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:
Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. 2023. Beyond isolation: OS verification as a foundation for correct applications. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595899

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

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595899

	seL4	Verve	Hyperkernel	CertiKOS	seKVM+VRM
Kernel memory safety	✓	✓	✓	✓	✓
Specification refinement	✓	✓	✓	✓	✓
Security properties	✓	✗	✓	(✓)	✓
Multi-processor support	✗	✗	✗	✓	✓
Process-centric spec	✗	✗	✗	✗	✗

Table 1: Comparison of OS verification projects

	seL4	Verve	Hyperkernel	CertiKOS	seKVM+VRM
Scheduler	✓	✓	✓	✓	✓
Memory management	✓	✓	✓	✓	✓
Filesystem	✗	✗	(✓)	✗	✗
Complex drivers	✗	✓	✗	✗	✓
Process management	✓	✗	✓	✓	✓
Threads and synchronization	✗	✓	✗	✓	✗
Network stack	✗	✗	✗	✗	✗
System libraries	✗	✗	✗	✗	✗

Table 2: Verified OS components

the verification of these client applications. We present our model in Section 3. Once we have defined correctness, we need to carefully select the OS design, the verification tool and the verification methodology.

The starting point for our proposed verified operating system is NrOS [6], a multi-core OS written in Rust. A key feature of NrOS is that its data structures use *node replication*, a technique that systematically turns a sequential data structure into a linearizable concurrent one. While node replication was intended to simplify kernel development, it also simplifies verification of the kernel's concurrent data structures while still getting good performance. We elaborate on NrOS and its design in Section 4.1.

We choose the Rust-based Verus [24] verification language to verify NrOS. Verus has good support for verifying low-level systems and inherits Rust's safety and ability to directly interact with the hardware. Verus also supports efficient verification techniques for complex systems and concurrency. These are a great fit for NrOS' design and make it plausible to broaden the verification scope to include the application dependencies we have listed. We introduce Verus in Section 4.2, and we describe our verification strategy in Section 4.3.

In this paper, we focus on functional correctness rather than security. Functional correctness already implies integrity, since no allowed behavior of a process can corrupt the state of an unrelated process. However, a verified OS should also guarantee some level of isolation between system components, either to allow secure multi-tenancy or as a defense-in-depth strategy to isolate and protect different components with different levels of trust. An isolation guarantee could also be verified for our proposed OS but we have not explored specifying and verifying such a property yet.

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`:

```
spec fn read_spec(pre: State, post: State, fd: usize,
  buffer: Seq<u8>, read_len: usize)
{    pre.files[fd].locked
  && read_len == min(buffer.len(), pre.files[fd].size -
     pre.files[fd].offset)
  && buffer[0 .. read_len] == pre.files[fd].contents[
     pre.files[fd].offset ..
     (pre.files[fd].offset + read_len)]
  && post.files[fd].offset == pre.files[fd].offset + read_len }
```

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.

```
pub fn read(sys: &mut Sys, fd: usize, buffer: &mut [u8]) -> (
  read_len: u64)
 requires sys.view().files[fd].locked
 ensures read_spec(old(sys).view(), sys.view(), fd, buffer.view(),
  read_len)
```

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].

It is also possible to implement and verify core "standard library" features like those in glibc and pthreads, connecting to the model of the operating system. This allows the kernel APIs to remain narrow while giving applications a higher-level programming API with an easier-to-use spec — for example, we might expose futexes [14] from the kernel and then verify a userspace mutex implementation on top.

In this design, the state machine and function specifications of the syscalls need to be connected correctly by the syscall mechanism. This entails an assumption about the hardware syscall mechanism transferring control between kernel-space and user-space while preserving relevant registers. It further entails three verification obligations: marshalling, mapping, and data-race freedom.

The marshalling obligation is guaranteeing that calling `read` results in its parameters and return values being correctly marshalled across the user- and kernel-space boundary. We can prove that values correctly round-trip through serialization and deserialization so that syscall arguments are consistent between user-space and kernel-space. For systems where some of the arguments are passed in registers, we would need to model the ABI as an assumption of the serialization library, and an unverified shim that unpacks the values from registers before transferring control to the syscall handler.

The mapping obligation is that the process memory for the `buffer` appear at a known location in kernel space. This becomes an additional verification condition for the kernel's page table mapping logic.

Finally, the data-race freedom obligation is that memory holding syscall data (e.g. the memory backing `buffer`) will not be modified or accessed by other threads while the syscall is being handled. Addressing this obligation in general is an open challenge. If the application is in Rust, its unique ownership properties can help: the mutable reference to `buffer` is guaranteed to be unique by the type system.

The atomic high-level spec in the example is unrealistic because it implies that the system state only changes in response to the current threads' actions. How to address this is an open question: we need to be able to "split up" this state so that each thread is only concerned with the fragment it needs to interact with. A separation-logic based approach leveraging Verus's linear ghost types, inspired by Iris [21] and IronSync [19], may allow composing the reasoning for the various threads and processes in the system.

