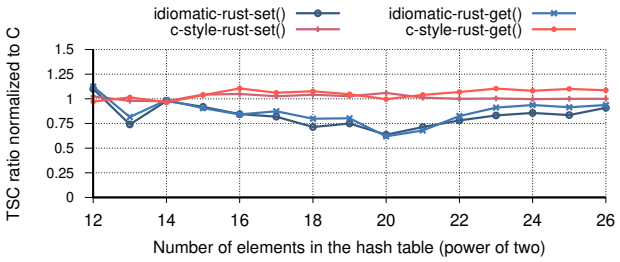


**Figure 5:** C vs Rust performance comparison

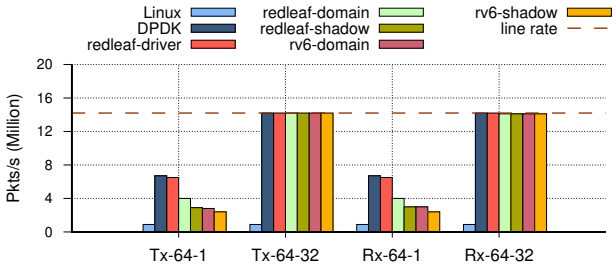


**Figure 6:** Ixgbe driver performance

## 5.2 Device Drivers

One of the critical assumptions behind RedLeaf is that Rust's safety is practical for development of the fastest subsystems of a modern operating system kernel. Today, operating with latencies of low hundreds of cycles per I/O request, device drivers that provide access to high-throughput I/O interfaces, network adapters and low-latency non-volatile PCIe-attached storage, have the tightest performance budgets among all kernel components. To understand if overheads of Rust's zero-cost abstractions allow the development of such low-overhead subsystems, we develop two device drivers: 1) an Intel 82599 10Gbps Ethernet driver (Ixgbe), and 2) an NVMe driver for PCIe-attached SSDs.

### 5.2.1 Ixgbe Network Driver

We compare the performance of RedLeaf's Ixgbe driver with the performance of a highly-optimized driver from the DPDK user-space packet processing framework [21] on Linux. Both DPDK and our driver work in polling mode, allowing them to achieve peak performance. We configure RedLeaf to run several configurations: 1) `redleaf-driver`: the benchmark application links statically with the driver (this configuration is closest to user-level packet frameworks like DPDK; similarly, we pass-through the Ixgbe interface directly to the RedLeaf); 2) `redleaf-domain`: the benchmark application runs in a separate domain, but accesses the driver domain directly via a proxy (this configuration represents the case when the network device driver is shared across multiple isolated applications [38]); 3) `rv6-domain`: the benchmark application runs as an Rv6 program, it first enters the Rv6 with a system call and then calls into the driver (this configuration is analogous to a setup of a commodity operating system kernel in which user

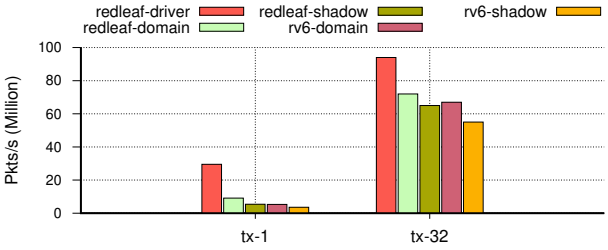


**Figure 7:** Software-only `nullnet` driver performance

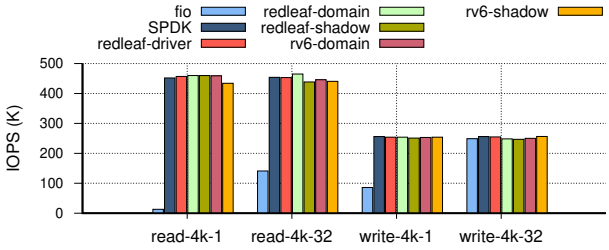


**Figure 8:** Performance of the NVMe driver

applications access I/O interfaces via a kernel network stack). Further, we run the last two configurations with and without the shadow driver (`redleaf-shadow` and `rv6-shadow`), which introduces an additional domain crossing into the shadow (these two configurations evaluate overheads of the transparent driver recovery). In all our tests, we pin the application thread to a single CPU core.

