Atmosphere: Towards Practical Verified Kernels in Rust

Xiangdong Chen[*]
University of Utah

Zhaofeng Li[*]
University of Utah

Lukas Mesicek
University of Utah

Vikram Narayanan
University of Utah

Anton Burtsev
University of Utah

Abstract

Historically, development of formally-verified operating systems was a challenging, time-consuming undertaking that relied on a narrow formal verification expertise and required many person-years of effort. We argue, however, that the balance of practicality is finally changing with development of automated verification tools that leverage a unique combination of the linear type system of Rust and automated verification based on satisfiability modulo theories (SMT). Our work leverages, Verus, a new SMT-based verifier for Rust, for development of a minimal yet practical microkernel, Atmosphere. Atmosphere is designed as a full-featured microkernel conceptually similar to the line of early L4 microkernels. We develop all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification with Verus. Our experience shows that Verus provides a collection of practical features that significantly lower the burden of a verification effort making it possible to reason about correctness of the low-level systems code, e.g., low-level memory and address space management, recursive data structures like linked lists and page tables, etc. On average our code has proof-to-code ratio of 7.5:1 which is significantly lower than in prior approaches.

CCS Concepts: • **Software and its engineering → Software verification**; **Functionality**.

1 Introduction

Despite decades of progress, reasoning about correctness of operating system kernels remains a complex, time-consuming undertaking. The kernel runs on bare-metal and requires reasoning about everything from the low-level details of hardware execution environment to the system call interface exposed to user applications. Correctness of the kernel rests on the proofs about physical and virtual memory, low-level details of memory and resource management, lifetimes of numerous kernel data structures, correctness of synchronization in a parallel and concurrent environment, implementation of recursive data structures that are optimized to extract the last bits of performance and more.

Historically, verification of even a simple kernel required tens of person-years to complete. For example, despite numerous careful design decisions aimed to minimize verification effort, e.g., avoiding memory management in the kernel and big-lock synchronization, verification of the first formally verified microkernel, seL4, took 20 person-years for a system of 10KLOC [25].

To address complexity of verification effort, several projects explored ideas of "push-button" verification that attempted direct translation of the kernel code into a satisfiability modulo theories (SMT) expression that was then checked by an SMT solver [32, 42]. While achieving nearly automated verification, such approaches required numerous simplifying assumptions about the kernel interface and kernel's internal organization, pushing most of the typical kernel functionality to unverified user libraries. Trying to approach complexity of verification from a different angle, several systems advocated for use of clean-slate programming languages like Dafny [27] designed for a high degree of proof automation [20, 21, 41]. Ironclad was the first system to scale verification effort to the whole system from the application layer down to the kernel assembly [21]. Yet numerous algorithmic simplifications and managed runtime of the Dafny language hindered performance of the system. For example, Ironclad relied on a simple and inefficient stop-the-world garbage collector from the Verve microkernel [41].

We argue, however, that the balance of practicality is finally changing with development of automated verification tools that leverage a unique combination of the linear type system of Rust [13, 24] and automated verification tools like Verus designed to provide a high-degree of proof automation similar to Dafny but in Rust [26]. Arguably, a combination of Rust and Verus for the first time provides support for practical, low-burden verification of low-level systems.

Rust is the first *practical* language that can enforce memory safety without garbage collection. In contrast to managed languages Rust relies on a restrictive type system that controls ownership of objects allocated on the heap [13]. Control

[*] Both authors contributed equally to the paper

KISV '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0411-6/23/10.
https://doi.org/10.1145/3625275.3625402

over aliases provides a way to statically reason about lifetimes of objects and generate an explicit destructor at compile time, hence ensuring temporal safety without garbage collection. Historically, even the fastest safe languages like Go and C# resulted in a 36-42% overhead compared to unsafe C on high-performance system workloads like network device drivers [12]. Rust, however, supports development of safe code that stays within few percents of unsafe C implementations [12, 33].

Verus complements Rust with the Dafny-like automated SMT-based reasoning [26]. Similar to Dafny, Verus provides a high degree of automation and user-friendliness which allow system developers to work on formally-verified systems directly in the language designed to build such systems and arguably with minimal background in formal verification. Verus uses one language for specifications, proofs and executable code. Moreover, due to the lack of managed runtime, verified Rust code can be compiled and executed on bare metal.

