Horizon: Practical Verified Kernels with Rust and Verus

Abstract

Recent advances in programming languages and automated formal reasoning have changed the balance of complexity and practicality of developing formally verified systems. Our work leverages Verus, a new verifier for Rust that uniquely combines the linear type system and automated verification based on satisfiability modulo theories (SMT), for the development of a formally verified microkernel, Horizon.

Horizon is a full-featured classical microkernel with a capability interface which mediates access to all resources of the system and controls inter-process communication. In contrast to many microkernels, however, Horizon is designed as a separation kernel with a formal proof of isolation, i.e., users can deploy mixed-criticality systems in which critical subsystems are isolated from non-critical parts while allowing controlled communication to shared subsystems. To support isolation, Horizon implements the abstraction of a "container" – a group of processes with a guaranteed memory and CPU reservation.

We develop all code in Rust and prove its functional correctness with Verus. Horizon supports proofs of the kernel's functional correctness as well as allows users to develop custom proofs of isolation and noninterference between containers. Development and verification of 4.9K lines of code required an effort of less than a 1.5 person-years (another person-year was spent developing non-verified parts of the system). On average, our code has proof-to-code ratio of 3.18:1 which we argue is practical for development of verified systems.

1 Introduction

Despite decades of progress, development of formally-verified operating systems remains a challenging undertaking that relies on a strong verification expertise and all too often requires years of human effort [29, 22, 49, 26, 25]. Correctness of the kernel rests on numerous low-level invariants about all data structures in the kernel and cover everything from lifetimes of manually managed data structures, correctness of pointer operations in hierarchical data structures like queues, lists, trees, and page tables to high-level contracts of the system call interface to cross-cutting properties that represent well-formedness of the system and high-level semantic invariants. In the past, development of a full-featured verified microkernel, seL4, required a heroic effort. Despite numerous careful design decisions verification of 8,700 lines of C code took 20 person-years and 165,000 lines of proof in Isabelle/HOL [29].

Fortunately, recent advances in programming languages and automated formal reasoning change the threshold of practical development of verified kernel code [31, 30]. Specifically, recent verifiers combine properties of linear types with automated verification based on satisfiability modulo theories (SMT) [31, 34, 30]. Linear types significantly lower the burden of reasoning about the heap due to a strict pointer aliasing discipline. Moreover, verifiers like Verus [31, 30] natively support verification of Rust, a programming language designed for systems development, hence enabling practical verification of driver code that can then be compiled and executed on bare metal.

Unique combination of linear type system and SMT-based verification allows us to re-think the practicality, i.e., complexity and burden, of developing the verified low-level systems code. We leverage Rust and Verus for demonstrating the possibility of developing a fully-verified microkernel, Horizon, at the speed and effort that, arguably, approach commodity unverified development.

Horizon is similar to classical microkernels: it runs on multi-CPU hardware, supports processes and threads, dynamic memory-management, inter-process communication (IPC), virtual address spaces and IOMMU. Instead of supporting *everything is a capability* model used in the line of seL4 microkernels, Horizon implements a simpler but, arguably, more ergonomic capability interface and a notion of "containers" – groups of processes that have a guaranteed reservation of system resources. To support fine-grained access control, Horizon implements capability interface for inter-process communication mechanisms, hardware devices and associated IOMMUs, but avoids a complex user-level memory management scheme used in seL4 [29]. Instead user processes are allowed to allocate memory as long as their parent container has available memory reserved on creation. Processes can share memory regions but are not able to revoke it forcibly like in seL4 [29]. This is a conscious choice aimed at enabling verification of user-level code (i.e., it's impossible to carry verification if mapped memory can be revoked at any time). Instead we rely on verified user-level proxies to control propagation of capabilities and volunteered release of resources. To support recursive revocation of capabilities, in Horizon revocation is done at the level of containers, i.e., resources of the container can be reclaimed when it's terminated.

We develop all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification, with Verus. At a high-level, we express functional correctness of the microkernel as two sets of specifications: system call specifications and global invariants. The system call specifications allow us to express the expected behavior of each system call as an effect on the high-level abstract model of the system. Global invariants allow us to express a broad collection of semantic invariants about all kernel data structures and the overall state of the system. For example, we prove memory safety of all pointer accesses and leak freedom, correctness of reference counting, page tables, scheduler, memory allocator, etc.

Despite impressive automation offered by Verus, several

design choices were critical for enabling verification of a semantically-complex codebase of a kernel and for ensuring speed and scalability of the proof.

First, in contrast to some prior work that relied on SMT-based verification [39], we did not have to simplify internal logic of the kernel or sacrifice traditional optimizations typical for kernel development to keep verification complexity under control. Moreover, despite using Rust, we develop Horizon without relying on any unverified types from the Rust standard library, e.g., vectors, reference counting types like `Rc<T>`, etc. The design choice we make is that we abandon rules of canonical Rust and instead use it in a manner similar to C, i.e., we develop all data structures using unsafe raw pointers like in C and avoid using any high-level language abstractions. This allows us to design performance-efficient data structures with all optimizations typical for unsafe C, e.g., our container tree uses internal storage, we use reverse pointers to support efficient, constant-time deletion from linked lists, etc. In contrast to C, however, we establish safety and correctness of all pointer operations. We do this through a combination of *tracked permission pointers* (an abstraction offered by Verus for verifying pointer operations) and a collection of carefully designed safety and correctness invariants that provide necessary preconditions for the proofs.

Second, Horizon involves proofs about complex hierarchical data structures like page tables, container trees, lists of processes and threads, data structures with manual and reference counted lifetimes, etc. Not surprisingly, the wide-spread use of pointers and the need to prove safety and correctness of all pointer accesses is challenging. At a high-level, in Verus each access through a pointer is mediated by a linear permission variable. Careless use of linear permission pointers blows up complexity of the proof. To keep complexity of the proof under control, we adapt a "flat" permission design in which permission pointers to inner objects in the hierarchy, e.g., all nodes of the page table, all threads in the system (irrespectively of their parent process and container) are stored at the topmost level of the system that requires a global view of all objects. Such flat approach allows us to develop recursive data structures like page tables and container trees with high proof automation and low verification time.

Third, throughout the kernel Horizon relies on manual memory management (i.e., we do not rely on the Rust borrow checker for reasoning about object lifetimes) Instead we develop proofs of safety (e.g., all heap accesses are type and memory safe as well as objects are alive) and leak freedom (we deallocate all memory correctly). To support preconditions required for the proof, we combine a proof technique that hierarchically maintains safety properties of all pointers with a minimal dynamic memory allocation scheme that allows us to reason about internal state of the allocator.

Fourth, we design Horizon as a separation kernel that can allow users of the system to construct proofs of isolation and controlled communication between isolated subsystems. Specifically, we demonstrate a possibility of an isolation and noninterference proof for a system in which two untrusted containers can establish controlled communication to a verified container. At a high level our use case illustrates the possibility of using Horizon for deployment of mixed-criticality systems in which mission critical and non-critical processes are isolated but can still stablish communication with verified subsystems that may mediate communication to shared subsystems, e.g., device drivers. To support such user-level proofs of isolation and noninterference, we design isolation specifications for Horizon system calls.

Our experience shows that a collection of careful design choices combined with the level of automation provided by Verus enable practical development of formally verified kernels. In total, development of Horizon took less than one and a half physical years and an effort of roughly two and a half person-years. But only one and a half person-years was spent on development of the verified parts (we used the second person-year on unverified parts such as the boot and initialization infrastructure, user-level device drivers, application benchmarks, and build environment). For the microkernel, we developed 4.9K lines of executable code and relied on 15.7k lines of proof code (11.4K lines of specifications, and 4.3K lines of hints to the verifier and proofs). On average our code has proof-to-code ratio of 3.18:1 which is lower than comparable prior approaches [29, 22].

