Atmosphere: 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, Atmosphere.

Atmosphere is a full-featured microkernel with a capability interface and support for the strict isolation of mixed-criticality systems. Atmosphere is 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.

We develop all code in Rust and prove its functional correctness with Verus. Atmosphere 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 executable code required an effort of less than a 2.5 person-years (but only 1.5 years were spent on verification, another person-year was spent developing non-verified parts of the system). On average, our code has a proof-to-code ratio of 3.18:1 and verifies as fast as it compiles, which we argue is practical for the development of verified systems.

1 Introduction

Historically, formal verification of operating systems has remained a challenging undertaking. While several projects demonstrated the possibility of developing verified kernels [25, 19, 41, 22, 21], verification required strong verification expertise and significant human effort. For example, verification of the first formally verified microkernel, seL4, required 11 person-years (an additional 9 person-years were needed for the development of formal language frameworks, proof tools, and libraries [25]).

In this work, we argue that recent advances in programming languages and automated formal reasoning radically change the burden of verified development, arguably, enabling a practical verification of operating systems for the first time [27, 26]. Specifically, our work leverages Verus, a recent verifier for Rust that combines properties of linear types with automated verification based on satisfiability modulo theories (SMT) [27, 29, 26]. Like other SMT-based verifiers [1, 28, 38], Verus encodes Rust code into an SMT expression, which is then checked by a solver (Z3 in the case of Verus [12]). Verus, however, relies on the properties of the Rust language, and specifically, single-ownership, for efficient encoding of data structures allocated on the heap.

Unique combination of the linear type system and SMT-based verification allows us to re-think the practicality of developing the verified low-level systems code. We demonstrate the possibility of developing a fully-verified microkernel, Atmosphere, at the speed and effort that, arguably, approach commodity unverified development. Atmosphere 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. To support fine-grained access control, Atmosphere implements capability interface for inter-process communication mechanisms, hardware devices, and associated IOMMUs.

We develop all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification, with Verus. Despite the impressive automation offered by Verus, several design choices were critical for enabling verification of a semantically complex codebase of a kernel and for ensuring the speed and scalability of the proof.

Pointer centric design Modern kernels implement their logic as a collection of recursive data structures with frequent cyclic references, e.g., doubly-linked lists, trees, and endless references that encode semantic relationships between objects in the kernel, e.g., a process tree will maintain a recursive list of all children of a given process as well as a reverse reference to a parent for each child. Neither Rust nor Verus supports such complex pointer-centric data structures well, i.e., Rust requires unsafe extensions to support cyclic references.

In Atmosphere, we make an unusual design decision: we choose to abandon canonical Rust and instead use it in a manner similar to an unsafe language like C, i.e., we develop all kernel data structures using unsafe raw pointers, but rely on Verus to establish the correctness of all pointer operations via a proof. This allows us to design performance-efficient data structures with all optimizations typical for unsafe code, e.g., our container tree uses internal storage, we use reverse pointers to support efficient, constant-time deletion from linked lists, etc. Intuitively, such a design choice is prohibitive due to the large number of pointers and the complexity of the proofs required to establish correctness of pointer operations. Surprisingly, however, a combination of *linear permission pointers* (an abstraction offered by Verus for reasoning about pointer operations) and our new technique for structuring the proofs – *flat* permission design – allows us to avoid complexity via non-recursive encoding of proofs about

unbounded data structures and clean separation of structural (the tree is well-formed) and non-structural arguments.

Flat approach to unbounded data structures Recursive specifications required to reason about recursive data structures are inherently difficult for SMT solvers as inductive proofs are not natively supported by SMT solvers and SMT solvers can only unroll recursive proofs a finite number of times, making them unsuitable for unbounded verification. While Verus supports recursive specifications and proofs through inductive proof functions, the verification complexity remains high and sometimes becomes unmanageable for complex, feature-rich recursive data structures. To keep the 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, all nodes of a tree, etc., are stored at the topmost level of the subsystem with the global view of a specific subsystem in a flat map. Such flat approach allows us to reason about recursive data structures like page tables, container and process trees, etc., in a non-recursive way. The global view of the data structure gives us freedom of encoding properties of unbounded, recursive data structures, in a simple non-recursive manner. Moreover, a global, flat view allows us to separate reasoning about the structure (e.g., the tree has no cycles, reverse pointers are correct, depth is maintained correctly) of each data structure and non-structural effects of individual nodes.

Manual memory management Rust ensures memory safety and automatic memory management via its borrow checker and a collection of trusted pointer types (e.g., `Rc<T>`). Unfortunately, automatic memory management is not sufficient for the kernel. First, in Rust memory leaks are not considered unsafe and Rust's ownership model does not prevent them (e.g., complex dependencies between kernel data structures can result in a memory leak due to a cycle in reference counted smart pointers). Second, reasoning about non-interference as a property over sets of all possible executions relies on the state of all memory in the system (e.g., the non-interference theorem requires one domain not to exhaust the memory of the system and in turn, affect other domains). Several system calls depend on the result of the memory allocation. Without the explicit state of the allocator, their behaviors (on the spec level) become nondeterministic.

Atmosphere implements manual memory management. We do not rely on the Rust borrow checker for reasoning about object lifetimes and instead allocate and deallocate all kernel objects explicitly. We, however, 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). This design choice allows us to support complex cyclic data structures and objects with reference-counted object lifetimes without the trust in Rust pointer types.

Finally, Atmosphere is a separation kernel that supports strict isolation of mixed-criticality systems. Careful design of system call specifications allows us to demonstrate how one can develop proofs of non-interference and functional correctness for the user code. Specifically, we demonstrate how users can define controlled non-interference (i.e., isolated subsystems are non-interfering but are allowed to establish communication with a verified shared process).