We send 64 byte packets and measure the performance on two batch sizes: 1 and 32 packets (Figure 6). For packet receive tests, we use a fast packet generator from the DPDK framework to generate packets at line-rate. On packet transmit and receive tests, Linux achieves 0.89 Mpps due to its overly general network stack and synchronous socket interface (Figure 6). On a batch of one, DPDK achieves 6.7 Mpps and is 7% faster than RedLeaf (6.5 Mpps) for both RX and TX paths (Figure 6). On a batch of 32 packets, both drivers achieve the line-rate performance of a 10GbE interface (14.2 Mpps). To understand the impact of cross-domain invocations, we run the benchmark application as a separate domain (`redleaf-domain`) and as an Rv6 program (`rv6-domain`). The overhead of domain crossings is apparent on a batch size of one, where RedLeaf can send and receive packets at the rate of 4 Mpps per-core with one domain crossing (`redleaf-domain`) and 2.9 Mpps if the invocation involves shadow domain (`redleaf-shadow`). With two domain crossings, the performance drops to 2.8 Mpps (`rv6-domain`) and 2.4 Mpps if the driver is accessed via a shadow (`rv6-shadow`). On a batch of 32 packets, the overhead of domain crossings disappears as all configurations saturate the device.

**Nullnet** To further investigate the overheads of isolation without the limits introduced by the device itself, we develop a software-only `nullnet` driver that simply returns the packet to the caller instead of queuing it to the device (Figure 7). On a batch of one, the overheads of multiple domain crossings limit the theoretical performance of `nullnet` driver from 29.5 Mpps per-core that can be achieved if the application is linked statically with the driver (`redleaf-driver`) to 5.3 Mpps when `nullnet` is accessed from the Rv6 application (`rv6-domain`). Adding a shadow driver lowers this number to 3.6 Mpps (`rv6-shadow`). Similarly, on a batch of 32 packets, `nullnet` achieves 94 Mpps if the applicaiton is run in the same domain as the driver. The performance drops to 67 Mpps when the benchmark code runs as an Rv6 application (`rv6-domain`), and to 55 Mpps if the Rv6 application involves a shadow driver (`rv6-shadow`).

### 5.2.2 NVMe Driver

To understand the performance of RedLeaf's NVMe driver, we compare it with the multi-queue block driver in the Linux kernel and a well-optimized NVMe driver from the SPDK storage framework [42]. Both SPDK and RedLeaf drivers work in polling mode. Similar to Ixgbe, we evaluate several configurations: 1) statically linked (`redleaf-driver`); 2) requiring one domain crossing (`redleaf-domain`); and 3) running as an Rv6 user program (`rv6-domain`). We run the last two configurations with and without the shadow driver (`redleaf-shadow` and `rv6-shadow`). All tests are limited to a single CPU core.

We perform sequential read and write tests with a block size of 4KB on a batch size of 1 and 32 requests (Figure 8). On Linux, we use `fio`, a fast I/O generator; on SPDK and RedLeaf, we develop similar benchmark applications that submit a set of requests at once, and then poll for completed requests. To set an optimal baseline for our evaluation, we chose the configuration parameters that can give us the fastest path to the device. Specifically, on Linux, we configure `fio` to use the asynchronous `libaio` library to overlap I/O submissions, and bypass the page cache with the `direct` I/O flag.

On sequential read tests, `fio` on Linux achieves 13K IOPS and 141K IOPS per-core on the batch size of 1 and 32 respectively (Figure 8). On a batch size of one, the RedLeaf driver is 1% faster (457K IOPS per-core) than SPDK (452K IOPS per-core). Both drivers achieve maximum device read performance. SDPK is slower as it performs additional processing aimed at collecting performance statistics on each request. On a batch size of 32, the RedLeaf driver is less than 1% slower (453K IOPS versus 454K IOPS SPDK). On sequential write tests with a batch size of 32, Linux is within 3% of the device's maximum throughput of around 256K IOPS. RedLeaf is less than one percent slower (255K IOPS). Since NVMe is a slower device compared to Ixgbe, the overheads of domain crossings are minimal for both batch sizes. With one domain crossing, the performance even goes up by 0.7% (we attribute this to a varying pattern of accessing the doorbell register of the device that gets thrashed between the device and CPU).