2 System Architecture

Horizon is a full-featured microkernel conceptually similar to the family of L4 microkernels with an object capability interface [29]. Similar to other microkernels, Horizon pushes most kernel functionality to user-space, e.g., device drivers, network stacks, file systems, etc. The microkernel supports a minimal set of mechanisms to implement address spaces, threads of execution, memory management, interrupt dispatch, interprocess communication, IOMMUs and containers – groups of user processes with guaranteed memory and CPU reservation.

In Horizon processes can communicate via endpoints. A sender thread can pass scalar data, references to memory pages, IOMMU identifiers, and references to other endpoints. A receiver thread must be waiting on the endpoint for the message transfer to happen. If no receivers are waiting, the sender gets enqueued on the endpoint until the first receiver arrives. The endpoint supports the queue of senders and receivers. Endpoints allow processes to establish regions of shared memory, which provide support for efficient communication [19, 12, 4]. Endpoints also provide notification mechanism that allows us to avoid polling on shared memory, i.e., a thread can wait on an endpoint for notification from other threads.

Horizon is a multiprocessor system, but to simplify verification we rely on a big-lock synchronization, i.e., all interrupts and system calls execute in the microkernel under one global lock and with further interrupts disabled.

Containers To support provable separaton between group of processes, Horizon implements an abstraction of container. A container is a group of processes that have a guaranteed reservation of memory (*container quota*) and a collection of CPU cores on which container can be scheduled (to enforce temporal isolation). Containers form a recursive parent-child tree (one container can create multiple child containers recursively). Each memory allocation from any process in the container (a user allocated page or a page allocated by the kernel to store information about a new thread of execution) is accounted towards the container's memory quota. If a container creates a child container it passes a subset of its memory quota (and its available CPU cores) to the child.

Revocation Several importand design choices differentiate Horizon from the line of L4 microkernels. We design Horizon to support proofs of about correctness of user code. Hence, we make a design choice of not allowing revocation of user-mapped memory – we do not see a feasible way to carry a proof about the code that accesses memory pages that can disappear from the address space in between any two instructions. In contrast to seL4 which tracks propagation of memory through the system and allows recursive revocation of memory by any process in the capability derivation chain, in Horizon pages *cannot be forcibly unmapped*.

To implement revocation of resources, Horizon relies on a combination containers and user code logic that mediates communication between untrusted containers. Resources return to a parent container when a child is terminated, however, only if they were not passed outside of the container via an endpoint.

Such approach migh seem limiting: if the system is configured to allow leakage of resources they cannot be reclaimed. However, we observe, that in practical system organizations, such mediation is natural. Untrusted containers communicate with trusted (or verified) parts of the system, e.g., device drivers, that release resources when the process on the other end of communication channel terminates. Even in case when multiple mistrusting containers communicate directly, it's hard to imagine a realistic scenario when one container refuses to release the resources of another container if it crashes and tries to restart.

In our running example which we discuss in Section 3.6, two mistrusting containers A and B establish communication with the verified container V (Figure 1). We establish functional correctness of V and proof that it releases all pages used for communication with unverified containers in case one of them crashes.

3 Verification

We prove functional correctness of the Horizon microkernel and demonstrate how to construct an example of a user-level proof of isolation and non-interference between containers.

Functional correctness We express functional correctness of the microkernel as two sets of specifications: system call specifications and global invariants. The system call specifications allow us to express the expected behavior of each system call as an effect on the high-level abstract model of the system (Listing 1 line 1). For example, we model the state of the page table as a logical map from virtual addresses to physical addresses along with access rights (read-only, write, etc.) Then the system call specification of a system call describes how it changes this abstract state. For example, `syscall_mmap_spec()` ensures that each virtual address covered by the range of virtual addresses the user intended to map will be assigned with a newly allocated physical page and other virtual addresses remain unchanged (Listing 1 line 8).

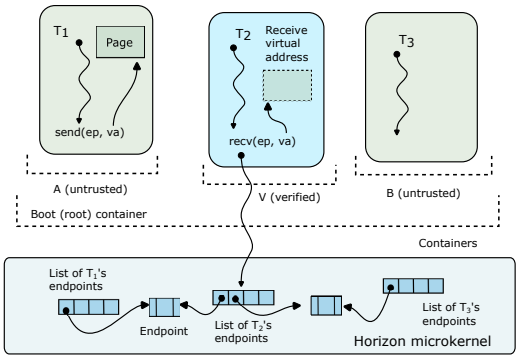

Figure 1: Architecture of Horizon. Containers A and B are untrusted but can establish communication with a verified container V. Thread T_1 invokes the `send()` system call to pass a page to T_2 which is already waiting on the endpoint inside the microkernel.

Global invariants allow us to express a broad collection of semantic invariants about all kernel data structures and the overall state of the system. We discuss most challenging invariants in the sections below, but at a high level, the invariants cover memory safety for all pointers in the kernel as well as correctness of memory deallocation , wellformedness and semantic correctness of page tables , CPU isolation between containers, wellformedness of the memory manager , endpoints, threads, and the container tree, etc.

Noninterference Similar to prior work [17, 22], we utilize unwinding for development of non-interference specifications [38]. Specifically, to prove noninterference, we combine output consistency (OC), i.e., the property that equivalent kernel states are indistinguishable from the results of the system calls; weak step consistency (WSC), i.e., that system calls do not leak information to containers; and local respect (LR), i.e. one container cannot affect another via any of the system calls [38]. To prove these unwinding conditions, we develop a separate proof (Section 3.6) that utilizes Horizon system call specifications and combines them with the proof about behavior of shared subsystems to demonstrate that isolated containers cannot affect each other and remain isolated in terms of their ability to establish communication channels, share memory, etc.

```
pub open spec fn syscall_mmap_spec(old:Kernel, new:Kernel,
      t_id: ThreadPtr, va_range: VaRange4K, perm_bits:
      MapEntryPerm, ret: SyscallReturnStruct) -> bool{
  ...
  if syscall_mmap_return_value(old, t_id, va_range).is_err(){
    new =~= old
  }else{
    ...
    // Virtual addresses outside of va range are not changed
    &&& forall|va:VAddr|
      va_range@.contains(va) == false
      ==> new.get_address_space(proc_ptr).dom().contains(va)
          == old.get_address_space(proc_ptr).dom().contains(va)
                && new.get_address_space(proc_ptr)[va]
                  =~= old.get_address_space(proc_ptr)[va]
    //Newly allocated pages were free pages
    &&& forall|page_ptr:PagePtr|
       mmapped_physical_pages_seq.contains(page_ptr)
       ==> old.page_is_free(page_ptr)
    //Each virtual address in va range gets a new page
    &&& forall|i:usize| 0<=i<va_range.len
       ==> new.get_address_space(proc_ptr)[va_range@[i]].addr
         == mmapped_physical_pages_seq[i]}}
```

Listing 1: Abstract specifications of mmap()

3.1 Flat, Pointer-Centric Design

In Horizon we make a design choice to use raw pointers in a manner similar to C, i.e., develop recursive pointer data structures like linked lists, and the container tree, and in general freely use poitners to implement complex logical dependencies between kernel data structures as well as typical kernel optimizations. Arguably, free use of pointers allows us to match performance of unsafe, unverified code as well as develop Horizon like we would do in an unsafe language.

Not surprisingly, establishing correctness of pointer operations is challenging. In Verus, permissioned pointers abstract the unique (linear) right to access the pointer. Operations on permission is therefore affect the scalability of the proof.