Our experience shows that despite the fact that we use Rust in a very pointer-centric way the combination of permission pointer reasoning offered by Verus, careful design choices for constructing the proofs about recursive data structures, and the level of automation provided by Verus enable practical development of formally verified kernels.

In total, development of Atmosphere 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 a proof-to-code ratio of 3.18:1, which is lower than comparable prior approaches [25, 19]. Moreover, careful design choices allow Atmosphere to finish full verification in less than 20 seconds on a modern laptop. To put this number in perspective, it takes less time to finish verification than compilation, which enables a truly interactive development cycle with a verifier in contrast to traditional testing. Atmosphere is developed without relying on any unverified types from the Rust standard library, e.g., vectors, reference counting types, etc. We only trust Verus's native types and trusted functions for mutating tracked permissions (at the moment, Verus lacks support for mutable references for tracked permissions).

2 Background: Verus

Verus is a new formal verification tool for Rust that supports semi-automated reasoning by using an SMT solver [27]. Verus allows one to write executable code, specifications, and proofs. Executable code is written in Rust, while specifications and proofs are written in a functional extension of Rust, which includes logical quantifiers like `forall` and `exists`. Verus modifies the Rust compiler to elide specifications and proofs (*ghost code*) during compilation time and thus they do not incur any runtime overhead.

In Verus, a typical flow of building and verifying a system involves defining the abstract state of the system and the expected behavior of the system interface. One then proves that implementation of the system is a refinement

```
1  pub struct PageTable{
2     pub cr3: usize
3     pub map: Ghost<Map<VAddr,MapEntry>>,
4     ... }
5  pub open spec fn syscall_mmap_spec(Ψ':Kernel, Ψ:Kernel, t_id
       : ThrdPtr, va_range: VaRange4K, perm_bits:MapEntryPerm,
        ret: SyscallReturnStruct) -> bool{
6    ...
7    // the state of each thread is unchanged
8    &&& Ψ'.thread_dom() =~= Ψ.thread_dom()
9    &&& forall|t_ptr:ThrdPtr|
10      Ψ'.thread_dom().contains(t_ptr) ==>
11      Ψ.get_thread(t_ptr) =~= Ψ'.get_thread(t_ptr)
12   ... // rest of the objects in the kernel
13   // virtual addresses outside of va range are not changed
14   &&& forall|va:VAddr| va_range@.contains(va) == false
15     ==> Ψ.get_address_space(proc_ptr).dom().contains(va)
16        == Ψ'.get_address_space(proc_ptr).dom().contains(va)
               && Ψ.get_address_space(proc_ptr)[va]
17                =~= Ψ'.get_address_space(proc_ptr)[va]
18   // newly allocated pages were free pages
19   &&& forall|page_ptr:PagePtr|
20      mmapped_physical_pages_seq.contains(page_ptr)
21      ==> Ψ'.page_is_free(page_ptr)
22   // each virtual address in va range gets a unique page
23   &&& forall|i:usize| 0<=i<va_range.len
24      ==> Ψ.get_address_space(proc_ptr)[va_range@[i]].addr
25       == mmapped_physical_pages_seq[i]
26   ...}
27 pub fn mmap(Ψ: &mut Kernal, t_ptr: ThrdPtr, va_range:
       VaRange4K) -> (ret: SyscallReturnStruct)
28   requires
29      old(Ψ).total_wf(), //global invariants
30      ...
31   ensures
32      syscall_mmap_spec(old(Ψ), Ψ, t_ptr, va_range, ret),
33 { ...
34   let tracked thrd_perm = Ψ.process_manager
35     .thrd_perms.tracked_borrow(t_ptr);
36   assert(thrd_perm@.addr()== t_ptr && thrd_perm.is_init());
37   let thread: &Thread = PPtr::<Thread>::from_usize(t_ptr)
38     .borrow(Tracked(thrd_perm));
39   let proc_ptr = thread.owning_proc;
40   ... }
```

Listing 1. Abstract and concrete state, specification and implementation functions.

of its high-level (abstract) specification. Verus allows one to define abstract state side by side with concrete executable code by using *ghost* variables. Ghost variables can be used to develop an abstract model of the system (e.g., as a state machine on ghost state) and to support the proofs. For example, the abstract state of the page table in Atmosphere is a map from virtual addresses to physical addresses with access permissions (Listing 1).

To prove refinement, one needs to establish equivalence between the abstract and concrete state of the system, e.g., in the case of the page table, the abstract map and the virtual to physical translation seen by the hardware memory management unit. That is, for each entry in the abstract map, if the MMU does a page table walk, the resolved physical address and access permission are equal to the value in the map.

Specification functions allow one to express the expected behavior of the system as a collection of pre- and post-conditions that reflect changes to the abstract state. For example, `syscall_mmap_spec()` defines the expected behavior for system call `mmap()`, which lets the process to allocate and map a consecutive region of memory at the virtual address `va_range` (Listing 1). At a high level, our specification captures that threads, processes, and other kernel objects is not changed, virtual addresses outside of the newly mapped range are not changed, newly allocated pages were free before, each page is mapped uniquely, etc.

The concrete implementation of the `mmap()` system call operates on concrete, non-ghost variables. The proof code provides hints to the verifier for how to complete the proof and helps to mutate ghost and tracked variables in executable functions. The system call requires that the kernel is well-formed before invocation (i.e., `old(kernel).total_wf()`), and ensures that the kernel is well-formed after the invocation as well as that the kernel state is updated according to the system call specification (i.e., ensures `syscall_mmap_spec()`).