Compared to Dafny, Verus further simplifies verification effort. First, Verus uniquely leverages the linear type system of Rust to lower complexity of reasoning about the heap [26, 29]. Second, Verus provides an elegant way to reason about a range of unsafe pointer constructs through an idea of linear permissions [26]. To ensure practicality of the otherwise linear language, Rust allows escape from the ownership rules through its unsafe subset, e.g., to implement doubly-linked lists, aliases, concurrent primitives, etc. Historically, reasoning about unsafe subset of Rust remained challenging [22] thus limiting verification of Rust code to its safe subset [3]. Linear permissions however provide a way to reason about limited but powerful subset of unsafe constructs that in practice support correctness proofs for typical kernel data structures, e.g., linked lists, synchronization primitives, and constructs that implement interior mutability.

Our work presents an early prototype that leverages Rust and Verus for development of a minimal yet practical microkernel, Atmosphere. Specifically, we develop all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification with Verus. Similar to prior work, we carefully design the kernel to keep verification complexity under control. Still, Verus allows us to implement typical kernel data structures like linked lists, support verified memory allocation, develop proofs about page tables, etc. Our initial experience shows that even through some compromises are necessary, a combination of Verus and Rust significantly reduces verification effort. On average our code has proof-to-code ratio of 7.5:1 which is significantly lower than in prior approaches [17, 25]. Moreover, Rust and Verus allow us to reason about a microkernel with a feature-rich interface that is conceptually similar to the line of classical L4 microkernels, i.e., without the capability interface [11].

2 Background

Early verification efforts were aimed at attaining the highest A1 assurance rating defined by the “Orange Book” [2] but remained largely unsuccessful due to limitations of existing verification tools [14, 16, 23, 36]. SeL4 became the first system to demonstrate a way to achieve verification of a practical microkernel [25]. SeL4 adopted a unique design choice in which the kernel does not perform any memory allocation at all, but instead pushes all allocation decisions to user processes (which remain unverified). This enabled proving isolation between subsystems, but resulted in an unusual system model which required user code to manage their memory through a capability interface. Verification of seL4 involved 200,000 lines of proof code of the Isabelle/HOL theorem prover for 8,700 lines of C and required 22 person-years [25].

Hyperkernel used LLVM intermediate representation (IR) generated from C which was then translated into a satisfiability modulo theories (SMT) expression checked by the Z3 SMT solver [32]. Hyperkernel demonstrated high degree of automation but at the cost of severe limitations in kernel functionality. To support automated translation, 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 [21] addressed complexity of the verification effort through a combination of Dafny [27], Boogie intermediate verification language [4], and Z3 SMT solver [9]. Designed explicitly for verification, Dafny allowed reducing the size of the proof without degrading expressiveness of implementation (3 person-years). Ironclad relied on a previously verified microkernel, Verve [41], and mainly concentrated on verifying cryptographic libraries, device drivers (trusted platform module), and several applications [21]. Note that verification of Verve addressed only safety but not functional correctness of the kernel [41].

CertiKOS [18] and μC/OS-II [40] were aimed at verification of concurrent systems through the use of the Coq interactive theorem prover [1]. 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 [30]. Specifically, SeKVM decomposed the hypervisor into two separate layers and verified the privileged core using Coq. The core was further split into layers built on top of an abstract machine model, with each successively simplifying the model for upper layers. The top-level specification was used to prove that any implementation of the deprivileged upper layer can maintain the confidentiality and integrity of the VM data. Even though SeKVM relied on

ClightGen [28] 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% [31].

Finally, similar to ours, a recent project, verified NrOS [6], uses Verus to verify an existing NrOS kernel [5] which is also developed in Rust. NrOS verifies the code for page-table management and its core concurrency mechanism, node replication, yet using Dafny [19]. Compared to our work NrOS aims at verification of a larger kernel that while being a microkernel implements a kernel-level file system. We choose a more pragmatic path of verifying only a minimal microkernel that we design from scratch to ensure that verification is feasible. We plan to approach verification of user-level services like device drivers, network stacks and file-systems separately.