Intuitively, one should follow the rules of modular programming and store permissions next to the pointers they control. For example, threads of a process are exclusively owned by the process, processes are owned by containers, etc. Hence the process should store linear permissions to access its threads. Surprisingly such hierarchical approach limits the scalability of the proof. Internal invariants of complex data structures and the global system-wide invariants require reasoning about entire hierarchy, i.e., all threads in the system, all children of a container, etc. With hierarchical ownership one has to navigate the hierarchy to retrieve the state of inner nodes. Maintaining specifications and developing proofs with encapsulated ownership resulted in several times more effort, and in some cases was becoming unmanageable with the nesting of more than two levels. We experienced a similar growth of verification time. In our early phases, with nested permissions, implementation of just processes, threads, and the scheduler required beyond acceptable verification time. The key design choice we made was to implement the "flat" approach for managing permissions in complex recursive data structures, e.g., lists of processes and threads, page tables, etc. For most subsystems, we use the recursive pointer data structures, e.g., linked-lists, container trees, entries of the page table, etc., to store the raw pointers to other objects but not permissions. Permissions are kept at the level of the system which provides the global view of the entire subsystem. For example, the process manager holds permissions to the threads of all processes (`thread_perms`), the page table stores permissions for all the levels of the page table (Listing 2).

```
pub struct PageTable{
    pub cr3: PagePtr,
    pub l4_table: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
    pub l3_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
    pub l2_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
    pub l1_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
    pub l3_rev_map: Ghost<Map<PagePtr, (L4I)>>,
    pub l2_rev_map: Ghost<Map<PagePtr, (L4I,L3I)>>,
    pub l1_rev_map: Ghost<Map<PagePtr, (L4I,L3I,L2I)>>,}
```

Listing 2: Page table definition

3.2 Memory allocation

In Horizon dynamic memory allocation for kernel objects, e.g., containers, processes, threads, endpoints, is done at the granularity of 4KB memory pages. While it seems wasteful in practice the system does not allocate a large number of objects. The simplicity of the allocator, however, allows us to reason about non-interference between containers as well as simplify specifications that expose internal state of the allocator required for maintaining safety and leak freedom invariants. In addition to 4KB pages, we support allocation of 2MB and 1GB superpages to support construction of large address spaces with low TLB pressure.

The page allocator uses a page array (similar to the page array in Linux) to maintain the metadata for each physical page in the system. We ensure that each physical page is in one of the following states: 1) *free* (on the list of free pages in the memory allocator), 2) *mapped*, (mapped by one or more processes), 3) *merged*, (merged to form a 2MB or 1GB superpage), or 4) *allocated* (allocated for one of the kernel data structures, e.g., a process).

The allocator uses three doubly-linked list to store the free pages of different sizes (4KB, 2MB, 1GB) We use the page metadata array to efficiently merge 4KB pages into superpages, e.g., to form a 2MB superpage out of free 4KB pages we scan the page array and remove merged 4KB pages from the list of free 4KB pages. Note that each element of the metadata array maintains a pointer to node of the liked list holding the page, which allows us to perform constant time removal when the page is merged.

To reason about the state of the allocator, we maintain three sets of pages in different state (one for each page size) and the mappings for all mapped pages. Note that the sets of free pages are abstracted from the linked lists of free pages. We use the aligned address of the page as its unique identifier. We then use a range of internal invariants to guarantee the internal correctness of the page allocator.

Correctness invariants We maintain correctness of the page states ensuring that pages will not be lost or allocated more than once. That is, bidirectionally, if a page is marked to be in a specific state, it is included in the corresponding abstract set of pages and vice versa. Hence we can prove that pages of different states do not overlap (memory safety) and all pages that are not mapped by user-level or allocated by the kernel are in the free lists (leakage freedom).

We also maintain the invariant that each superpage is formed correctly, i.e., by combining a range of 4KiB pages and the 4KB pages are correctly marked as merged. A page will be marked as "merged" if and only if it has been used to form a huge page, and for each huge page in the system, physical pages covered by its range must be marked as "merged" with correct size.

3.3 Safety and Leak Freedom

In a safe subset of Rust, Verus can guarantee memory safety via a combination of the linear type system and borrow-checked ghost permission pointers. An execution environment of a microkernel, however, is inherently unsafe – we freely use raw pointers to implement efficient data structures. We however establish safety and correctness of all pointer operation via a proof that utilizes Verus ghost permissions, correctness of reference counting, and manual memory management throughout the entire kernel. In addition, neither Rust nor Verus guarantees leakage freedom. However, in order to consider a microkernel as correct, leakage freedom cannot be avoided.

We define memory safety and leak freedom as a combination of: type safety (each allocated region of memory is used by exactly one data structure), spatial safety (we check that data types are within allocated memory, and all memory accesses to variable sized data structures like arrays are in-bounds), and temporal safety (all pointers are alive, i.e., memory is not deallocated or reallocated for another object). To prove leak freedom we show that all allocated memory is used by exactly one data structure (i.e., if the safety guarantees require us to prove that memory is used by *at-most one* data structure, for leak freedom we show that it is *exactly one*).

To prove type safety we need to prove that all allocated memory is used by disjoint objects, to prove leak freedom we need to proof that the sum of all memory used by all objects in the system equals allocated memory (i.e., a combination of pages in the "allocated", "mapped", and "merged" states). We prove functional correctness of the page mapping reference counting to ensure leak freedom for user-mapped pages, and a collection of well-formedness invariants of the page allocator to prove the correctness of merging and splitting super-pages. The challenging part, however, is to prove safety and leak freedom for kernel-allocated pages. For this, we need to establish safety, i.e.: 1) all the objects in the kernel are pair-wise disjoint in memory and 2) the sum of all memory used by all objects in the system is equal to the collection of "allocated" pages.

Bottom-up recursive memory reasoning A straightforward approach is to prove that all the objects in the kernel are pair-wise disjoint in memory and second to prove the sum of the memory used for storing these objects is equal to the collection of "allocated" pages. Such approach is challenging as it requires the knowledge of *all* objects (i.e., sizes and addresses) used in the kernel.

To keep track of all allocated objects, we adopt a hierarchical approach which lowers the proof complexity through proof modularity. Specifically, we leverage the fact that objects of individual types are used in the kernel in a hierarchical manner, i.e., memory pages used to form page tables are used in the memory ... subsystem, threads, processes, and containers are used by the process management subsystem. Then for each data type and for each instance we proof that the instances are disjoint.

In Horizon for each data structure in the kernel we implement a `page_closure()` specification function, which returns a set of pages used by the data structure and all objects owned by the data structures. Here, the ownership means either a direct ownership of Rust (e.g., an array of objects) or ownership via a permission pointer (permission pointers guarantee unique ownership of the pointer and hence memory dereferenced through the pointer). For example, the page table subsystem does not own any other objects, besides the physical pages used to form the page table. The memory management subsystem, however, owns all page tables and IOMMUs in the kernel. Therefore, we hierarchically proof that memory manager's page closure is equal to the sum of the page table closures of the page tables and IOMMUs, and their page closures are disjoint. Similarly, for the entire kernel we prove that all subsystems have their page closures disjoint and the sum of all pages in all page closures equals to all "allocated" pages.

Internal state of the memory allocator We expose the internal state of the allocator as sets of free, allocated, merged, etc. pages. Specifically, each time we allocate memory (i.e., a page) we need to establish that: 1) the page closure of the allocating subsystem is extended by the new freshly allocated page, 2) the set of total "allocated" pages is extended by this page, 3) the page was previously "free", i.e., not used by any other subsystem, and 4) other subsystems in the kernel are not changed.