Linear ghost (tracked) permissions Verus supports reasoning about raw pointers. Typically, operations on pointers, e.g., dereferencing a pointer, is an unsafe operation in Rust that bypasses the borrow checker and breaks the rules of the linear type system, which makes reasoning about the pointers problematic. Verus, however, offers an abstraction of *permissioned pointers*, i.e., `PPtr<T>`, and a linear tracked permission type `PointsTo<T>` that allow construction of proofs about raw pointers [27]. For example, to access the parent process from a raw thread pointer (Listing 1 line 39), one gets an immutable reference to the thread's permission (line 35) and proves that the address of the tracked permission matches the pointer (line 36).

3 System Architecture

Atmosphere is a full-featured microkernel with an object capability interface. The microkernel supports a minimal set of mechanisms to implement address spaces, threads of execution, memory management, interrupt dispatch, interprocess communication, IOMMUs, demand paging, and containers – groups of user processes with guaranteed memory and CPU reservations that are used to enforce isolation between mixed-criticallity systems. In a Atmosphere system device drivers, network stacks, file systems, etc. run as user-space processes. Atmosphere supports multiprocessor machines, 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.

The system call interface of the microkernel allows processes to communicate via endpoints. A sender thread can pass scalar data, references to memory pages, IOMMU identifiers, and references to other endpoints. Endpoints allow processes to establish regions of shared memory, which then

can be used for efficient asynchronous communication [17, 10, 4].

Isolation and non-interference To support provable separaton between groups of processes, Atmosphere implements an abstraction of a container. A container is a group of processes that have a guaranteed reservation of memory (*container quota*) and CPU cores, i.e., when container is created the parent container passes a subset of its own reservation of memory and cores to a child. The microkernel implements careful accounting of all memory allocations from any process in the container, i.e., memory allocated for the user process or allocated by the kernel to keep metadata about threads, endpoints, etc.

Access control and revocation In Atmosphere, containers form a tree to maintain parent-child relationships. Containers higher in the hierarchy have the capability to terminate their direct and indirect children and harvest their resources. Inside each container, the constituent processes form a separate process tree which enables parent-child tracking between processes in the same container. Similarly, the parent process has the capability to terminate child processes. Note that container and process hierarchies are *unbounded* – processes and nested containers can be created for as long as memory is available.

We make a design choice to prohibit forcible, fine-grained revocation of resources. Resources return to the parent container when the child container is terminated, but of course only if they were not passed outside of the container via an endpoint. This design choice simplifies verification of user code – we have a guarantee that memory cannot be revoked while the verified code is executed.

The ability to exchange resources via endpoints requires a careful construction of communication channels between containers to control leakage and propagation of resources. In our running example, which we discuss in Section 4.3, we demonstrate how such controlled communication can be established and verified. Specifically, we show how two mistrusting containers A and B establish communication with the verified container V (Figure 1). Both A and B share its memory with V, however, since we establish functional correctness of V we prove that it 1) never leaks memory between untrusted containers and 2) releases all memory used for communication with unverified containers in case they crash.

To support demand paging, a process can allow its direct and indirect parents to manage its address space.

4 Verification

To specify the expected behavior of the system, we model the system as a state machine operating on the abstract state. We rely on a collection of high-level specifications that describe how the abstract state of the kernel is updated on each kernel invocation and a hierarchy of invariants that describe correctness of each kernel data structure, as well as express cross-cutting properties like safety and isolation. We then prove the following theorems: *refinement* and *well-formedness*.

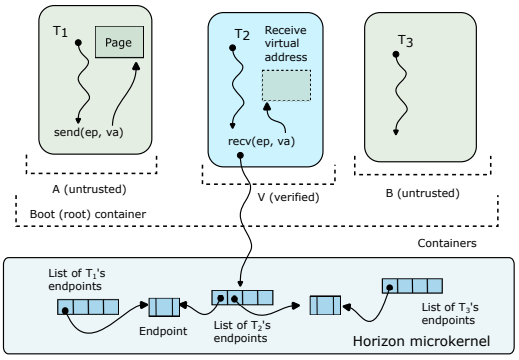

Figure 1. Architecture of Atmosphere. Containers A and B are untrusted but can establish communication with a verified container V. Thread T_1 invokes the `send()` system call to pass a page to T_2 which is already waiting on the endpoint inside the microkernel.

The refinement theorem proves that the implementation of each system call is functionally correct, i.e, it is a refinement of its high-level abstract specification. That is, when a microkernel call is executed, its effect on the system is equivalent to the high-level specification. The well-formedness theorem proves that high-level correctness invariants hold after each state transition.

We use a collection of invariants to express a broad range of semantic properties of each object in the kernel, relationships between them, safety and leak freedom. Specifically, we prove the following properties:

- semantic correctness of data structures (e.g., trees and lists are acyclic, etc.),
- safety and leak freedom,
- and in a user-level non-interference proof, resource isolation between containers.

4.1 Flat, Pointer-Centric Design

In Atmosphere we make a design choice to avoid any limitations on the use of unsafe pointers, i.e., develop recursive pointer data structures like linked lists, container and process trees, and in general, freely use pointers to implement complex logical dependencies between kernel data structures. Free use of pointers provides the freedom of unsafe development, e.g., use coding paradigms and performance optimizations typical for unsafe, unverified code, e.g., reverse pointers from children to parents for efficient lookup, internal list and tree storage, etc. Not surprisingly, establishing correctness of pointer operations is challenging. Reasoning about recursive data structures is inherently difficult for SMT solvers as they provide only limited support for inductive proofs and can only unroll recursive proofs a finite number of times, making them unsuitable for unbounded verification.

```
pub struct ProcessManager{
  pub root_container: CtnrPtr,
  pub cntr_perms: Tracked<Map<CtnrPtr, PointsTo<Container>>>,
  pub proc_perms: Tracked<Map<ProcPtr, PointsTo<Process>>>,
  pub thrd_perms: Tracked<Map<ThrdPtr, PointsTo<Thread>>>,
  pub edpt_perms: Tracked<Map<EdptPtr, PointsTo<Endpoint>>>,
  ...}
pub struct Container{
  parent: Option<CtnrPtr>, // root has no parent
  children: StaticList<CtnrPtr>,
  depth:usize,
  path :Ghost<Seq<CtnrPtr>>, //direct and indirect parents
  subtree :Ghost<Set<CtnrPtr>>, //all reachable children
  ...}
```