## 5.3 Application Benchmarks

To understand the performance overheads of safety and isolation on application workloads, we develop several applications that traditionally depend on a fast data plane of the op-

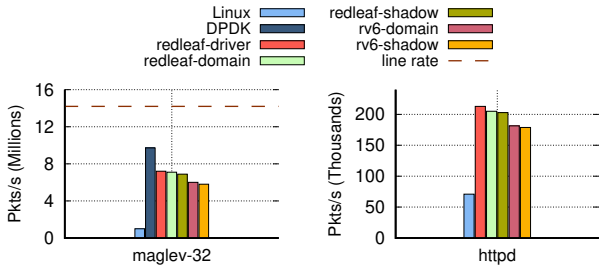


**Figure 9:** Performance of Maglev and Httpd

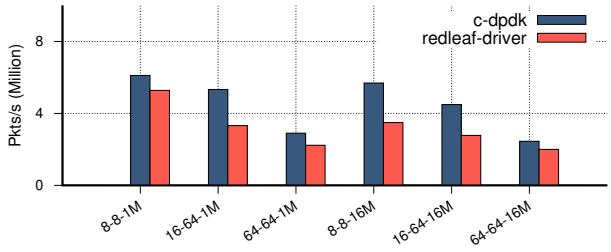


**Figure 10:** Key-value store

erating system kernel: 1) Maglev load balancer (`maglev`) [26], 2) a network-attached key-value store (`kv-store`), and 3) a minimal web server (`httpd`).

**Maglev load-balancer** Maglev is a load balancer developed by Google to evenly distribute incoming client flows among a set of backend servers [26]. For each new flow, Maglev selects one of the available backends by performing a lookup in a hash table, size of which is proportional to the number of backend servers (65,537 in our experiments). Consistent hashing allows even distribution of flows across all servers. Maglev then records the chosen backend in a hash table, a *flow tracking table*, that is used to redirect packets from the same flow to the same backend server. The size of the flow tracking table is proportional to the number of flows (we choose 1 M flows for our experiments). Processing a packet requires a lookup in the flow tracking table if it is an existing flow, or a lookup of a backend server and an insertion into the flow tracking table to record the new flow. To compare RedLeaf performance with both a commodity and the fastest possible setup, we develop C and Rust versions of the core Maglev logic. Moreover, we evaluate two C versions: one to run as a normal Linux program that uses the socket interface and another developed to work as a network function for the DPDK network processing framework [21]. In all versions we follow the same code logic and, if possible, apply the same optimizations. Again, on all setups, we restrict execution to one CPU core. Running as a Linux program, `maglev` is limited to 1 Mpps per-core due to the synchronous socket interface of the Linux kernel and a generic network stack (Figure 9). Operating on a batch of 32 packets, the `maglev` DPDK function is capable of achieving 9.7 Mpps per-core due to a well-optimized network device driver. Linked statically against the driver, RedLeaf application (`redleaf-driver`) achieves 7.2 Mpps per-core. Performance drops with additional domain crossings. Running as an Rv6 application, `maglev` can forward at 5.3 Mpps per-core without and 5.1 Mpps with the shadow domain.