Verus Verus is a new verification tool for Rust that supports semi-automated reasoning by using an SMT solver [26]. Verus attempts to replicate success of earlier verification frameworks combining proofs and development in a single language [1, 10, 27, 37]. Verus, however, is different in several important ways. First, instead of relying on a verification-centric language, Verus works by extending Rust, a language which is already designed for safe development of low-level systems. Second, Verus benefits from the Rust's linear type system to simplify proofs, support verification of pointer-manipulating and concurrent Rust code, and implement efficient SMT encoding. For instance, instead of having to reason about the complex semantics of heap based mutation, Verus can encode operations on any any mutable reference as a sequence of transformations on immutable values, as Rust guarantees such references will be linear.

To introduce the basic features of Verus, we describe a partial specification for a page backed doubly linked list that we use in Atmosphere to support management of dynamic data structures like lists of endpoints, threads, processes, etc., (Listing 1). At a high level, Verus allows one to write executable code, specifications for modelling the behavior of a system, and proofs that the executable code conforms to the specified behavior. 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` as well as keywords like `requires` and `ensures` (lines 31-39) to specify preconditions and postconditions of functions. For example, the `well_formed()` function specifies what conditions the list must satisfy to be in a valid state (one such condition being every node pointer in the list must have a corresponding owned page). Verus modifies the Rust compiler to elide specifications and proofs (*ghost code*) during compilation time. For instance the `Map` and `Seq` types are mathematical models of the underlying code, but are ghost types thus do not incur any runtime overhead.

The list itself is composed of a combination of a standard doubly linked list of nodes (lines 14-15), a reverse singly linked list of free nodes (line 18) (representing the capacity of the linked list), and a reverse singly linked list of memory pages that provide memory for the individual elements of the list (lines 20-21). Each sublist is further modelled through the ghost `Seq` type: as an abstract sequence of raw pointers which mirror the order of the list elements. This allows us to model the linked list as a simple sequence of values, hiding the inner complexity of the linked list from any other data structures that need to reason about it.

```
1   struct Node<T> {
2     contents: T, next: Option<NodePtr<T>>, prev: Option<NodePtr<T>>
3   }
4   struct PageNode { prev: Option<PageNodePtr>, }
5
6   type PagePerm<T> = PageArena<Node<T>, PageNode>,
7   type NodePtr<T> = PageElementPtr<Node<T>>;
8   type PageNodePtr = PageMetadataPtr<PageNode>;
9
10  struct LinkedList<T> {
11    perms: Map<PagePPtr, PagePerm<T>>,
12    // Doubly linked list of elements
13    ptrs: Seq<NodePtr<T>>,
14    head: Option<NodePtr<T>>,
15    tail: Option<NodePtr<T>>,
16    // Reverse singly linked list of free nodes
17    free_ptrs: Seq<NodePtr<T>>,
18    free_tail: Option<NodePtr<T>>,
19    // Reverse singly linked list of backing pages
20    page_ptrs: Seq<PageNodePtr>,
21    page_tail: Option<PageNodePtr>,
22  }
23
24  spec fn well_formed<T>(list: &LinkedList<T>) -> bool {
25    // ...
26    && forall |i: nat| 0 <= i < list.ptrs.len() ==>
27      list.perms.domain().contains(list.ptrs[i].page_pptr())
28  }
29
30  fn push_back<T>(list: &mut LinkedList<T>, v: T)
31    requires
32      well_formed(old(list)),
33      old(list).free_ptrs.len() > 0,
34    ensures
35      well_formed(list),
36      list.free_ptrs.len() == old(list).free_ptrs.len() - 1,
37      list.ptrs.len() == old(list).ptrs.len() + 1,
38      v == list.perms[list.ptrs.last().page_pptr()]
39        .value_at(list.ptrs.last().index()).contents
40  {
41    let ptr: NodePtr<T> = list.free_tail.unwrap();
42    let perm: &PagePerm<T> = list.perms[ptr.page_pptr()];
43    let node: &Node<T> = ptr.borrow(perm);
44    // Update free ptrs linked list and model sequence
45    list.free_tail = node.prev;
46    list.free_ptrs = list.free_ptrs.take(list.free_ptrs.len() - 1);
47    // Update contents of this ptr
48    ptr.put(perm, Node { contents: v, prev: list.tail, next: None });
49    // Update ptrs linked list and model sequence
50    if list.tail.is_none() {
51      list.head = Some(ptr);
52    }
53    list.tail = Some(ptr);
54    list.ptrs = list.ptrs.push(ptr);
55  }
```

Listing 1. Partial implementation of a resizable linked list.

