Beyond isolation: OS verification as a foundation for correct applications

Matthias Brun
ETH Zurich
matthias.brun@inf.ethz.ch

Reto Achermann
University of British Columbia
achreto@cs.ubc.ca

Tej Chajed
VMware Research
tchajed@vmware.com

Jon Howell
VMware Research
howell@vmware.com

Gerd Zellweger
VMware Research
gzellweger@vmware.com

Andrea Lattuada
VMware Research
lattuada@vmware.com

ABSTRACT
Verified systems software has generally had to assume the correctness of the operating system and its provided services (like networking and the file system). Even though there exist verified operating systems and file systems, the specifications for these components do not compose with applications to produce a fully verified high-performance software stack.

In this position paper, we lay out our vision for what it would look like to have a verified OS with verified applications, all with good multi-core performance. We’ve explored a part of the verification by proving a page table correct already, but the larger goal is to lay out a vision for an ambitious project that supports an application verified from its high-level specification down to the hardware.

ACM Reference Format:

1 INTRODUCTION
Suppose we wanted to write a high-assurance distributed system, down to its implementation. Today, we have no way to verify such a system together with the operating system it depends on. Instead, the user-space components have to be verified against a manually derived, error-prone environment specification [15]. It is time to build a high-performance, multi-core, and formally verified OS kernel whose specification supports verifying realistic multi-core applications, and which provides enough services to even implement these interesting applications in the first place. There exists verified concurrent software that implicitly assumes the correctness of the operating system and some of its essential services like the file system and networking. There also exist verified operating systems and file systems. However, the specifications and sometimes even programming APIs for all of these components are not mutually compatible.

As an example of the kind of application we are interested in verifying, consider the data-storage node in a distributed block store like GFS [16] or S3. In fact, Amazon even describes their use of lightweight formal methods to verify such a storage node for the upcoming version of S3 [8]. If we wanted to verify not just core logic but also its system dependencies, we would want to have verified versions of the following OS components:

- a scheduler (to run processes),
- memory management (physical memory, page tables),
- a filesystem (persistence, sharing),
- device drivers (network controller, disk controllers, interrupt controller, timer, serial/graphical output),
- process management (spawning, waiting, signals, killing),
- per-process threads and synchronization mechanisms: mutexes, semaphores, condition variables,
- some network stack for communication,
- system libraries (e.g., libc).

We believe that with the right combination of OS design and verification tooling and techniques, it is finally possible to construct a verified operating system that can be a foundation for fully verified client applications, with the complexity and performance characteristics of the unverified but critical software running in our data centers and mobile devices right now. To build such a foundation, we need to define correctness for our operating system in a way that enables
Table 1: Comparison of OS verification projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>seL4</th>
<th>Verve</th>
<th>Hyperkernel</th>
<th>CertiKOS</th>
<th>seKVM+VRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel memory safety</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Specification refinement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Security properties</td>
<td>✓</td>
<td>✗</td>
<td>(✓)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multi-processor support</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process-centric spec</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 2: Verified OS components

<table>
<thead>
<tr>
<th>Feature</th>
<th>seL4</th>
<th>Verve</th>
<th>Hyperkernel</th>
<th>CertiKOS</th>
<th>seKVM+VRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Filesystem</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Complex drivers</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process management</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Threads and synchronization</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Network stack</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>System libraries</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

2 BACKGROUND

The most closely related work to our envisioned system are earlier verified operating systems, namely seL4 [23], Verve [37], Hyperkernel [32], CertiKOS [17] and SeKVM [26, 27] including weak memory (VRM) [35]. Table 1 summarizes the achievements of these major OS verification projects.

All of them verify memory safety of the kernel memory and the system’s adherence to some high-level spec. SeL4 [30], Hyperkernel and SeKVM go one step further and formally prove crosscutting theorems about their high-level specifications to ensure security properties such as integrity and confidentiality. CertiKOS does so only for an earlier, single-threaded version of the system with IPC disabled [13]. Only CertiKOS and SeKVM are verified for multi-processor hardware. However, their performance is not competitive with unverified, scalable kernel designs [4, 12, 36] proposed by the community over the past decade. These systems can achieve near-linear scalability with a large number of cores for kernel operations whereas conventional OS designs suffer from degraded performance due to lock contention.