**Key-value store** Key-value stores are de facto standard building blocks for a range of datacenter systems ranging from social networks [64] to key-value databases [23]. To evaluate RedLeaf's ability to support the development of efficient datacenter applications, we develop a prototype of a network-attached key-value store, `kv-store`. Our prototype is designed to utilize a range of modern optimizations similar to Mica [52], e.g., a user-level device driver like DPDK, partitioned design aimed at avoiding cross-core cache-coherence traffic, packet flow steering to guarantee that request is directed to the specific CPU core where the key is stored, no locks and no allocations on the request processing path, etc. Our implementation relies on a hash table that uses open addressing scheme with linear probing and the FNV hash function. In our experiments, we compare the performance of two implementations: a C version developed for DPDK, and a Rust version that executes in the same domain with the driver (`redleaf-driver`), i.e., the configuration that is closest to DPDK. We evaluate two hash table sizes: 1 M and 16 M entries with three sets of key and value pairs (`<8B,8B>`, `<16B,64B>`, `<64B,64B>`). The RedLeaf version is implemented in a C-style Rust code, i.e., we avoid Rust abstractions that have run-time overhead (e.g., `Option<T>`, and `RefCell<T>` types). This ensures that we can control the memory layout of the key-value pair to avoid additional cache misses. Despite our optimizations, RedLeaf achieves only 61-86% performance of the C DPDK version. The main reason for the performance degradation is that being implemented in safe Rust, our code uses vectors, `Vec<T>`, to represent packet data. To create a response, we need to extend this vector thrice by calling the `extend_from_slice()` function to copy the response header, key, and value into the response packet. This function checks if the vector needs to be grown and performs a copy. In contrast, the C implementation benefits from a much lighter unsafe invocation of `memcpy()`. As an exercise, we implemented the packet serialization logic with unsafe Rust typecast that allowed us to achieve 85-94% of the C's performance. However, we do not allow unsafe Rust inside RedLeaf domains.

**Web server** The latency of web page loading plays a critical role in both the user experience, and the rank of the page assigned by a search engine [15, 66]. We develop a prototype of a web server, `httpd`, that can serve static HTTP content. Our prototype uses a simple run-to-completion execution model that polls incoming requests from all open connections in a round-robin fashion. For each request, it performs request parsing and replies with the requested static web page. We compare our implementation with one of the de facto industry standard web servers, Nginx [63]. In our tests, we use the `wrk` HTTP load generator [1], which we configure to run with one thread and 20 open connections. On Linux, Nginx can serve

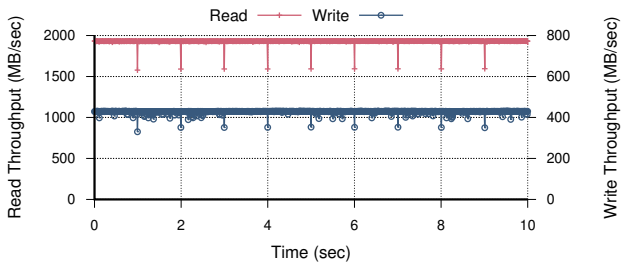


**Figure 11:** Block device recovery (FS write).

70.9 K requests per second, whereas our implementation of `httpd` achieves 212 K requests per second in a configuration where the application is run in the same domain as the driver (`redleaf-driver`) and network stack (Figure 9). Specifically, we benefit from low-latency access to the network stack and the network device driver. Running as an Rv6 domain, `httpd` achieves the rate of 181.4 K packets per second (178.9 K if it uses a shadow).

## 5.4 Device Driver Recovery

To evaluate the overheads introduced by the transparent device driver recovery, we develop a test in which an Rv6 program accesses the Rv6 file system backed by an in-memory block device. Running as an Rv6 program, the benchmark application continuously reads and writes files in the Rv6 file system using 4K blocks. The Rv6 file system accesses the block device via a shadow driver that can perform recovery of the block device in case of a crash. During the test, we trigger a crash of the block device driver every second (Figure 11). Automatic recovery triggers a small drop in performance. For reads, the throughput with and without restarts averages at 2062 MB/s and 2164 MB/s respectively (a 5% drop in performance). For writes, the total throughput averages at 356 MB/s with restarts and 423 MB/s without restarts (a 16% drop in performance).