As we will discuss in Section 4, the linked list leverages linear ghost types [26] and the `PageArena` abstraction to reason about raw pointers in a way which is safe and memory efficient. Because we model permissions as linear objects, we can only read from a given raw pointer if we hold an immitable reference to the corresponding permission (line 43), and we can only write to it if we hold a mutable reference to the corresponding permission (line 48). As such all accesses of the raw pointer are checked by the borrow checker and proven to be safe.

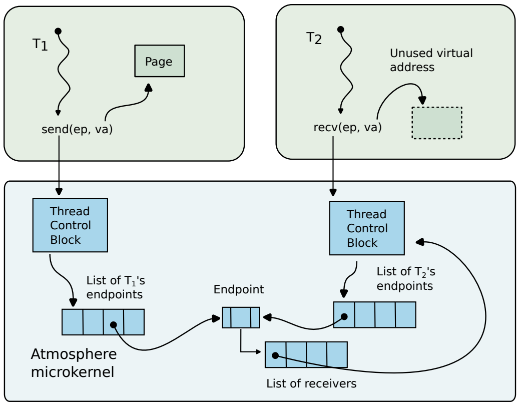

Figure 1. Architecture of Atmosphere. Thread T_1 invokes the `send()` system call to pass a page to T_2 which is already waiting on the endpoint inside the microkernel.

3 Architecture

Atmosphere is a full-featured microkernel conceptually similar to the line of classical L4 microkernels before the introduction of a capability interface in seL4 [11]. Similar to other microkernels, Atmosphere pushes most kernel functionality to user-space, e.g., device drivers, network stack, file systems, etc. The microkernel supports a minimal set of mechanisms to implement address spaces, memory management, interrupt dispatch, inter-process communication, and threads of execution that together with address spaces implement an abstraction of a process. Each process has a page table and a collection of schedulable threads. Atmosphere allows threads to control layout of their virtual address space through a collection of system calls that support mapping and unmapping of pages as well as receiving pages from other threads via communication endpoints. Atmosphere 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.

Atmosphere implements a verified page allocator and develops a novel scheme that allows allocation of fine-grained objects somewhat similar to Slab in Linux, which allows the kernel to implement dynamic data structures like linked lists.

Atmosphere allows processes to communicate via endpoints. A sender thread can pass scalar data, references to memory pages, 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.

The endpoints work as capabilites that allow connections between processes (processes can exchange endpoints and then establish regions of shared memory). Shared memory regions provide support for efficient communication [7, 8, 15]. 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.

System call interface Atmosphere provides support for creating new processes and threads, allocating and mapping pages of memory, creating and exchanging communication endpoints. The microkernel however does not support loading of new processes and instead delegates it to a user-space protocol. Specifically, the parent process is allowed to share an endpoint with the child (by passing it as an argument to the system call that creates a new process). The kernel creates a minimal address space for the child with a simple statically-linked trampoline code – same for all new processes in the system. The boot code uses the endpoint to communicate with the rest of the system that allows it to load the linker and the process binary from the file system. Control over the process boot protocol allows us to implement traditional fork and exec primitives. To implement fork, the child process communicates with the parent to first map its address space and then copy it. Similarly the parent allows the child to inherit its file descriptors in the file system.

Device drivers and interrupts Atmosphere implements all device drivers as user-space processes. The microkernel allows a user thread to register for an interrupt by trying to receive a message from an interrupt endpoint. A low-level interrupt handler unblocks all threads waiting on a specific interrupt. A trusted boot loader creates the first process that has access to all PCIe regions and can share them with device driver processes.