Listing 2. Flat permissions

Flat ownership We make a key architectural choice: "flat" organization of pointer permissions for complex recursive data structures. Specifically, in all recursive pointer data structures we use raw pointers like one would do in an unsafe C. Yet permissions to access these pointers are stored as a single flat map at the topmost level of the subsystem which provides the global view of the entire subsystem. Flat ownership provides us with a global view of the data structure and specifically allows us to 1) convert recursive specifications into non-recursive ones; 2) decouple the proofs about the structure of the data structure (e.g., a tree is well-formed) from other non-structural proofs (e.g., object lifetimes); and 3) enable all direction traversal (e.g., traversing up and down in a tree requires the node to access both the parent and children).

Global invariants For example, the process manager holds permissions to all threads, processes, containers, endpoints, etc., as a collection of flat maps. Availability of the global system state via the permission pointer map allows us to formulate global invariants without navigating object hierarchies, i.e., peering into the threads of a process within a container. For example, we can directly access the state of all the threads in the system and establish invariants about them:

```
// all threads in the system are well formed
  pub spec threads_wf(pm: &ProcessManager) -> bool{
    forall|t_ptr:ThrdPtr|
      pm.thrd_perms@.dom().contains(t_ptr)
    ==>
    pm.thrd_perms@[t_ptr].value().wf()
  }
```

Non-recursive invariants To illustrate how we use flat ownership to avoid inductive proofs about unbonded data structures, consider an example of a container tree – an unbounded data structure that can grow for as long as the system has memory. In Atmosphere, containers form a single recursive tree (internally, each container has a separate tree of processes). We implement the tree in a manner similar to unsafe C: each node has a list of children and a pointer to its parent (Listing 2). However, all permissions to access the nodes of the container tree are stored at the level of the process manager that holds the non-ghost root of the container tree.

To support various aspects of the proof, we expose the parent-child relationships between nodes as a ghost sequence of nodes that describes the path from the root to the current node `path`. The path along with the set of reachable 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 are disjoint with the address spaces in the subtree of another).

Without the flat storage, maintaining the `path` invariant requires recursive definitions about the parents and its direct children – only local information about immediate children is available in a non-flat, hierarchical case. For example, one first encodes that the child's `path` equals to the parent's `path` plus the parent:

```
pub spec fn child_resolve_path_wf(&self:Container) -> bool{
for|child:Container|
  self.children@.contains(child) ==>
    self.get_child(child).path =~= self.path.push(self)}
```

This recursive definition can be used to reconstruct the full path from the root to any nodes in the tree. Verus supports either automatic unrolling of the definition or constructing a manual inductive proof. However, SMT solvers can only unroll a recursive proof a finite number of times (bounded recursion depth) and lack native support for inductive reasoning. Although the unbounded recursions can be proven in Verus with the help of inductive proof functions, the verification difficulty and manual effort are high.

With the global view of all children available in the flat case, we instead can describe the `path` invariant in a non-recursive way:

```
pub spec fn resolve_path_wf(&self:ProcessManager) -> bool {
  for|c:container, i:int|
    0 <= i < self.cntr_perms@[c].value().path.len() ==>
    self.cntr_perms@[c].value().path.subrange(0, i)
      =~= self.cntr_perms@[self.cntr_perms@[c].value().path[i
           ]].value().path
}
```

I.e., for any node `i` on the path of container `c` its subpath from the root (node 0) to node `i` is equal to the path of `i`th node. Instead of relying on recursive reasoning about pairs of parents and children, we directly express the correctness of the `path` invariant for all nodes of the tree. Global permission map `cntr_perms` allows us to reason about any node of the tree.

Modularity and isolation of structural and non-structural proofs Pointer-centric design of the kernel data structures requires us to maintain structural invariants that control correctness of each data structure, e.g., the tree has no cycles besides child to parent references. Maintaining such invariants is challenging as they have to be ensured on any update even if it is not structural, e.g., `new_container()` adds a new child to the container but otherwise does not change

```
1  pub fn new_container(Φ :&mut PM, C_P:CntrPtr, C_C:CntrPtr
       ,...)
2  requires
3     container_tree_wf(Φ'), ...
4  ensures
5     container_tree_wf(Φ), ...
6  { ...
7    // at the end of the function
8    assert(container_tree_wf(Φ)) by {
9      assert(new_container_ensures(Φ', Φ, C_P, C_C));
10     new_container_preserve_tree_wf(Φ', Φ, C_P, C_C);
11 }}
12 pub open spec fn new_container_ensures(Φ':&PM, Φ:&PM, C_P:
       CntrPtr, C_C:CntrPtr) -> bool{
13 &&&
14 ...
15 // all parents will contain new container in their subtrees
16 forall|c_ptr:CntrPtr|
17   Φ'.cntr_perms@.dom().contains(c_ptr) &&
18     Φ'.cntr_perms@[C_P].value().resolve_path.contains(c_ptr))
           ==>
19     Φ'.cntr_perms@[c_ptr].value().subtree@.insert(C_C) =~=
20       Φ.cntr_perms@[c_ptr].value().subtree@
21 ...
22 // other containers are not changed
23 }
24 pub proof fn new_container_preserve_tree_wf(Φ':&PM, Φ' :&PM,
       C_P:CntrPtr, C_C:CntrPtr,)
25 require
26    container_tree_wf(Φ'),
27    new_container_ensures(Φ', Φ, C_P, C_C),
28 ensures
29    container_tree_wf(Φ),
30 {...}
```

Listing 3. Isolation of structural and non-structural proofs

the structure of the tree. Flat ownership along with `closed` specification functions allow us to isolate the proofs about structural invariants from non-structural arguments (e.g., each process has at least one thread).