SeKVM is a modified version of the Linux KVM hypervisor. It consists of a small core that is formally verified to enforce confidentiality and integrity of VM data, whereas most functionality is provided by unverified code. The security properties are proven as noninterference properties over the high-level spec. SeKVM’s original proofs assume sequential consistency of the memory model but are extended in a later publication [35] to assume a model of ARM’s relaxed memory semantics instead.

SMT solvers have been used in Verve [37], Hyperkernel [32] and the Serval framework [31], which adapted Hyperkernel’s approach to verify other kernels. These approaches severely limit the scope of either the verified properties or the verified code’s expressiveness: Verve uses only a single address space and focuses on verifying type and memory safety, while Hyperkernel and Serval require statically-bounded loops and disallow recursion.

The specifications in all these projects are designed primarily to facilitate proofs of certain properties, such as isolation...
or type safety. As a result, they do not lend themselves easily to interfacing with verified user code. For example, while all projects verify memory management to some degree, seL4 and Hyperkernel among others rely on self-paging or external paging logic, and only verify that page directory entries are installed safely by the kernel, but do not verify the address space management logic in user-space. SeKVM additionally proves that the page table corresponds to an abstract map from virtual to physical addresses. However, in a user-space process, we are mainly interested in knowing that the implemented paging logic results in the expected memory semantics, i.e., we want a process-centric spec.

Table 2 looks at the scope of these projects concerning the components mentioned in Section 1. The verified operating systems provide a minimal subset of verified services at the OS level, enough to run applications and to verify their isolation but they push most of these components (file system, network, and device drivers) into the unverified userspace. The system spec that a process interacts with is very narrow. Aside from the projects already discussed in this paper, several other efforts have verified parts of an OS such as hardware initialization [3], file systems [1, 10, 11], locking primitives [28, 33], network functions [34], cryptographic primitives [5, 7] etc. Even though many such verified components exist, it is not clear how they can be composed to form a complete, verified operating system with a rich contract provided to applications.

3 THE CLIENT APPLICATION CONTRACT

What does it mean for an OS to be correct? We propose a definition of OS correctness based on the behavior of applications running on top.

Following OSTEP [2], an operating system does three things: it virtualizes resources (especially memory and the CPU, but also for example the network device), enables concurrency (both thread- and process-level), and exposes a file system for persistence. Our goal is to carry out verification of the applications running on top of this spec, which then needs to directly describe the virtualized execution environment each process experiences: an abstract model which only has virtualized memory, processes, threads, and the abstract state of the network and file system. The organization we propose for this high-level OS spec is to divide it into two categories: the execution model provided by the OS (which captures how the memory and CPU are virtualized) and the behavior of the supported system calls (to access the network and file system, and to use the OS-provided concurrency primitives).

Execution model. In this approach, the memory and CPU execution model that the OS provides is still at the level of machine instructions even in its abstract spec, which would be cumbersome to directly write verified software against.

We propose that pragmatically it makes sense to take a systems programming language like Rust and assume that the operating system’s abstract model of CPU execution and memory (including concurrency between OS threads) matches the semantics expected by the Rust compiler. That is to say, we would verify application code assuming correctness of the compiler, which is less likely to be a source of bugs than a manually derived execution environment spec, which is often error-prone even in verified systems [15].

System calls. We illustrate our proposed approach for system calls with the example of a hypothetical, simplified read syscall. The high-level spec for the system call is a state machine, whose state contains the file descriptors’ current state. Execution of the syscall corresponds to a transition, which relates the old state pre to the new state post:

```rust
pub fn read(pre: &mut Sys, fd: usize, buffer: &[u8]) -> (mut usize, usize)
```

The transition defines the operation that needs to be refined by the kernel’s implementation of the syscall as well as the semantics that a user space application can rely on. From the perspective of user space code, this interface is represented as part of a type Sys that encapsulates the syscall interface.

```rust
pub fn read(pre: &mut Sys, fd: usize, buffer: &[u8]) -> (mut usize, usize)
```

The function’s ensures clause describes how the system model – as perceived by the client application – changes in response to the call to read. The view() functions abstract the concrete runtime values to mathematical representations. sys.view() and old(sys).view() are State objects from the high-level OS spec above and represent the system state seen by the user space application before and after the call.

Notably, this high-level spec for system calls hides many details of the OS implementation. For example, when the OS makes a context switch, processes view this as just another interleaving of threads. Spawning a thread might require allocating kernel data structures and modifying scheduler state, but at the abstract level simply creates a new thread associated with the current process. Even reading and writing memory is a complicated hardware operation (involving page table and TLB lookups) which is abstracted to a simple high-level operation; this refinement is covered by our existing NrOS prototype and described in Section 5. This design is similar to IronFleet [20] and VeriBetrKV [18].