4 Verification

Specifications Atmosphere captures high-level behavior of the system as a collection of high-level specifications for the microkernel interface. This is similar to previous approaches [6, 32]. For example, the specification of a system call that creates a new thread reflects that a new thread is added to the list of threads of the same process and can access the same address space. We then prove that the implementation of each system call is a refinement of its high-level specification. That is, when a microkernel call is executed, its effect to the whole system is equivalent to the change in a high-level specification.

Internally, we structure specifications as the ones that define well-formedness of the kernel state, i.e., all data structures and resources managed by the kernel are well-formed, and the ones that capture functional behavior, i.e., updates to the system's state. For example, a well-formed pool of physical pages is such that for each physical page it is either mapped by one or more alive processes or stored in the free pool. The kernel is structurally well-formed at all times if and only if it maintains kernel structural integrity at boot time and after each function call.

We capture functional behavior of the system using Floyd-Hoare logic: as a collection of pre and post conditions on its

```
pub fn pop_scheduled_thread(&mut self)
  -> (thread_ptr: ThreadPtr)
  requires
    old(self).wf_scheduler(),
  ensures
    self.threads[thread_ptr].state == running,
    self.scheduler ==
    old(self).scheduler.subrange(1, old(self).scheduler.len()),
    self.wf_scheduler(),
```

Listing 2. Functional correctness specification

state. For example, Listing 2 illustrates a simplified specification for the scheduler function, `pop_scheduled_thread()`, that picks the next thread to run. The `requires` clause contains preconditions that must hold before the invocation of the function (in this case the scheduler must be in a well-formed state). The postconditions in `ensures` clause describe how the state of the system will change after the invocation. Specifically, `pop_scheduled_thread()` contains three postconditions, in which the first two define the functional correctness of `pop_scheduled_thread()` as: (1) The state of the scheduler after the invocation will be the same as equivalent to its previous state with its oldest thread popped (FIFO scheduling). (2) The popped thread is correctly marked as running. (3) Scheduler remains well-formed in the new state.

We execute Atmosphere under a big lock with interrupts disabled to sidestep complexity of reasoning about concurrency (previous work argues that big-lock insignificantly affects performance of a microkernel system [34]). This allows us to model each kernel invocation as an atomic transaction on the kernel state. Atmosphere ensures kernel structural integrity and functional correctness before and after each system call (and each interrupt transition). Each kernel invocation transitions the kernel from one well-formed state to another and adheres to the high-level specification which captures the effect of the system call.

Overall, this allows us to proof high-level properties of the kernel. For example, we define and prove Atmosphere kernel memory correctness specifications as: (1) The kernel memory pages and user-mapped pages are disjoint. (2) The kernel components do not overlap. (3) The whole system has no memory leaks.

Raw pointers and memory management The true power of Verus comes from its support for reasoning about raw pointers and objects allocated on the heap via a combination of *permissioned pointers*, i.e., `PPtr<T>`, and a linear ghost permission type `PointsTo<T>` [26]. Specifically, to read from a raw pointer one requires an immutable reference to the permission corresponding to that pointer (writing requires a mutable reference to the permission). Hence, accesses to raw pointers follow the normal ownership model in Rust. This proves that accesses are safe, linearized, and that pointer provenance is upheld.

However, Verus makes a critical simplifying design choice – it trusts the memory allocator to create objects behind permissioned pointers – something we would like to avoid. While it's possible to verify the allocator, the ergonomics of the proof relies on the abstraction of permission pointers resulting in the chicken-and-egg problem.

In order to leverage the abstraction of linear permissioned pointers, we introduce several mechanisms that support reasoning about raw pointers but without requiring trust in the memory allocator. First, we develop a simple verified memory allocator that can allocate memory in the units of pages, (i.e., 4096 bytes in our system). The allocator tracks page state with a static array and hence does not depend on permission pointers. We then change Verus to allow construction of only one permission pointer type – a pointer to an untyped page of bytes, `PPtr<[u8; 4096]>`, which can only be created from a page of memory obtained from the allocator. In other words, instead of trusting a generic allocator, we first allocate a page from a simple page allocator and then retype it into a permission pointer to that page. Finally, to support allocation of smaller objects, we develop an abstraction of an *arena* that allows us to split a page into a collection of smaller objects.