# 6 Related Work

Several recent projects use Rust for building low-level high-performance systems, including data storage [33, 47, 60], network function virtualization [68], web engine [74], and several operating systems [17, 24, 50, 51], unikernels [49] and hypervisors [4, 36, 40]. Firecracker [4], Intel Cloud Hypervisor [40], and Google Chrome OS Virtual Machine Monitor [36] replace Qemu hardware emulator with a Rust-based implementation. Redox [24] utilizes Rust for development of a microkernel-based operating system (both microkernel and user-level device drivers are implemented in Rust, but are free to use unsafe Rust). The device drivers run in ring 3 and use traditional hardware mechanisms for isolation and system calls for communication with the microkernel. By and large, all these systems leverage Rust as a safe alternative to C, but do not explore the capabilities of Rust that go beyond type and memory safety.

Tock develops many principles of minimizing the use of unsafe Rust in a hardware-facing kernel code [50]. Tock is structured as a minimal core kernel and a collection of device drivers (capsules). Tock relies on Rust's language safety for isolation of the capsules (in Tock user applications are isolated with commodity hardware mechanisms). To ensure isolation, Tock forbids unsafe extensions in capsules but does not restrict sharing of pointers between capsules and the main kernel (this is similar to language systems using pointers as capabilities, e.g., SPIN [13]). As a result, a fault in any of the capsules halts the entire system. Our work builds on many design principles aimed at minimizing the amount of unsafe Rust code developed by Tock but extends them with support for fault isolation and dynamic loading of extensions. Similar to Tock, Netbricks [68] and Splinter [47] rely on Rust for isolation of network functions and user-defined database extensions. None of the systems provides support for deallocating resources of crashing subsystems, recovery, or generic exchange of interfaces and object references.

# 7 Conclusions

"A Journey, not a Destination" [39], Singularity OS laid the foundation for many concepts that influenced the design of Rust. In turn, by enabling the principles of fault isolation in Rust itself, our work completes the cycle of this journey. RedLeaf, however, is just a step forward, not a final design—while guided by principles of practicality and performance, our work is, first, a collection of mechanisms and an experimentation platform for enabling future system architectures that leverage language safety. Rust provides systems developers the mechanisms we were waiting for decades: practical, zero-cost safety, and a type system that enforces ownership. Arguably, the isolation that we implement is the most critical mechanism as it provides a foundation for enforcing a range of abstractions in systems with faulty and mistrusting components. By articulating principles of isolation, our work unlocks future exploration of abstractions enabled by the isolation and safety: secure dynamic extensions, fine-grained access control, least privilege, collocation of computation and data, transparent recovery, and many more.

# Acknowledgments

We would like to thank USENIX ATC 2020 and OSDI 2020 reviewers and our shepherd, Michael Swift, for numerous insights helping us to improve this work. Also, we would like to thank the Utah CloudLab team, and especially Mike Hibler, for his continuous support in accommodating our hardware requests. We thank Abhiram Balasubramanian for helping with RedLeaf device drivers and Nivedha Krishnakumar for assisting us with low-level performance analysis. This research is supported in part by the National Science Foundation under Grant Numbers 1837051 and 1840197, Intel and VMWare.

## References


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

[2] Erlang on Xen. http://erlangonxen.org/, 2012.

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 419–434, 2020.

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 3, pages 147:1–147:30.

[6] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[7] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and 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, 2017.

[8] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. Verifying Rust Programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 11138 of *Lecture Notes in Computer Science*, pages 528–535. Springer, 2018.

[9] Fred Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A Raw-Metal occam Experiment. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 182–196, September 2003.

[10] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A Fork() in the Road. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, page 14–22, 2019.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 49–65, October 2014.

[12] D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., March 1976.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, page 267–283, 1995.

[14] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and 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, 2007.

[15] Google Webmaster Central Blog. Using site speed in web search ranking. https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html.

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

[17] Kevin Boos and Lin Zhong. Theseus: A State Spill-Free Operating System. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*, page 29–35, 2017.

[18] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[19] Hank Bromley and Richard Lamson. *LISP Lore: A Guide to Programming the Lisp Machine*. Springer Science & Business Media, 2012.

[20] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys '11)*, pages 5:1–5:5, 2011.

[21] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[22] Cody Cutler, M Frans Kaashoek, and Robert T Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th*

*USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 89–105, 2018.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and 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, 2007.

[24] Redox Project Developers. Redox - Your Next(Gen) OS. http://www.redox-os.org/.