Beyond isolation: OS verification as a foundation for correct applications HotOS ’23, June 22–24, 2023, Providence, RI, USA
We propose that the design of the NrOS operating system [6] to provide such an application interface as a foundation to verify client programs.

4.1 NrOS design

NrOS is an open-source multikernel OS, written in Rust, that can run many POSIX applications. NrOS was constructed primarily with sequential logic and sequential data structures, which are scaled across cores and nodes using node replication ("NR") [9], a log-based shared-memory synchronization mechanism inspired by state machine replication in distributed systems. While this approach had the intention of simplifying the kernel development and complexity, this design can also simplify the formal verification, especially when reasoning about concurrency. IronSync [19] verifies the node replication algorithm, which is a key part of our verification methodology (Section 4.3).

NR replicates sequential code and its data structures on each NUMA node and maintains consistency through an operation log. It achieves read-concurrency with a readers-writer lock and write-concurrency through flat combining, which batches operations from multiple threads and logs them atomically. To scale writes further, NrOS shards kernel state into multiple NR instances and replicates them over independent logs, allowing for scalability to many cores.

The NrOS kernel provides the following main services: memory and device management, processes, scheduling, and a file system. In user space, NrOS provides a user-level thread scheduler with synchronization primitives and a memory allocator. Further, programs can link against rump [22], a NetBSD-based library operating system, which provides OS components so that programs can use the POSIX API.

The BSD components are written in C and are unlikely to be verifiable in their current form. However, they provide an incremental path during the transition to fully verified components as they can be replaced one by one with verified components. A similar approach to constructing concurrent, scalable subsystems as taken in the kernel with NR may be applicable to many of the user-space components such as the network stack, user-level thread scheduler, etc.

4.2 Verus

The verification tool we choose, Verus [24], is an open-source integrated verification language based on the Rust programming language with semi-automated reasoning using an SMT solver.

Verus is designed to support efficient verification of low-level sequential code. Recent work has demonstrated that similar verification languages can scale up to software of thousands of lines of implementation code with a manageable proof burden [18]. Verus leverages the ownership type system in the Rust programming language to reason about
the program’s memory and potential aliasing. This is known to improve the developer experience by reducing proof burden and verification times [25].

In our experience so far, Verus has been effective in reasoning about sequential logic and sequential data structures, which constitute the primary building blocks of NrOS. In addition, Verus supports distributed system-state-machine reasoning, in the style of IronFleet [20], which we plan to use to model the high-level interaction between the kernel replicas, the hardware, and the client application interface. Verus also supports shared-memory concurrency reasoning, borrowing ideas from IronSync [19]. Concurrency reasoning is necessary to verify the correctness of the node replication scheme across replicas and the lock-based operations within each replica shared by multiple concurrent cores.

Crucially, Verus is designed to verify low-level systems software. We have initial evidence (Section 5) that it is an effective tool for a modern OS verification project, thanks to its ability to directly reason about imperative systems code that interacts with the hardware, thanks to Rust’s ability to directly interact with hardware at a low level, and thanks to Rust’s safety properties [29], which Verus inherits.

4.3 Verification methodology
NrOS relies on node replication to implement scalable data structures and consequently the correctness of the system depends on the correctness of the node replication algorithm.

IronSync [19] verified the node replication algorithm in Dafny, showing that a sequential data structure replicated with NR remains linearizable. We just completed porting the implementation from the combination of the implementation and the relevant hardware (e.g., memory accesses translated by the MMU), (ii) is effective at proving the correctness of low-level systems code that interacts with the hardware, and (iii) results in code that has comparable performance to NrOS’ unverified implementation.

4.4 The refinement theorem
The theorem we need to prove is that the high-level spec described in Section 3 is refined by a model of the hardware execution, which includes the operating-system code and all the physical state of the CPU and devices. Refinement says that for every behavior of the hardware execution there exists a corresponding execution of the abstract model with the same behavior. In this case the behavior we want to preserve is the return values of instructions, including reading from memory and system calls.