For example, executable function `new_container()` creates a child container C_C for the given parent container C_P and transition the state of the ProcessManager from Φ' (old state) to Φ (new state). We use a closed spec function `container_tree_wf()` to define all the structural invariants of the container tree in a separate container tree module. While the open spec function `new_container_ensures()` defines how each container's state changes in `new_container()`.

Note that since `new_container_ensures()` does not check for any invariants, it has very little verification overhead. In the preconditions, `new_container()` requires the container tree to be well-formed before the execution and proves to satisfy `new_container_ensures()` (line 9) at the end of the function. The proof function `new_container_preserve_tree_wf()` in the container tree module proves that if the transition from the old state to the new state satisfies the specification `new_container_ensures()`, the new state also satisfies `container_tree_wf()`. Therefore, function `new_container()` can prove `container_tree_wf()` (line 10) without being exposed to the complex structural invariants of the container tree and significantly reduces the size of the SMT search space.

Note that such proof isolation would not be possible with traditional hierarchical ownership, as the executable function would not be able to modify the state of the new container's indirect parents without knowledge of the hierarchy (needs to traverse up to modify their subtrees).

4.2 Manual memory management

The execution environment of a microkernel, however, is inherently nonlinear – kernel objects have a combination of manual and reference-counted lifetimes, and the kernel relies on pointer data structures for optimal performance. We 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.

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).

Memory allocation In Atmosphere dynamic memory allocation for kernel objects, e.g., containers, processes, threads, endpoints, is done at the granularity of 4KB, 2MB, or 1GB 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. 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.

Safety 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.

A straightforward approach is to prove that all the objects in the kernel are pair-wise disjoint and then prove that 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 establishing explicit invariants for *all* objects (i.e., sizes and addresses) in the kernel, making the SMT search space grow exponentially. Instead, we leverage the fact that objects of individual types are used in the kernel in a hierarchical manner, i.e., page tables are used in the memory subsystem; threads, processes, and containers are used by the process management subsystem.

In Atmosphere, 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 it. Here, the ownership means either a direct ownership of Rust (e.g., an array of objects) or ownership via a tracked permission. For example, a page table does not own any other objects, besides the physical pages used to form the page table.

Bottom-up recursive memory reasoning To form the system's memory usage, we hierarchically maintain the `page_closure()` for each kernel object, prove pair-wise disjointness locally, and merge the `page_closure()` of each subsystem. For example, the virtual memory management subsystem owns all page tables and IOMMU tables in the kernel. It maintains a set of invariants to ensure that each page table and IOMMU table's `page_closure()` are pair-wise disjoint, and their union is equal to the `page_closure()` of the virtual memory management subsystem. With this, we can easily infer the desired memory properties (i.e., all the objects in the kernel are pair-wise disjoint) without explicitly proving them as global invariants. Although Rust's linear ownership guarantees each objects are pair-wise disjoint, to maintain the page closures, we still need to explicitly ensure pair-wise disjointness, otherwise, we cannot correctly maintain `page_closure()`.

Explicit memory allocator state Establishing memory leak freedom and hyperproperties related to system memory requires the state of the memory allocator to be visible. Explicit invocations of the allocator allow exact reasoning about

```
1  pub fn alloc_page_4k(alloc:&mut Allocator)
2   ensures
3     alloc.free_pages_4k() =~=
4       old(alloc).free_pages_4k().remove(ret.0),
5     alloc.allocated_pages_4k() =~=
6       old(alloc).allocated_pages_4k().insert(ret.0),
7     ...
8
9  pub fn new_endpoint(pm:&mut ProccessManager)
10  ensures
11    pm.page_closure() =~=
12      pm(self).page_closure().insert(page_ptr),
13
14 // allocate new endpoint
15 let (p_ptr,mut p_perm) = self.page_alloc.alloc_page_4k();
16 self.proc_man.new_endpoint(p_ptr, p_perm, ...);
```

Listing 4. Establishing safety of allocation

the allocator's state through the invocations' pre- and post-conditions. We expose the internal state of the allocator as sets of free, allocated, merged and mapped pages. 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.

For example, allocation of a new endpoint uses the post-condition of the `alloc_page_4k()` and `new_endpoint()` functions (lines 15-16, Listing 4). First, we establish that a newly allocated page was previously not allocated via the postconditions of `alloc_page_4k` This ensures that using this page to form a new endpoint object in the kernel is safe. Furthermore, the postconditions of the two functions ensure that the set of allocated pages grows by exactly one page and process manager's `page_closure()` grows exactly by this one page as well. Since other subsystems do not change, we can prove that the memory safety and leak freedom invariants still hold.

4.3 Noninterference

Atmosphere is a separation kernel designed to support the development of the system configurations with provable isolation and noninterference between containers. To illustrate the possibility of using Atmosphere 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 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 Atmosphere 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 that runs one thread of execution (naturally, this simplifies verification, but is also acceptable for containers that implement system security policy, e.g., mediate the 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.