Therefore, to allocate an object of type `T` (the object must be smaller than a page), we allocate a ghost data structure, `PageArena`, which splits a 4 KiB page into an array of values. A `PageArena` is created from an untyped page (`PPtr<[u8; 4096]>`) and its corresponding permission. Once created, fat pointers to typed elements (`PageElementPtr<T>`) can then be derived from the arena. In order to access an element, both the fat pointer and the underlying ghost arena (immutable borrow for reads, mutable borrow for writes) are required. Access to typed elements is thus linearized in a manner similar to permissioned pointers in Verus.

Since a ghost `PageArena` instance represents the permission to access all elements in the page, deallocation is made possible by converting the ghost arena back to an untyped page permission (`PointsTo<[u8; 4096]>`). Without the ghost arena, all previously-allocated elements become permanently inaccessible and the page can thus be reused. This is in contrast to splitting an untyped page into individual permissions, where deallocation of a page would require tracking all permissions to typed values in the page.

For each `PageArena<T, MT>`, we further provide a way to include optional metadata (`MT`) for each page. This metadata is included after the typed elements and can be used to link multiple page arenas together to construct resizeable containers like linked lists.

To bootstrap the verified memory allocator, we extend Verus to support the creation of `PPtr<[u8; 4096]>`s from physical pages passed by the boot manager. We enforce type safety with a transparent wrapper type, `BootPage`, that follows Rust's *new type idiom* [38] and acts as a token that represents a unique physical page. The wrapper has a single, private field that contains the physical address of the page. Since the wrapper type does not have a public constructor method, the Rust type system guarantees that there is no way to construct it

```
 1  fn grow<T>(list: &mut LinkedList<T>, page: PagePPtr, perm: PagePerm<T>)
 2    requires
 3      well_formed(old(list)),
 4      page.id() == perm.page_base(),
 5    ensures
 6      well_formed(list),
 7      list.free_ptrs.len() == old(list).free_ptrs.len()
 8        + PagePerm::<T>::capacity(),
 9      list.page_ptrs.len() == old(list).page_ptrs.len() + 1,
10  {
11    let offset = list.free_ptrs.len();
12    // Generate pointers for each element in the page,
13    // and add them to the free pointers list
14    let mut idx: usize = 0;
15    // Inductively show all new free pointers are valid and unique
16    while idx < PagePerm::<T>::capacity();
17      invariant
18        idx <= PagePerm::<T>::capacity(),
19        list.free_ptrs.len() == offset + idx,
20        forall |k: nat| 0 <= k < idx ==>
21          (list.free_ptrs[offset + k].page_ptr() == page &&
22          list.free_ptrs[offset + k].index() == k),
23        // ...
24    {
25      // Extend the free pointers list and model sequence
26      let ptr = NodePtr::<T>::new(page, idx);
27      ptr.put(perm, Node {
28        contents: undefined(), prev: list.free_tail, next: None
29      });
30      list.free_tail = Some(ptr);
31      list.free_ptrs = list.free_ptrs.push(ptr);
32      idx += 1;
33    }
34    // Using the metadata type, store and extend the list
35    // of pages owned by this data structure
36    let page_node = PageNodePtr::new(page);
37    page_node.put(perm, PageNode { prev: list.page_tail });
38    list.page_tail = Some(page_node);
39    list.page_ptrs = list.page_ptrs.push(page_node);
40    // Take ownership of the page permission
41    list.perms.insert(page, perm);
42  }
```

Listing 3. Example use of `PageArena` for growing the linked list.

in safe Rust. The `#[repr(transparent)]` attribute further provides ABI stability by ensuring that the wrapper has the same data layout as the enclosed raw pointer, so the `BootPage` tokens may be handed off by the boot manager to the microkernel.