Exact reasoning about the state of the allocator forces us to rely on manual memory management in the kernel. For example, the two lines of code which allocate a new endpoint:

```
1  let (p_ptr,mut p_perm) = self.page_alloc.alloc_page_4k();
2  self.proc_man.new_endpoint(p_ptr, p_perm, ...);
```

use the postcondition of the `alloc_page_4k()` and `new_endpoint()` functions. First, we establish that a newly allocated page was previously not allocated or mapped:

```
pub struct Container{
  pub parent: Option<ContainerPtr>, //Root has no parent
  pub children: StaticList<ContainerPtr>,
  pub depth:usize,
  // ghost fields
  pub resolve_path : Ghost<Seq<ContainerPtr>>,
  pub subtree: Ghost<Set<ContainerPtr>>,
  // other fields (used by the process manager)
}
pub struct ContainerTree{
   pub root_container: ContainerPtr,
   pub container_perms: Tracked<Map<ContainerPtr,
    PointsTo<Container>>,
   }
```

Listing 3: Container and container tree data structures

```
pub fn alloc_page_4k(...)
  ensures
    self.free_pages_4k() =~=
      old(self).free_pages_4k().remove(ret.0),
    self.allocated_pages_4k() =~=
      old(self).allocated_pages_4k().insert(ret.0),
```

This ensures that using this page to form a new endpoint object in the kernel is safe. Furthermore, the postcondition of the allocator ensures that the set of allocated pages grows by exactly this new page, and the postcondition of the process manager ensures that its `page_closure()` grows exactly by this new page as well:

```
pub fn new_endpoint(...)
  ensures
    self.page_closure() =~=
      old(self).page_closure().insert(page_ptr),
```

Since other subsystems do not change, the total memory used by all the kernel subsystems also provably grows by this new page. Therefore, we can prove that the memory safety and leakage freedom invariants still hold.

3.4 Container Tree

In Horizon, containers form a nested tree. This provides support for fine-grained delegation of resources (e.g., control over access to physical hardware, memory quotas, etc.) and recursive revocation via container termination. Parent containers have the ability to terminate their direct and indirect children hence triggering resource revocation. In addition, the container tree is designed to expose the state of the container hierarchy to the upper level subsystems and the application layer via the system call specifications. To capture recursive parent-child relationships between containers, we implement an unbounded doubly linked tree. Careful use of ghost permissions allows us to implement a tree with a compact internal storage, i.e., we maintain the tree within the container data structure itself (Listing 3). Each container node has a list of children (the list has pre-allocated fixed size storage) and a pointer to its parent. Similar to other complex data structures in the kernel, we use the flat design: the container tree stores the root pointer, and a flat map of ghost permissions to each container. The flat architecture provides a convenient way of exposing the state of each container to the upper-level process manager (e.g., via the `spec_get_container()` specification function). To support various aspects of the proof, we expose the parent-child relationships between containers as a ghost sequence of nodes that describes the path from the root to the current node `.resolve_path` and a ghost set of all nodes in container's subtree `.subtree_set`. Explicit path and a recursive set of children are critical for constructing user-level proofs about the properties of subtrees (e.g., in the isolation proof, we prove that all the address spaces in the subtree of one isolated container is disjoint with the address spaces in the subtree of another).

A tree has many properties, but most of them can by infered by a small set of critical properties. Keeping all the properties as invariants will introduce massive verification overhead, thus, in our flat design, we idenity the critical properties and use a range of internal invariants to prove these properties.

Specifically, there are three critical properties: 1) bidirectionally, iff the parent holds a child pointer to a child, the child must also hold the parent pointer to the parent. Child container has depth equals to the depth of the parent plus one. 2) the child's `resolve_path` equals to the parent's `resolve_path` plus the parent. 3) bidirectionally, iff a subchild's `resolve_path` includes a direct/indirect parent, the direct/indirect parent includes the subchild in the `subtree_set` .

These invariants (plus some other significant invariants) are enough to form a tree and prove about reachability, acyclicity, disjointness of subtrees, etc.

Despite the fact that the flat approach is advantageous for exposing container states to the upper-levels of the proof, we observed that the invariant 2) of the are prone to trigger loops and large SMT expansion. For example, a straightforward way of describing the `resolve_path` between the child and the parent:

```
for|child:container, parent:container|
  self.get_container(child).parent == parent
  ==> self.get_container(child).resolve_path
    =~= self.get_container(parent).resolve_path.push(parent)
    && self.get_container(child).depth
     == self.get_container(parent).depth + 1
```

It is obvious that a node's grandparent (its parent's parent) would have

```
self.get_container(child).resolve_path
  =~= self.get_container(grandparent).resolve_path
        .push(grandparent).push(parent)
```

Verus tends to do the same analysis and expends this invariant recursively when reasoning about the node and its indirect parents, causing a long verification time. We found that describing these invariants in a none-recursive way tends to have better proof automation and better verification time:

```
for|child:container, index:int|
  0 <= index < self.get_container(child).resolve_path.len()
       ==>
  self.get_container(child).resolve_path.subrange(0, index)
    =~= self.get_container(self.get_container(child).
          resolve_path[index]).resolve_path
  && index == self.get_container(
       self.get_container(child).resolve_path[index]).depth
```

Here, instead of relying on recursive reasoning, we directly express the invariants for a node against its direct and indirect parents.

3.5 Page table

We implement support for a 4-level page table with variable size pages: 4KiB, 2MiB, and 1GiB. The abstract state of the page table is represented by three maps (one for each page size) from virtual to physical addresses along with the permission bits.

```
pub closed spec fn mapping_4k(&self) -> Map<VAddr,MapEntry>
pub closed spec fn mapping_2m(&self) -> Map<VAddr,MapEntry>
pub closed spec fn mapping_1g(&self) -> Map<VAddr,MapEntry>
```

Refinement proof The page table uses a range of invariants to prove its correctness. For example, each entry in any PML level only maps to pages which the page table holds ghost permission to; each none zero entry in any PML level has the "present" bit set; etc. Most importantly, it proves refinement between its concrete state and its abstract mapping. For example, in mappings of 4KiB pages, we use four-level spec functions to simulate the address resolution of the MMU and prove that the `mapping_4k()` matches what the MMU will theoretically see.

```
forall|l4i: L4I,l3i: L3I,l2i: L2I,l1i: L2I|
  0<=l4i<512 && 0<=l3i<512 && 0<=l2i<512 && 0<=l1i<512
  ==>self.mapping_4k().contains(index2va((l4i,l3i,l2i,l1i)))
    == self.resolve_mapping_4k(l4i,l3i,l2i,l1i).is_Some()
forall|l4i: L4I,l3i: L3I,l2i: L2I,l1i: L2I|
  0<=l4i<512 && 0<=l3i<512 && 0<=l2i<512 && 0<=l1i<512
    && self.resolve_mapping_4k_l1(l4i,l3i,l2i,l1i).is_Some()
  ==> self.mapping_4k()[index2va((l4i,l3i,l2i,l1i))]
    =~= self.resolve_mapping_4k(l4i,l3i,l2i,l1i).unwrap()
```

The first for all statement ensures the domain of virtual addresses mapped in the concrete page table is equal to the domain of the abstract mapping. The second for all statement ensures the equivalency of the mappings.

Flat storage of page table structures Similar to other subsystems we leverage flat design for storing the permissioned pointers. We use ghost permissions for managing the ownerships of the page table structures, and four tracked ghost maps to store the permissions in a flat manner. Compared to the hierarchical design, we need extra invariants to make sure the correct structure of the page table, i.e., the page table is a tree which trivially comes from the linearity of permissions in a hierarchical design. These invariants, however, have are easy to maintain and have negligible verification overhead.