State machine model We model the system as a state machine:

- Ψ: is the kernel as a state machine.
- C_A, C_B, C_V are all the containers in the subtrees of A, B, V respectively (including A, B, V).
- P_A, P_B, P_V are all the processes in C_A, C_B, C_V respectively.
- T_A, T_B, T_V are all the threads in C_A, C_B, C_V respectively.
- $Step(\Psi, \Psi', t, a)$ represents a system call from thread t with system call argument a that transitions kernel state from Ψ to Ψ'

Constructing state machine model with flat permission storage The direct access to the objects' state provided by flat storage gives flexibility to the system abstract state. For example, to construct T_A with recursive ownership, one would need to rely on two unbounded recursive specs to first *fold* the container subtree to form C_A, second *fold* the process tree of each container to form P_A, and finally *union* the threads of P_A to form T_A. With flat storage, we can simply construct T_A with the following bidirectional invariants to ensure T_A contains and only contains all the threads in C_A:

```
spec fn T_A_wf(Ψ:Kernel,A:CntrPtr,T_A:Set<ThrdPtr>) -> bool{
  &&& forall|c_ptr: CntrPtr, t_ptr:ThrdPtr|
    (c_ptr == A || Ψ.get_cntr(A).subtree@.contains(c_ptr))
    && Ψ.get_cntr(c_ptr).owned_thrds@.contains(t_ptr)
    ==> T_A.contains(t_ptr)
  &&& forall|t_ptr:ThrdPtr| T_A.contains(t_ptr) ==>
    (Ψ.get_thrd(t_ptr).owning_cntr == A || Ψ.get_cntr(A)
      .subtree@.contains(Ψ.get_thrd(t_ptr).owning_cntr))}
```

Memory and communication channel isolation We prove memory isolation between P_A and P_B.

```
pub open spec fn memory_iso(Ψ: Kernel, P_A:Set<ProcPtr>, P_B:
     Set<ProcPtr>) -> bool{
  forall|a_p_ptr: ProcPtr, a_va:VAddr,
    b_p_ptr: ProcPtr, b_va:VAddr|
    P_A.contains(a_p_ptr) && P_B.contains(b_p_ptr) &&
    Ψ.get_address_space(a_p_ptr).dom().contains(a_va) &&
    Ψ.get_address_space(b_p_ptr).dom().contains(b_va)
    ==>
    Ψ.get_address_space(a_p_ptr)[a_va].addr !=
      Ψ.get_address_space(b_p_ptr)[b_va].addr}
```

That is, the pages mapped by the address spaces of P_A are proven not to be mapped by the address spaces of P_B. Additionally, to allow fast communication, shared endpoints can be established between T_V and T_A or T_V and T_B, but not between T_A and T_B.

```
pub open spec fn endpoint_iso(Ψ: Kernel, T_A:Set<ThrdPtr>, T_B
     :Set<ThrdPtr>) -> bool{
  forall|a_t_ptr:ThrdPtr, a_e_idx: EdptIdx,
    b_t_ptr:ThrdPtr, b_e_idx: EdptIdx|
    T_A.contains(a_t_ptr) && T_B.contains(b_t_ptr)
    ==>
    Ψ.get_thrd_edpt_descriptors(a_t_ptr)[a_e_idx] !=
      Ψ.get_thrd_edpt_descriptors(b_t_ptr)[b_e_idx]}
```

To prove A and B cannot break memory and communication channel isolation, we show that after a random system call with random system call arguments from any thread in T_A and T_B, the memory isolation and endpoint isolation are still satisfied:

```
forall|Ψ: Kernel, Ψ': Kernel, t:ThrdPtr, a:SysCallArg|
  Step(Ψ, Ψ', t, a) && (T_A.contains(t) || T_B.contains(t)) &&
    memory_iso(Ψ, P_A, P_B) && endpoint_iso(Ψ, T_A, T_B)
  ==> memory_iso(Ψ', P_A, P_B) && endpoint_iso(Ψ', T_A, T_B)
```

Noninterference Similar to prior systems [15, 19], 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 Atmosphere, the return value of system calls is deterministic and only depends on the state of the old kernel state and arguments.Thus, OC is trivial to prove.

Weak step consistency (WSC): In our model, we show that the system call return value of each arbitrary system call from container B remains unaffected before and after an arbitrary system call from container A.

Local respect (LR): We establish local respect via the post-conditions of the system calls. An arbitrary system call executed by one container cannot affect the observable state of another container.

Limitations In the non-interference proof, we only reason about the effects of system calls but not the user code that despite being isolated on different CPUs can introduce timing channels in shared caches and other hardware resources. Also, at the moment, Atmosphere 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.

5 Evaluation

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

Name	Language	Spec Lang.	Proof-to-Code Ratio
seL4	C+Asm	Isabelle/HOL	20:1 [25]
CertiKOS	C+Asm	Coq	14.9:1 [19]
SeKVM	C+Asm	Coq	6.9:1 [30]
Ironclad	Dafny	Dafny	4.8:1 [22]
NrOS	Rust	Verus	10:1 [3]
VeriSMo	Rust	Verus	2:1 [42]
Atmosphere	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
Atmosphere Pagetable	33s	–	2168	496	4.37
Mimalloc	8m 12s	1m 40s	13703	3178	4.3
VeriSMo	61m 24s	12m 11s	16101	7915	2.0
Atmosphere	3m 29s	1m 7s	15701	4937	3.18

Table 2. verification time of different systems on CloudLab c220g5

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

We conduct evaluation on the publicly available CloudLab [36] 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 the verification effort of Atmosphere in perspective of prior work, we collect the proof-to-code ratio across several recent kernel projects (Table 1. Atmosphere has a proof-to-code ratio of 3.18:1 which is a significant improvement compared to the existing formally verified microkernels SeL4 [25] and CertiKOS [19], which have proof-to-code ratio of 20:1 and 14.9:1, respectively (Table 1). Interestingly, 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 Atmosphere, i.e., it does not support leak freedom, only support page table identity mapping, etc.