Dynamic data structures A combination of page arenas and the ability to store metadata for each page allows us to construct complex dynamic data structures such as page backed linked lists (Listing 1). Crucially, using the `PageArena` abstraction, we can grow the linked list using our verified page allocator in a way which is memory efficient (as it gives us the ability to allocate many smaller objects from each pages). In the `grow()` function (Listing 3) we take a single page, split it into a large number of pointers to smaller node objects and append these pointers to the free list (lines 16-33). In doing so we grow the capacity of the linked list. We also use the metadata object to store a list of pages owned by the linked list (lines 36-39), which is used for deallocation.

Field-level mutation At the moment, Verus does not support returning mutable references from functions and support for mutable references in function arguments is limited to special cases. To allow mutation of the value, permissioned pointers in Verus expose setters. This comes with the downside of not being able to modify specific fields of a data structure without copying the entire structure – an overhead prohibitive for a microkernel.

To support efficient field-level mutation, we implement a procedural macro that generates getters and setters for each field in a data structure, as well as the corresponding trusted specifications that update the abstract state. The generic nature of the procedural macro facilitates the auditing of generated specifications for all structures it's applied to.

In the future, it's expected that Verus will add support for returning mutable references by adopting prophecy variables [39].

Page memory allocator Atmosphere implements page memory allocator with two data structures: 1) a fixed size array that tracks the state of every page, and 2) a fixed size queue that implements a single-linked list of free pages. A trusted boot manager enumerates all available memory in the system and creates the initial list for the page allocator. We use the page allocator to allocate coarse-grained data structures like threads, processes, endpoint, pages for the page table, and to allocate page arenas for dynamically growing linked lists. A page in Atmosphere is in one of the three states: `free`, `allocated` to use inside the kernel, i.e., provide memory for one of the kernel data structures, or `mapped` by user processes as part of the process address space. A mapped page contains a reference counter that tracks the number of times the page is mapped (we allow pages to be mapped multiple times inside the process and share pages across processes).

Paging Atmosphere contains a verified subsystem to support four-level paging. For simplification, we model an address space as a tree where each present entry in a paging structure points to a paging structure in the next level, or a 4 KiB data page. We ensure that it is impossible for an entry to point to a paging structure of the wrong level or is not well-formed. Furthermore, it should not be possible to interpret a data page as a paging structure or vice versa.

To achieve these goals, Atmosphere implements paging using strongly-typed tables for each level of the paging structure, with linearity enforced using permissioned pointers. Each paging structure is modeled as a generic data structure `PagingLevel<E, T>` containing 4096 entries (`E`) with each present entry pointing at a target (`T`). The allowed types of `E` and `T` are constrained by bounds on traits which provide common methods that indicate entry presence as well as perform lookup and mutation. For each present entry, the paging structures also contain ghost state to keep track of the permission corresponding to the target.

Translation lookaside buffer (TLB) Atmosphere uses tagged TLB to implement specifications and verification of the code responsible for flushing the TLB in a multi-processor system. CPU running in tagged TLB mode uses the first 12 bits of the `cr3` as a process context identifier (PCID) to identify different address spaces and completely ignores the `cr3` address bits (lower bits) while performing a TLB lookup. We specify the expected behavior of the virtual to physical

Name	Language	Spec Lang.	Proof-to-Code Ratio
seL4	C+Asm	Isabelle/HOL	20:1 [25]
CertiKOS	C+Asm	Coq	14.9:1 [17]
SeKVM	C+Asm	Coq	6.9:1 [30]
Ironclad	Dafny	Dafny	4.8:1 [21]
NrOS	Rust	Verus (Rust eDSL)	10:1 [6]
Atmosphere	Rust	Verus (Rust eDSL)	7.5:1

Figure 2. Proof effort for existing verification projects.

memory translation mechanism as a map containing correspondence between the virtual and physical addresses for each PCID. For each CPU's TLB, we model it as a similar map between the virtual and physical addresses for each PCID.

Interrupts and context switch Unlike a typical kernel, Atmosphere does not save execution state of the thread on a kernel stack during the context switch. Instead, all kernel functions (interrupt handlers and system calls) run to completion and if the context switch is required a small trusted helper saves the user-state of the thread (i.e., it's trap frame) in the thread data structure. In Atmosphere, each CPU has one kernel stack for all threads (note, each CPU maintains another stack for processing non-maskable inter-processor interrupts that we use for TLB invalidation). This design choice allows us to simplify the proof. When the kernel function returns, the kernel is in a well-formed state, and the kernel stack pointer is restored to the original position.