One of the most complicated page table proof is to prove that if the page table adds a new virtual to physical mapping, the abstract mapping of other virtual addresses does not change. NrOS has a verified page table that adopts the recursive approach and is similar in code size [42]. In NrOS, such a proof requires indirect recursive reasoning between different levels and requires around 200 lines of proof code in `map_frame_aux()` function [43]. In our design, we maintain a reverse map for each level that describes the resolve path from the root to the current PML table. Such reverse map would be very difficult to maintain and use in a recursive design, as PML1 does not have direct access to PML4, thus cannot reason about the correctness of the reverse map. Since we have direct access to the states of all PML level tables, we can easily express and prove the property that all other entries in all PML levels do not change, which is sufficient for the refinement proof:

```
pub fn map_4k_page(&mut self, dst_l4i:L4I, dst_l3i: L3I,
      dst_l2i: L2I, dst_l1i: L2I, va:VAddr, ...)
{ //executable code
  //other virtual addresses do not change
  assert(
  forall|l4i: L4I,l3i: L3I,l2i: L2I, l1i: L1I|
    0<=l4i<512 && 0<=l3i<512 && 0<=l2i<512 && 0<=l1i<512
    && ((dst_l4i, dst_l3i, dst_l2i, dst_l1i)
      != (l4i,l3i,l2i,l2i))
    ==> self.spec_resolve_mapping_l1(l4i,l3i,l2i,l1i) =~=
      old(self).spec_resolve_mapping_l1(l4i,l3i,l2i,l1i));
}
```

3.6 Noninterference

We design Horizon to support development of system configurations which require provable isolation and noninterference between containers with the possibility of establishing controlled communication to trusted (or in our case verified) subsystems. As an example, we develop a noninterference proof for a system with three containers: two untrusted, unverified and isolated containers A and B and a verified container V which is allowed to establish communication channels with processes from A and B (Figure 1). At a high level our use case illustrates the possibility of using Horizon for deployment of mixed-criticality systems in which mission critical and non-critical containers are isolated but can still establish communication with verified subsystems that may mediate communication to shared device drivers, etc.

We implement V as a container with one process which runs one thread of execution (naturally, this simplifies verification, but is also acceptable for containers that implement system security policy, e.g., mediate setup of communication channels between untrusted containers and system device drivers). We proof functional correctness of V, i.e., that it behaves according to its abstract specifications. For example, V only performs a few system calls that are required to establish communication channels with A and B, but will never allow A and B to exchange endpoints or share pages. We, however, make no assumptions about A and B. They are not trusted, not verified and thus can perform arbitrary system calls with arbitrary system call arguments to establish an explicit communication channel, observe state of the kernel, and even try to affect each other by exhausting system resources.

Memory and communication channel isolation We prove memory isolation between the two unverified containers and their subtrees. That is, the pages mapped by the address spaces of one unverified container or its reachable children are proven not to be mapped by the address spaces of another unverified container and its children. Additionally, to allow fast communication between V and A or V and B (and their subtrees), shared memory regions and endpoints can be established between V and A or V and B (and their subtrees).

Observable state In Horizon, a return value of a system call invoked by a thread is only dependent on the owning con-

```
pub open spec fn syscall_mmap_return_value(old:Kernel, t_ptr:
      ThreadPtr, va_range: VaRange4K) -> UserRetValueType{
   let p_ptr = old.proc_man.get_thread(t_ptr).owning_proc;
   let c_ptr = old.proc_man.get_proc(proc_ptr).
        owning_container;
   if old.get_quota(c_ptr) < va_range.len * 4{
      UserRetValueType::ErrorNoQuota
   }else if !old.address_space_range_free(p_ptr, &va_range){
      UserRetValueType::ErrorVaInUse
   }else{
     UserRetValueType::Success
   }
}
```

Listing 4: System call return value spec of mmap()

tainer, the owning process and its address space, and all the endpoints accessible from the thread's endpoint descriptors. Listing 4 shows that the return value of `mmap()` is determined by checking the address space of the process and the memory quota of the container. These states form the observable state of a thread, and we can form the observable state of containers and processes by collecting the observable state of threads owned by them.

Similar to prior systems [17, 22], to prove noninterference between containers A and B, we prove the following unwinding conditions:

- *Output consistency (OC):* Output consistency requires the kernel to provide the same system return value given two identical system states. In Horizon, the return value of system calls is only dependent on the state of the old kernel and system call arguments. Therefore, identical system state will produce identical system call return values. Thus, OC is trivial to prove.
- *Weak step consistency (WSC):* In our model, weak step consistency implies that the kernel does not leak information between containers when they performs arbitrary system calls. We prove this property by showing that the system call return value of each arbitrary system call from container B remains unaffected and the state of container B remains unchanged before and after an arbitrary system call from container A.
- *Local respect (LR):* We establish local respect via the postconditions of the system calls. An arbitrary system call executed by one container cannot affect the observable state of another container.

State machine model: To develop the proof, we model the system as a state machine that tracks the state of the kernel and transitions according to the abstract specifications of system calls. We maintain a range of invariants about the state machine to prove memory isolation and noninterference, and prove that regardless of the steps it takes, the invariants still hold. For example, the memory isolation invariant specifies that each page mapped by the address spaces of one unverified container and its subtree is not mapped by the address spaces of another. And each page mapped by address spaces of A or B or their subtrees is not mapped by any other address spaces other than V. To support these invariants between arbitrary system calls, we rely on a range of invariants on the sharing of endpoints that control communication and resource sharing between unverified container subtrees. Each endpoint that is accessible to one unverified container and its subtree is not accessible to another unverified container. This guarantee that unverified containers cannot establish new communication channels between each other and hence share pages.

Limitations In the noninterference proof we only reason about the effects of system calls and not the user code that despite being isolated on different CPUs can introduce timing channels in shared caches and other hardware resources. Also, that at the moment, Horizon allows execution of several long running system calls (e.g., operations like mapping and sending large regions of memory, or terminating containers and processes), which can block other processes due to a big lock in the kernel for a long period of time and leak timing information. This limitation can be removed in the future by bounding the size of the memory region size and by implementing iterative versions of the "kill" system calls.

4 Implementation

Build environment The Horizon microkernel consists of both *verified* and *non-verified* components built using a trusted compilation environment, e.g., Horizon relies on a trusted boot manager to initialize the system. Compilation of the microkernel is done in two passes. The build system first invokes the Verus toolchain on the *verified* components. Then a regular Rust toolchain is used to compile the entire kernel with ghost code erased. We use the same Rust version that the Verus toolchain is based on to minimize potential differences in code generated by the two toolchains.

Stack size analysis Verus cannot guarantee the absence of stack overflows since it does not model the hardware and relies on Rust to correctly abstract details of the machine executing the code. To ensure that micorkernel has sufficient stack space, we statically compute the maximum stack size that may be used by the microkernel on all possible execution paths. During compilation, Rust summarizes the stack sizes of individual functions. We rely on the LLVM bitcode and extract the call graph of the microkernel using an LLVM IR pass. Based on the call graph, we attempt to derive the upper bounds of stack usage for all entry points (e.g., the main function, system calls, and interrupt handlers). The binary is rejected if the upper bound cannot be found, which can occur in the presence of cycles in the call graph. The boot loader then allocates sufficiently big microkernel stacks on each CPU.