Verification time Rapid verification is essential for interactive development. On the c220g5 server, Atmosphere finishes full verification in about 1 minute 10 seconds with 8 threads (Figure 2). On a newer laptop with a recent CPU, Intel i9-13900hx, Atmosphere finishes full verification in just 15 seconds (on 32 threads), and 47 seconds (on 1 thread).

Operation	Atmosphere	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

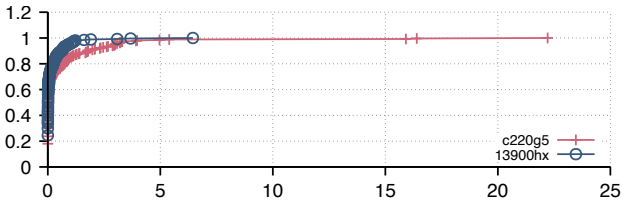

Figure 2. Verification time for each function

5.2 Impact of flat design

To isolate the impact of our flat approach, we compare the proof-to-code ratio and verification times for the page table subsystems in NrOS [34] and Atmosphere – both systems use Verus to implement a comparable page table logic with similar verification goals, and a similar size of executable code.

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. The tracked permissions of each PML level are stored at the topmost level of each page table. NrOS's page table, however, follows Rust recursive ownership.

```
spec fn mapping_4k(&PT:PageTable) -> Map<VAddr,MapEntry>
spec fn mapping_2m(&PT:PageTable) -> Map<VAddr,MapEntry>
spec fn mapping_1g(&PT:PageTable) -> Map<VAddr,MapEntry>
pub struct PageTable{
  l4_table: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
  l3_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
  l2_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,
  l1_tables: Tracked<Map<PagePtr,PointsTo<PageMap>>>,}
```

Refinement proof The page table uses a range of invariants to prove its structural invariants (e.g, each entry in any PML level only maps to the next PML level). Most importantly, it proves refinement between its concrete state and its abstract mapping. For example, in mappings of 4KB 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
  ==>PT.mapping_4k().contains(index2va((l4i,l3i,l2i,l1i)))
    == PT.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
    && PT.resolve_mapping_4k_l1(l4i,l3i,l2i,l1i).is_Some()
  ==> PT.mapping_4k()[index2va((l4i,l3i,l2i,l1i))]
    =~= PT.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

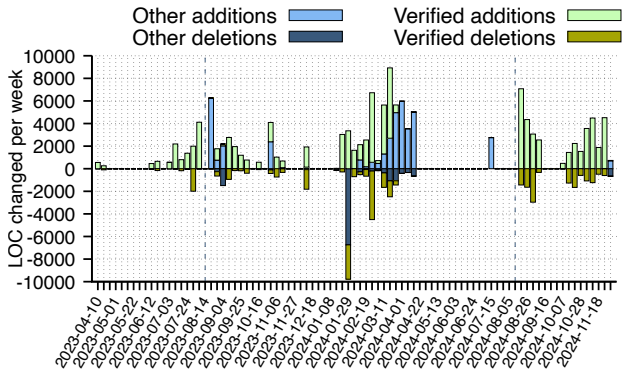

Figure 3. Horizon commit history (vertical lines separate versions)

domain of the abstract mapping. The second for all statement ensures the equivalency of the mappings.

One of the most complicated page table proofs 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. In NrOS, such a proof requires manual unrolling of the recursive specs through different PML levels and requires around 200 lines of proof code in `map_frame_aux()` function [35]. Since we have direct access to the states of all PML level tables, no unrolling is needed. With around 30 lines (PML4 requires 7 lines) of proof code, we directly prove that all other entries in all PML levels do not change:

```
1  pub fn map_4k_page(PT: &mut PageTable, dst_l4i:L4I, dst_l3i:
       L3I, dst_l2i: L2I, dst_l1i: L2I, va:VAddr, ...)
2  { //executable code
3    // other virtual addresses at PML1,2,3 do not change
4    //other virtual addresses at PML4 do not change
5    assert( forall|l4i: L4I,l3i: L3I,l2i: L2I, l1i: L1I|
6      0<=l4i<512 && 0<=l3i<512 && 0<=l2i<512 && 0<=l1i<512
7      && ((dst_l4i, dst_l3i, dst_l2i, dst_l1i)
8        != (l4i,l3i,l2i,l2i))
9      ==> PT.resolve_mapping_l1(l4i,l3i,l2i,l1i) =~=
10       old(PT).resolve_mapping_l1(l4i,l3i,l2i,l1i));
11 }
```

Due to the flat design, compared to the page table from NrOS [3] which also uses Verus, Atmosphere page table has 3x lower proof to code ratio (13.3:1 and 4.4:1). Moreover, on a single thread, Atmosphere page table is over 3x faster than the NrOS page table.

5.3 Development speed and effort

We completed development of Atmosphere 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 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

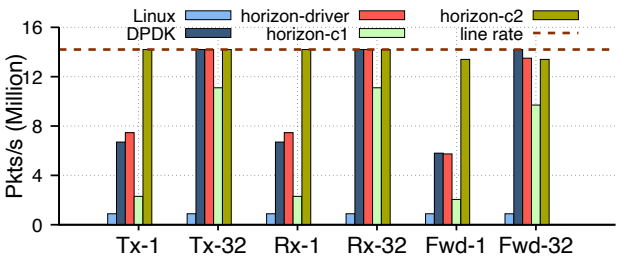

Figure 4. Ixgbe driver performance

specifications and support of isolation and noninterference proofs. The entire development of Atmosphere took around 2 person years, with roughly 14 months spent on developing verified code.

5.4 Microbenchmarks

We compare our call/reply with the synchronous IPC mechanism implemented by the seL4 microkernel. A call/reply mechanism in Atmosphere 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).