5 Implementation

Build environment The Atmosphere microkernel consists of both *verified* and *non-verified* components built using a trusted compilation environment, e.g., Atmosphere 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.

6 Evaluation

We carry development of Atmosphere on one of the CloudLab [35] c220g5 servers which are configured with two Intel Xeon Silver 4114 10-core CPUs running at 2.20 GHz, 192 GB RAM. Those machines run 64-bit Ubuntu 20.04 Linux with a 5.4.0 kernel. Verus takes approximately 20 seconds to reason about verified parts of the kernel. Individual functions take 4 seconds at most when the runtime checks are turned on (this is inline with recent work [6]). With runtime checks turned off, Verus takes roughly 30 minutes to finish the proof and spends upto 850 seconds at most on a function. Therefore, we believe that there is room for improvement both the Verus verification toolchain and in how we structure our proofs.

Atmosphere has a proof-to-code ratio of 7.5:1 which is a significant improvement compared to the existing formally verified microkernels SeL4 [25] and CertiKOS [17], which have proof-to-code ratio of 19:1 and 20:1, respectively (Figure 2).

7 Conclusions

Our early experience with Atmosphere demonstrates that a combination of a linear type system, a practical language designed for development of low-level systems, and an automated reasoning tool, Verus, takes a huge step towards enabling low-burden verification of kernel code. While we had to be conscious of the internal organization of the system to make sure that verification is possible, in most cases Verus provides a way to move forward without significant limitations on the kernel code and with an excellent degree of proof automation.

Acknowledgments

We would like to thank KISV'23 reviewers for various insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 2313411, 1837127 and 2341138, and Amazon.

References

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

[2] *Department of Defense Trusted Computer System Evaluation Criteria*, pages 1–129. Palgrave Macmillan UK, London, 1985.

[3] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. Verifying rust programs with smack. In Shuvendu K. Lahiri and Chao Wang, editors, *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 11138 of *Lecture Notes in Computer Science*, pages 528–535. Springer, 2018.

[4] 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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] 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 '21, pages 295–312. USENIX Association, July 2021.

[6] 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 '23, page 158–165, 2023.

[7] 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)*, pages 313–326, 2009.

[8] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, pages 136–140, 2008.

[9] 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)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[11] Kevin Elphinstone and Gernot Heiser. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, 2013.

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

[13] Mozilla Foundation. The Rust programming language. `https://doc.rust-lang.org/book/`.

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

[15] 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.

[16] B. D. Gold, R. R. Linde, and P. F. Cudney. Kvm/370 in retrospect. In *1984 IEEE Symposium on Security and Privacy*, pages 13–23, 1984.

[17] 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 '11, page 3, 2011.

[18] 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 '16, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[19] 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 '23, pages 911–929, Boston, MA, July 2023. USENIX Association.

[20] 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 '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[21] 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 '14, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. volume 2, New York, NY, USA, December 2017. Association for Computing Machinery.

[23] 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*, 17(11):1147–1165, 1991.

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

[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 '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[26] 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*, 7(OOPSLA1):85:286–85:315, April 2023.

[27] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[28] Xavier Leroy. The compcert c verified compiler: Documentation and user's manual. September 2014.

[29] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–28, 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)*, pages 3953–3970, August 2021.

[31] 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 '23, pages 851–869, Boston, MA, July 2023. USENIX Association.

[32] 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 '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

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

[34] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a Microkernel, a Big Lock Is Fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[36] 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*, pages 238–245, October 1988.

[37] 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 '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery.

[38] The Rust Project. New Type Idiom. `https://doc.rust-lang.org/rust-by-example/generics/new_types.html`, 2023.

[39] The Verus Contributors. Returning mutable references. `https://github.com/verus-lang/verus/discussions/35`, 2023.

[40] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 59–79, Cham, 2016. Springer International Publishing.

[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 '10, pages 99–110, New York, NY, USA, 2010.

[42] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 275–290, New York, NY, USA, 2019. Association for Computing Machinery.