[25] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[26] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 523–535, March 2016.

[27] Kevin Elphinstone and 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, 2013.

[28] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[29] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and 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, 2006.

[30] Manuel Fahndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 13–24, 2002.

[31] Bryan Ford and 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, 1994.

[32] Lester J Fraim. Scomp: A Solution to the Multilevel Security Problem. *Computer*, 16(07):26–34, July 1983.

[33] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, page 213–231, 2018.

[34] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. *Addison-Wesley Longman Publishing Co., Inc.*, 1983.

[35] Michael Golm, Meik Felser, Christian Wawersich, and 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, 2002.

[36] Google. Google Chrome OS Virtual Machine Monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm.

[37] Haskell Lightweight Virtual Machine (HaLVM). http://corp.galois.com/halvm.

[38] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and 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, July 2019.

[39] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[40] Intel. Cloud Hypervisor VMM. https://github.com/cloud-hypervisor/cloud-hypervisor.

[41] Intel. Side Channel Mitigation by Product CPU Model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html.

[42] Intel Corporation. Storage Performance Development Kit (SPDK). https://spdk.io.

[43] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang.

Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference (ATC '02)*, pages 275–288, June 2002.

[44] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the ACM on Programming Languages (POPL)*, volume 2, pages 1–34, 2017.

[45] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.

[46] Kenneth C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–624, October 1965.

[47] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 627–643, October 2018.

[48] Butler W. Lampson and 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)*, page 98–105. 1979.

[49] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS'19)*, page 8–15, 2019.

[50] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, page 234–251, 2017.

[51] Alex Light. Reenix: Implementing a Unix-like operating system in Rust. Undergraduate Honors Theses, Brown University, 2015.

[52] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 429–444, April 2014.

[53] John Lions. *Lions' commentary on UNIX 6th edition with source code*. Peer-to-Peer Communications, Inc., 1996.

[54] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, page 1607–1619, 2015.

[55] Peter W Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JavaOS: A Standalone Java Environment. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[56] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, March 2013.

[57] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, page 115–128, 2011.

[58] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*, 2019.

[59] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[60] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, Iterative Data Processing with Timely Dataflow. *Communications of the ACM*, 59(10):75–83, September 2016.

[61] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 269–284, July 2019.

[62] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, page 157–171, 2020.

[63] Nginx. Nginx: High Performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com/.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, April 2013.

[65] Oreboot developers. Oreboot. https://github.com/oreboot/oreboot.

[66] Addy Osmani and Ilya Grigorik. Speed is now a landing page factor for Google Search and Ads. https://developers.google.com/web/updates/2018/07/search-ads-speed.

[67] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, page 305–318, 2011.

[68] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 203–216, November 2016.

[69] Matthew Parkinson. Digital Security by Design: Security and Legacy at Microsoft. https://vimeo.com/376180843, 2019. ISCF Digital Security by Design: Collaboration Development Workshop.

[70] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 1–16, October 2014.

[71] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, 1980.

[72] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[73] Robert Morris Russ Cox, Frans Kaashoek. Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2019/xv6.html, 2019.

[74] Servo, the Parallel Browser Engine Project. http://www.servo.org.

[75] Christopher Small and Margo I. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR 30-94, Harvard University, Division of Engineering and Applied Sciences, 1994.

[76] Marc Stiegler. The E Language in a Walnut, 2000. http://www.skyhunter.com/marcs/ewalnut.html.

[77] Jeff Vander Stoep. Android: protecting the kernel. *Linux Security Summit*, 2016.

[78] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. Recovering Device Drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[79] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[80] Daniel C Swinehart, Polle T Zellweger, Richard J Beach, and Robert B Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):419–490, 1986.

[81] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

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

[83] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, August 2019.

[84] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[85] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System

for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[86] Philip Wadler. Linear Types Can Change the World! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, 1990.

[87] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures (Extended Version). Technical Report TR2000-1787, Cornell University, March 2000.

[88] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Memory Management*, pages 1–42, 1992.

[89] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 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, 2005.

[90] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and 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, 2014.