5.5 Device Drivers

In Atmosphere we implement device drivers in isolated domains with two configurations: 1) implement user-level access to a subset of the device interface (similar to DPDK [9] and SPDK [23]), and 2) utilize asynchronous driver interfaces that are popular in virtualized environments. 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 Atmosphere processes.

5.5.1 Ixgbe Network Driver We compare the performance of Atmosphere'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 Atmosphere 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 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`). Atmosphere 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 and the application achieves 11.1 Mpps (`horizon-c1-b32`)

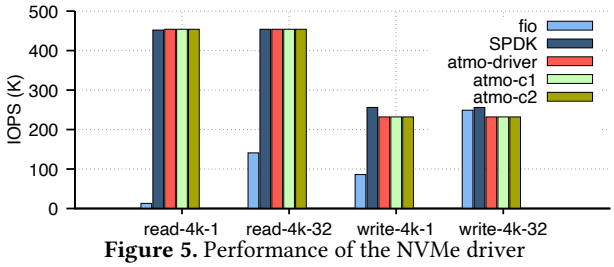

Figure 5. Performance of the NVMe driver

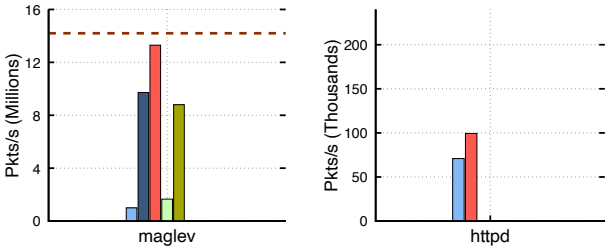

Figure 6. Performance of Maglev and Httpd

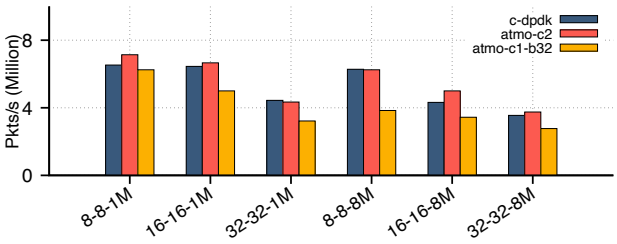

Figure 7. Key-value store

5.5.2 NVMe Driver To understand the performance of Atmosphere'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 [23]. Similar to SPDK, the Atmosphere 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 ring buffer (`horizon-c2`); 3) application runs on the same core with the driver and use the endpoint to invoke the driver process after sending a batch of requests (`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 Linux, we use `fio`, a fast I/O generator; on SPDK and Atmosphere, 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 Atmosphere 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. Atmosphere driver incurs an overhead of 10% on all the configurations for writes (232K IOPS).

5.6 Application Benchmarks

To further evaluate the performance of Atmosphere under real-world scenarios, we develop three data-intensive applications on top of the device drivers:) the Maglev load balancer (Maglev) [14], 2) a memcached-compatible key-value store (`kv-store`), as well as 3) a tiny web server (`httpd`).

Since device drivers in Atmosphere 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 [14]. 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 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 Atmosphere 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 Atmosphere, 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 [33] to key-value databases [11]. To evaluate Atmosphere'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 Atmosphere 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 continuously 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 [32]. We configure the `wrk` HTTP load generator [39] 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

Early verification efforts were aimed at attaining the highest A1 assurance rating defined by the "Orange Book" [13] but remained largely unsuccessful due to limitations of existing verification tools [18, 24, 16, 37]. SeL4 became the first system to demonstrate a way to achieve verification of a practical microkernel [25]. Verification of seL4 involved 180,000 lines of proof code of the Isabelle/HOL theorem prover for 8,700 lines of C and required 20 person-years (11 years for the proofs and 9 years for the development of supporting formal language frameworks, proof tools, and libraries [25]). Similar to seL4, Atmosphere uses pointer-centric design for ultimate performance, yet leverages Verus for high-degree of automation. CertiKOS [20] and μC/OS-II [40] were aimed at verification of concurrent systems with fine-grained locking through the use of the Coq interactive theorem prover [8]. 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. Although these systems demonstrate the level of complexity of an OS kernel that interactive theorem provers (ITP) can verify the verification effort remain high and the verification speed is far from ideal, hindering the development experience.

Hyperkernel demonstrated a high degree of automation through the use of SMT solvers but at the cost of severe limitations in kernel functionality [31]. 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. And despite the automation offer by Z3, verification time is roughly 30 minutes on quad-core i7-7700K according to its GitHub page.

Verified NrOS [3] is the project to suggest the use of Verus for verification of operating systems and specifically for retrofitting verification into the existing NrOS kernel incrementally [2]. At the moment, only the page table code was verified [34]. Unlike Atmosphere, NrOS follows the classical hierarchical approach to proof architecture. Flat ownership in Atmosphere significantly lowers complexity of the proofs by avoiding bounded unrolling and inductive proofs.

VeriSMo uses Verus to implement a verified security module for confidential VMs on AMD SEV-SNP [42]. VeriSMo proves functional correctness of the security module as well as correctness of the information flow. Despite the large codebase, the core of VeriSMo is semantically simple – simple state of the system can be described with mostly a safe subset of Rust and minimal pointer references. As a result, it does not hit scalability problems related to the verification of complex, recursive data structures. We, on the other hand, demonstrate how to handle the semantic complexity of a typical kernel.

7 Conclusions

Our experience shows that a collection of careful design choices combined with the level of automation provided by Verus enable the practical development of formally verified kernel code. Pointer-centric, flat permission design allowed

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

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

[37] 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.

[38] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), St. Petersburg, FL, USA, 2016.

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

[40] 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.

[41] 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.

[42] 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.