5 THE PAGE TABLE PROTOTYPE
We have used our proposed approach to implement and verify page table management code for x86-64. This provides initial evidence that our approach (i) can faithfully model the relevant hardware (e.g., memory accesses translated by the MMU), (ii) is effective at proving the correctness of low-level systems code that interacts with the hardware, and (iii) results in code that has comparable performance to NrOS’ unverified implementation.

Figure 2 shows the structure of our prototype implementation. Our code includes functions to map and unmap frames in a page table and a function to resolve a virtual address (if it is mapped). To verify the code, we developed a hardware spec (1) that describes the hardware environment, and a high-level spec (2) that describes the behavior of map, unmap, and resolve. We prove that the page table implementation (3) refines the high-level spec, if it is run in the intended hardware environment.

High-level spec. This spec defines the behavior of a correct implementation from a client application’s perspective as described in Section 3. Namely, accessing memory and how an application’s view of its virtual memory expands when mapping memory and shrinks when unmapping it. The high-level spec is a state machine with transitions for memory reads and writes as well as map, unmap, and resolve. The spec describes the page table as a mathematical map from virtual addresses to page table entries storing the physical address and permission bits.

Hardware spec. The hardware spec describes the intended runtime environment of the implementation. In our prototype this is a single-core x86-64 processor and includes a description of how the MMU translates memory addresses by interpreting the page table bits in memory, i.e., walking the page table, or using cached translations from the TLB.

Implementation. We implement executable, concrete functions in Rust for the map, unmap, and resolve operations. Those functions read and write memory locations of the page table to perform mapping or unmapping of frames, as well as allocate or free memory used to store the page table.

Correctness Proof. We prove that the resulting state machine from the combination of the implementation and the hardware spec refines the high-level spec. In other words, given the MMU’s interpretation function of the page table in
memory, the implemented map, unmap and resolve functions have the same behavior as their counterparts in the abstract high-level spec representing the client application interface. The implementation is then functionally correct. This correspondence represents the lion’s share of the proof effort, as it requires us to map from a multi-level tree structure encoded as bits to a flat abstract data type, i.e. the logical map from virtual addresses to page table entries.

Evaluation. We hypothesize that our tools of choice can achieve a good enough verification experience, characterized by an acceptably low proof burden and short iteration time, and that the verified code matches the performance of unverified NrOS. To evaluate this hypothesis, we measure the proof-to-code ratio for the page table code and the total time to verify the code.

Our results show that the proof-to-code ratio is 10:1. This metric is only an indication and it is hard to compare across verification efforts with even small differences between their theorems. However, this number is in line with or better than the verification burden reported by other OS verification projects: The approximate ratios for SeL4 and CertiKOS are 19:1 and 20:1, respectively. SeKVM for weak memory has a ratio of approximately 10:1, though this does not include the framework that was developed for that project. Verve — by verifying less extensive properties — achieves a ratio of 3:1.

Extending verification to our proposed scope may be unrealistic with a ratio of 10:1. In particular, even small libc implementations such as musl comprise tens of thousands lines of code. However, we expect that verifying library code can be done with significantly lower proof effort. A considerable portion of our page table proofs is dedicated to the layered refinement structure that will not be necessary for the relatively simpler properties of many library functions. The SeKVM authors explicitly highlight this proof step as the most complex refinement proof in SeKVM [26].

The total time to verify our code is approximately 40 seconds. Figure 1a shows further that all functions are individually verified in at most 11 seconds. These verification times compare favorably with previous semi-automated approaches [18] and demonstrate that the tooling and methodology enable sufficiently rapid development iteration times.

To compare the performance of the verified page table implementation to the unverified implementation in NrOS, we measure the latency of repeatedly executing system calls to map (Figure 1b) frames and unmap a frame (Figure 1c) in the address space of the benchmark process. Our results show that the verified implementation can closely match the performance of the unverified implementation.

6 CONCLUDING REMARKS

Verifying a complete OS is a huge undertaking and we do not believe the scope we are proposing is feasible for just one research group. Therefore, we hope to inspire the community to work together towards this vision. To make this possible, we would first need to understand how to define interfaces between verification efforts that use different tooling while still connecting the final artifacts.

There are two specific research challenges within the scope of a verified operating system we think are interesting. First, to our knowledge, there is no prior work that verified a threading library (e.g., with the scope of pthreads) nor did we find a verified high-performance network stack. These artifacts in themselves would be interesting to pursue. Second, can we as a community define a common standard for verifying libraries and components that can be shared and linked as easily as we do for implementations?
REFERENCES