5 Evaluation

Primarily, we evaluate Horizon to answer the two main questions: 1) How practical is development of verified kernel code with Verus in terms of verification speed and development

Name	Language	Spec Lang.	Proof-to-Code Ratio
seL4	C+Asm	Isabelle/HOL	20:1 [29]
CertiKOS	C+Asm	Coq	14.9:1 [22]
SeKVM	C+Asm	Coq	6.9:1 [35]
Ironclad	Dafny	Dafny	4.8:1 [26]
NrOS	Rust	Verus	10:1 [3]
VeriSMo	Rust	Verus	2:1 [50]
Horizon	Rust	Verus	3.2:1

Table 1: Proof effort for existing verification projects.

System	1 thread	8 threads	Proof	Exec.	P/E Ratio
NrOS Pagetable	1m 52s	51s	5329	400	13.3
Horizon Pagetable	33s	–	2168	496	4.37
Mimalloc	8m 12s	1m 40s	13703	3178	4.3
VeriSMo	61m 24s	12m 11s	16101	7915	2.0
Horizon	3m 29s	1m 7s	15701	4937	3.18

Table 2: verification time of different systems on CloudLab c220g5

effort? 2) Can Horizon be used as a practical kernel which does not sacrifice performance for formal correctness? We conduct a range of experiments that evaluate complexity of verification, complexity of verified development, and evaluate performance of Horizon on typical datacenter workloads.

We conduct evaluation on the publicly-available CloudLab [44] machines: c220g5[6], c220g2[5], and d430[7]. We measure verification time on c220g5 which is configured with two Intel Xeon Silver 4114 10-core CPUs running at 2.20 GHz, 192 GB RAM. Network experiments use a pair of c220g2 machines with Intel X520 10Gb network interfaces. NVMe experiments utilize d430 which are configured with PCIe-attached 400GB Intel P3700 Series SSDs. All the machines run 64-bit Ubuntu 20.04 with a 5.4.0 kernel. To reduce variance in benchmarking, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling for all the experiments.

5.1 Verification complexity

To put verification effort of Horizon in perspective of prior work, we collect the proof-to-code ratio across several recent kernel projects (Table 1. Horizon has a proof-to-code ratio of 3.18:1 which is a significant improvement compared to the existing formally verified microkernels SeL4 [29] and CertiKOS [22], which have proof-to-code ratio of 20:1 and 14.9:1, respectively (Table 1). Interestingly, through, VeriSMo has an even lower proof-to-code ratio of 2.0. This is due to the fact that semantically VeriSMo is less complex than Horizon, i.e., it does not support leakage freedom, only support page table identity mapping, etc.

Verification time Rapid verification is essential for interactive development. On the c220g5 server, Horizon finishes full verification in about 1 minute 10 seconds with 8 threads, with the longest function (`container_tree_insert_leaf()`) finishing within 25 seconds (Figure 2). On a newer laptop with a recent CPU, Intel i9-13900hx, with 8 performance and 16

Operation	Horizon	seL4
Call/reply	1334	1339
Map a page	283	-
Create a process	6519	-
Create a thread	1862	-

Table 3: Latency of communication and typical system calls

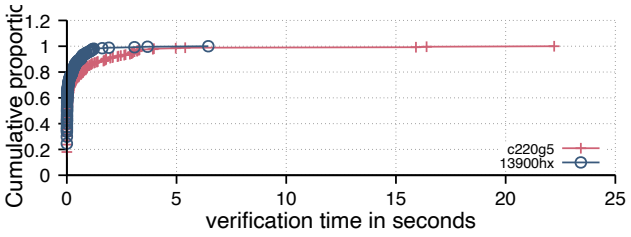

Figure 2: Verification time for each function

efficient cores running at a maximum frequency of 5.4GHz with hyper-threading, turbo boost, and frequency scaling enabled, Horizon finishes full verification in just 17 seconds (on 32 threads), and 47 seconds (on 1 thread). While it's hard to isolate impact of individual techniques, several techniques are critical for improving verification speed: flat design, closed specifications, careful manual triggers.

Impact of flat design To isolate the impact of our flat approach we compare proof-to-code ratio and verification times for the page table subsystems in NrOS and Horizon – both systems use Verus implement a comparable page table logic with similar verification goals, and similar size of executable code. Due to the flat design, compared to the page table from NrOS [3] which also uses Verus, Horizon page table has 3x lower proof to code ratio (13.3:1 and 4.4:1). Moreover, on a single thread, Horizon page table is over 3x faster than the NrOS page table.

5.2 Development speed and effort

We completed development of Horizon in three stages, or, more specifically, we developed three versions which were clean-slate rewrites borrowing design and implementation lessons from a previous version (Figure 3): 1) *version 1* (2 month, one person) resulted in a simplistic kernel centered around the process manager and page allocator which was aimed primarily at familiarizing ourselves with Verus, 2) *version 2* (8 months two people, clean-slate rewrite) resulted in a simple but functioning kernel with a capability interface but no support for revocation, and 3) *version 3* (4 months, one person, 50% code reuse from previous version) developed support for revocation via recursive container trees, superpages, practical user-level specifications and support of isolation and noninterference proofs. The entire development of Horizon took around 2 person years, with roughly 14 months spent on developing verified code and 10 months spend developing non-verified parts of the system, e.g., user-level subsystems, build system, integration with verus, etc. Arguably, our ex-

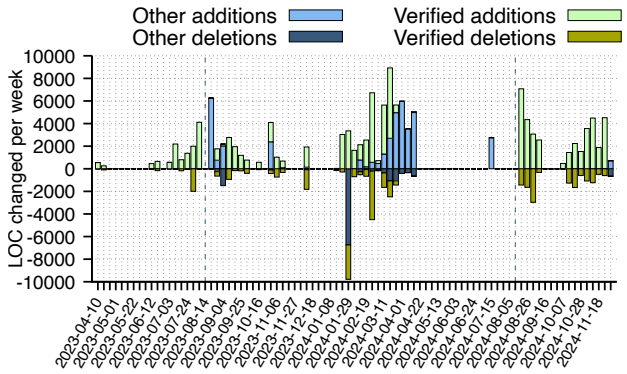

Figure 3: Horizon commit history (vertical lines separate versions)

plerience shows that verified kernel code can be developed nearly as fast as canonical non-verified systems, especially if the team is trained in verification and re-uses lessons from this work.

5.3 Microbenchmarks

We perform several microbenchmarks to evaluate whether the choice of Rust and Verus significantly affect performance of the microkernel. We compare our call/reply with the synchronous IPC mechanism implemented by the seL4 microkernel. A call/reply mechanism in Horizon takes around 1334 cycles, whereas seL4 takes 1339 cycles. We also measure the overhead of three system calls: mapping a page costs 283 cycles, whereas creating a process and a thread takes 6519 and 1862 cycles respectively (Table 3).

5.4 Device Drivers

Microkernel systems execute device drivers in isolated domains. A relatively expensive crossing of the isolation boundary, makes it impractical to implement systems in which the access to a device interface requires switching of a hardware isolation boundary. Instead, two practical options are: 1) implement user-level access to a subset of the device interface (similar to DPDK [9] and SPDK [27]), or 2) utilize asynchronous driver interfaces that are popular in virtualized environments. In Horizon we implement both the approaches. To understand the impact of microkernel construction on I/O intensive systems, we develop two device drivers: 1) an Intel 82599 10Gbps Ethernet driver (Ixgbe), and 2) an NVMe driver for PCIe-attached SSDs. Both device drivers are implemented as user-level Horizon processes.

5.4.1 Ixgbe Network Driver

We compare the performance of Horizon's Ixgbe driver with a highly-optimized driver from the DPDK user-space packet processing framework [9] on Linux. Similar to DPDK, we use polling mode to achieve peak performance. We configure Horizon to run several configurations: 1) `horizon-driver`: the benchmark application is statically linked with the driver (this configuration is very similar to user-level packet frameworks like DPDK). 2) `horizon-c2`: the benchmark application runs

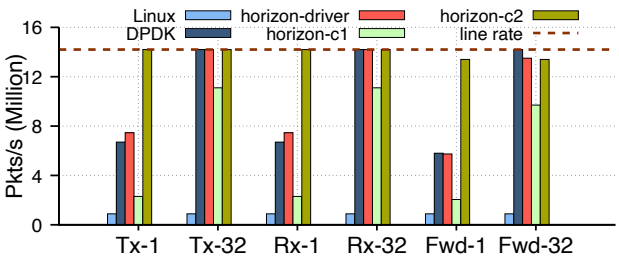

Figure 4: Ixgbe driver performance

in a separate process on a different core, and communicate with the driver driving on a separate core through a shared ring buffer. 3) `horizon-c1-b1` and `horizon-c1-b32`: the benchmark application runs as a separate process on the same CPU alongside the driver. The application uses a shared ring buffer with the driver and invokes the driver through an IPC endpoint which involves a context switch. The number after `b` represents the batch size of the request, i.e. the number of requests the application sends to the ring buffer before invoking the driver.

We send 64 byte UDP packets and measure the performance on two batch sizes: 1 and 32 packets (Figure 4). For packet receive tests, we use `Pktgen`, a packet generator that utilizes DPDK framework to generate packets at line-rate. Linux achieves 0.89 Mpps as it uses a synchronous interface and crosses the syscall boundary for every packet and goes through layers of abstraction in the kernel (Figure 4). 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 interprocess invocations, we run the benchmark application as a separate process (`horizon-c1-b1`) and (`horizon-c1-b32`). The overhead of domain crossings is apparent on a batch size of one, where Horizon can send and receive packets at the rate of 2.3 Mpps per-core with one context switching per packet (`horizon-c1-b1`). On a batch of 32 packets, the overhead of context switching is less pronounced as it gets amortized over a batch of requests and the application achieves 11.1 Mpps (`horizon-c1-b32`)

5.4.2 NVMe Driver

To understand the performance of Horizon'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 [27]. Similar to SPDK, the Horizon driver works in polling mode. Similar to Ixgbe, we evaluate NVMe driver under various configurations: 1) statically linked (`horizon-driver`); 2) application and driver run on different cores and communicate through shared ringbuffer (`horizon-c2`); 3) application runs on the same core with the driver and use endpoint to invoke the driver process after sending a batch of requests through the shared ring buffer (`horizon-c1-b1` and `horizon-c1-b32`).

We perform sequential read and write tests with a block size of 4KB on a batch size of 1 and 32 requests (Figure 5). On

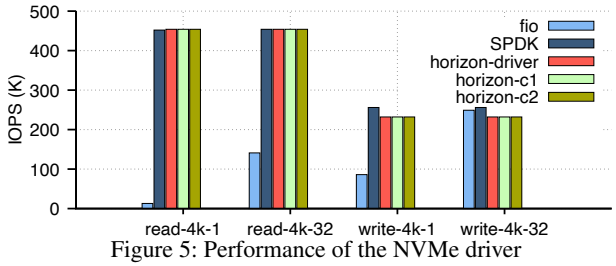

Figure 5: Performance of the NVMe driver

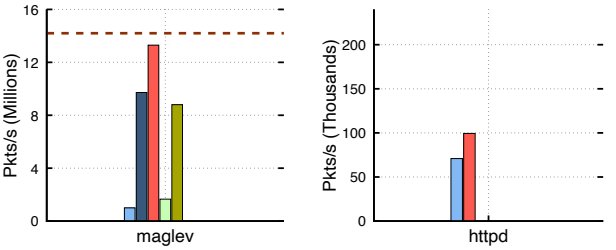

Figure 6: Performance of Maglev and Httpd

Linux, we use `fio`, a fast I/O generator; on SPDK and Horizon, 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 5). On a batch size of 1 and 32, the Horizon driver performs similar to SPDK, achieving maximum device read performance. On sequential write tests with a batch size of 32, Linux is within 3% of the device's maximum throughput of around 256K IOPS. Horizon driver incurs an overhead of 10% on all the configurations for writes (232K IOPS).

5.5 Application Benchmarks

To further evaluate the performance of Horizon under real-world scenarios, we develop three data-intensive applications on top of the device drivers:) the Maglev load balancer (Maglev) [16], 2) a memcached-compatible key-value store (`kv-store`), as well as 3) a tiny web server (`httpd`).

Since device drivers in Horizon are in userspace, we evaluate the applications in two scenarios: Where the application itself communicates with the device driver through shared memory, and where the device driver is embedded within the application as a single domain, similar to DPDK.

Maglev load-balancer Maglev is Google's load balancer with an algorithm that evenly distributes incoming traffic among backend servers [16]. It uses two hash tables to achieve even distribution of flows. A flow hash is generated for each new flow and used as the key to search the *look-up table*, whose size is proportional to the number of backend servers. The look-up table is statically populated and remains constant for

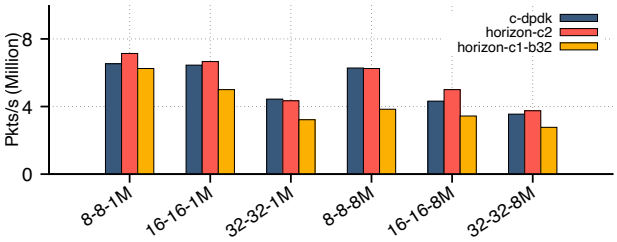

Figure 7: Key-value store

the same set of backend servers. To ensure existing flows are directed to the same servers when the set of backends change, the chosen backends are inserted into the *flow tracking table*.

Each packet therefore requires at least a lookup in the flow tracking table, and if it's a new flow, a lookup in the look-up table as well as an insertion into the flow tracking table.

We develop two additional implementations to compare Horizon against: A normal Linux program that uses the socket interface, and a DPDK-powered application that directly accesses the network card via PCIe passthrough [9]. With normal Linux socket interface, Maglev only achieves 1.0 Mpps per core because of system call overheads as well as the generic design of the Linux network stack (Figure 6). The DPDK-powered application has direct access to the NIC and achieves 9.72 Mpps per core. In Horizon, Maglev achieves 13.3 Mpps with the device driver running on a separate core with communication established over a shared ring buffer (`horizon-c2`). With Maglev invoking the device driver on the same core, the load balancer runs at 8.8 Mpps with a batch size of 32 (`horizon-c1-b32`), and 1.66 Mpps with a batch size of 1 (`horizon-c1-b1`).

Key-value store Key-value stores are crucial building blocks for modern datacenter systems ranging from social networks [41] to key-value databases [13]. To evaluate Horizon's ability to meet the performance requirement of datacenter applications, we develop a prototype of a network-attached key-value store, `kv-store`. Our implementation relies on an open addressing hash table with linear probing and the FNV hash function. In our experiments, we compare the performance of DPDK implementations: a C version developed for DPDK, and a Rust Horizon program that executes in a different process on a different core with the driver (`horizon-c2`) and on the same core with batch size of 32 (`horizon-c1-b32`). Similar to our Ixgbe and Nvme benchmarks, (`horizon-c1-b32`) uses `send()` `receive()` IPC mechanism for sharing the CPU and a shared ring buffer to transmit network packets between the application and device driver. We evaluate two hash table sizes: 1 M and 8 M entries with three sets of key and value pairs (`<8B,8B>`, `<16B,16B>`, `<32B,32B>`).

Web server The ability for a web server to serve an influx of users at high throughput and low latency is critical for a smooth user experience.

We develop a simple web server, `httpd`, that serves static

HTTP context. The web server continously polls for incoming requests from open connections in a round-robin manner, parses the request, and returns the static web page. We compare `httpd` against one of the de facto industry standard web servers, Nginx [40]. We configure the `wrk` HTTP load generator [47] to dispatch requests for 10 seconds using one thread and 20 concurrent open connections. Nginx on Linux achieve 70.9 K requests per second, whereas our implementation of `httpd` is able to serve 99.4 K requests per second when `httpd` is directly linked to the device driver.

6 Related Work

Operating system verification Early verification efforts were aimed at attaining the highest A1 assurance rating defined by the "Orange Book" [15] but remained largely unsuccessful due to limitations of existing verification tools [21, 28, 18, 45]. SeL4 became the first system to demonstrate a way to achieve verification of a practical microkernel [29]. Verification of seL4 involved 180,000 lines of proof code of the Isabelle/HOL theorem prover for 8,700 lines of C and required 22 person-years [29].

Hyperkernel demonstrated high degree of automation through the use of SMT solvers but at the cost of severe limitations in kernel functionality [39]. To support automated verification, Hyperkernel required all paths in the kernel to be *finite*, e.g., the system call interface forced the process to provide a file descriptor number for opening a file instead of choosing an available one in the kernel.

Ironclad [26] addressed complexity of the verification effort through a combination of Dafny [32], Boogie intermediate verification language [1], and Z3 SMT solver [14].

CertiKOS [23] and μC/OS-II [48] were aimed at verification of concurrent systems through the use of the Coq interactive theorem prover [8]. CertiKOS developed a concurrent OS kernel that supported fine-grained locking, interrupts and threads. Verification of CertiKOS and μC/OS-II took 2 and 5.5 person-years, respectively but required nearly the same proof-to-code ratio as seL4.

SeKVM utilized Coq to verify the core of the Linux KVM hypervisor [35]. Specifically, SeKVM decomposed the hypervisor into two separate layers and verified the privileged core using Coq. Even though SeKVM relied on ClightGen [33] for translating the C implementation to Coq it still required a large manual effort to address numerous unsupported C idioms. Recently, Spoq improved on ClightGen cutting the verification effort of SeKVM by 70% [36].

Verified NrOS [3] is the project to suggest the use of Verus for verification of operating systems and specifically for retrofitting verification into existing NrOS kernel incrementally [2]. NrOS outlines a range of possible design ideas and develops verified code for page-table management and its core concurrency mechanism, node replication, yet using Dafny [24]. Unfortunately, NrOS leaves a number of important design decisions about memory management, developing specifications and proofs for numerous semantically complex kernel data structures, exposing kernel state for user verification, etc., unanswered.

VeriSMo uses Verus to implement a verified security module for confidential VMs on the AMD SEV-SNP technology [50]. Verismo proves functional correctness of the security module as well as correctness of the information flow (Verismo relies on procedural macros to rewrite Rust code replacing regular types with wrappers that can track information flow). Despite developing a large codebase, Verismo is semantically simple. As a result it does not hit scalability scalability problems related to management of multiple semantically complex, inter-dependent data structures (e.g., processes and threads lists, page-tables, container trees, datastructures with complex lifetimes, etc.). Our work provides an example of how to address semantic complexity of a typical kernel.

Noninterference Noninterference is first proposed in 1982 by Goguen and Meseguer [20]. It is formally defined as a set of specifications given security policy in systems. The purpose of noninterference is to impose restrictions on information flows between subsystems and prevent low-security level user to infer information about high-level user in system context.

Over the years, the concept of noninterference has been reshaped and applied in different systems. Luke et. al. [38] demonstrated that the formulation of noninterference and its variants such as *nonleakage* can be either parameterized by program traces (a sequence of actions), or a set of unwinding conditions (desired properties of all actions). For example, works such as NiStar[46] and PROSPER[11] prove traditional noninterference by elimination of control flow between processes or enforcing decentralized information control flow via programming languages. seL4[37] proves nonleakage (more flexible version of noninterference), and works such as mCertiKOS[10] and Komodo monitor [17] prove a subset of unwinding conditions.

7 Conclusions

Our experience shows that a collection of careful design choices combined with the level of automation provided by Verus enable practical development of formally verified kernel code. Pointer-centric, flat permission design allowed us to develop Horizon in a manner very similar to an unsafe, unverified kernel and surprisingly with a low development effort. We hope that our design and development experiences can help building the next generation of practical, formally-correct kernels.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, 2006.

[2] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. Nros: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), July 2021.

[3] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. Beyond Isolation: OS Verification as a Foundation for Correct Applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (HOTOS), Providence, RI, USA, 2023.

[4] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N Bairavasundaram, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC'09)*, 2009.

[5] cloudlab machine c220g2 info. https://www.emulab.net/portal/show-hardware.php?type=c220g2.

[6] cloudlab machine c220g5 info. https://www.emulab.net/portal/show-hardware.php?type=c220g5.

[7] cloudlab machine d430 info. https://www.emulab.net/portal/show-hardware.php?type=d430.

[8] The Coq proof assistant. https://coq.inria.fr/.

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

[10] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for c and assembly programs. *SIGPLAN Not.*, (6):648–664, June 2016.

[11] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. *Proceedings of the ACM Conference on Computer and Communications Security*, November 2013.

[12] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, 2008.

[13] 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)*, 2007.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science), 2008.

[15] *Department of defense trusted computer system evaluation criteria*. In *The 'Orange Book' Series*. London, 1985, pages 1–129.

[16] 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)*, March 2016.

[17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP), Shanghai, China, 2017.

[18] L. J. Fraim. Scomp: a solution to the multilevel security problem. *Computer*, (7):26–34, July 1983.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (OASIS), 2004.

[20] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, 1982.

[21] B. D. Gold, R. R. Linde, and P. F. Cudney. Kvm/370 in retrospect. In *1984 IEEE Symposium on Security and Privacy*, 1984.

[22] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (APSys), 2011.

[23] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), November 2016.

[24] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *17th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), July 2023.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (SOSP), Monterey, California, 2015.

[26] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-End security via automated Full-System verification. In *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

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

[28] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Transactions on Software Engineering*, (11):1147–1165, 1991.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles* (SOSP), Big Sky, Montana, USA, 2009.

[30] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (SOSP), Austin, TX, USA, 2024.

[31] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages*, (OOPSLA1):85:286–85:315, April 2023.

[32] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, 16th international conference, lpar-16, dakar, senegal edition, April 2010.

[33] Xavier Leroy. The compcert c verified compiler: documentation and user's manual, September 2014.

[34] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, (OOPSLA1), April 2022.

[35] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*, August 2021.

[36] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling Machine-Checkable systems verification in coq. In *17th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), July 2023.

[37] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. Sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[38] Luke Nelson, James Bornholt, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Noninterference specifications for secure systems. *SIGOPS Oper. Syst. Rev.*, (1):31–39, August 2020.

[39] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP), Shanghai, China, 2017.

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

[41] 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)*, April 2013.

[42] nros pagetable github. https://github.com/utaal/verified-nrkernel.

[43] nros pagetable map frame aux function url. https://github.com/utaal/verified-nrkernel/blob/main/page-table/impl_u/l2_impl.rs#L953.

[44] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:* (6), December 2014.

[45] W. R. Schockley, T. F. Tao, and M. F. Thompson. An overview of the gemsos class a1 technology and application experience. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[46] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, October 2018.

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

[48] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *Computer Aided Verification*, 2016.

[49] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Toronto, Ontario, Canada, 2010.

[50] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, July 2024.