4 VERIFIED NROS

We propose that the design of the NrOS operating system [6] and the Verus [24] semi-automated verification tool for Rust, combined with the IronSync [19] concurrency-reasoning methodology, are a promising choice to build an OS proven 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 proofs to Verus and Rust.

This lets us take verified sequential data structures, such as those backing the paging or scheduler logic of the OS, and have atomic access to them from concurrent kernel replicas at the system's cores. We can then compose the atomic state machines for the replicated data structures with the state machine representing each kernel thread's logic to prove that the kernel supports the abstract client application interface as described in Section 3.

The replication of sequential data structures using NR as a common concurrency mechanism is a design element of NrOS that both provides good multi-core scalability and is a great fit for our verification methodology. Instead of needing to reason about fine-grained concurrency for all interacting OS components, we can verify NR once and reason about their linearizable interface once they have been made scalable with NR.

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.

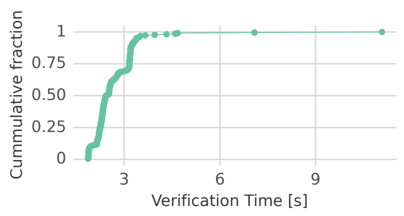
Figure 2 shows the structure of our prototype implementation. Our code includes functions to map and unmap frames in a page table and a function to resolve a virtual address (if it is mapped). To verify the code, we developed a *hardware spec (1)* that describes the hardware environment, and a *high-level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

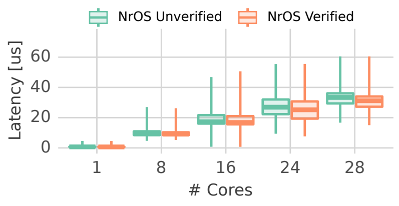
Hardware spec. The hardware spec describes the intended runtime environment of the implementation. In our prototype this is a single-core x86-64 processor and includes a description of how the MMU translates memory addresses by interpreting the page table bits in memory, i.e., walking the page table,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

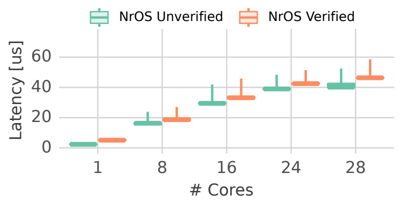
Correctness Proof. We prove that the resulting state machine from the combination of the implementation and the hardware spec refines the high-level spec. In other words, given the MMU's interpretation function of the page table in



(a) CDF of all 220 verification conditions.



(b) Map Latency



(c) Unmap Latency

Figure 1: Verification times and map/unmap latencies of the verified page table implementation

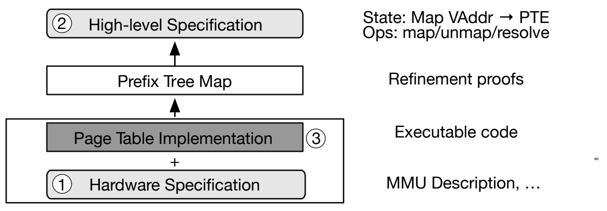


Figure 2: Proof structure of the page table prototype

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

[1] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. Cogent: Verifying high-assurance file system implementations. *SIGPLAN Not. 51*, 4 (mar 2016), 175–188.

[2] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books, 2018.

[3] Athalye, A., Belay, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)* (Hunstville, ON, Canada, Oct. 2019).

[4] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 29–44.

[5] Beringer, L., Petcher, A., Ye, K. Q., and Appel, A. W. Verified correctness and security of openssl hmac. In *Proceedings of the 24th USENIX Conference on Security Symposium* (USA, 2015), SEC'15, USENIX Association, p. 207–221.

[6] Bhardwaj, A., Kulkarni, C., Achermann, R., Calciu, I., Kashyap, S., Stutsman, R., Tai, A., and Zellweger, G. Nros: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (2021), A. D. Brown and J. R. Lorch, Eds., USENIX Association, pp. 295–312.

[7] Bond, B., Hawblitzel, C., Kapritsos, M., Leino, R., Lorch, J., Parno, B., Rane, A., Setty, S., and Thompson, L. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium* (August 2017), USENIX.

[8] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Geffen, J. V., and Warfield, A. Using lightweight formal methods to validate a key-value storage node in amazon S3. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 836–850.

[9] Calciu, I., Sen, S., Balakrishnan, M., and Aguilera, M. K. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[10] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F., and Zeldovich, N. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)* (July 2022).

[11] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 18–37.

[12] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst. 32*, 4 (jan 2015).

[13] Costanzo, D., Shao, Z., and Gu, R. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (2016), C. Krintz and E. D. Berger, Eds., ACM, pp. 648–664.

[14] Drepper, U. Futexes are tricky. *Futexes are Tricky, Red Hat Inc, Japan 4* (2005).

[15] Fonseca, P., Zhang, K., Wang, X., and Krishnamurthy, A. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (2017), G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, pp. 328–343.

[16] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, Association for Computing Machinery, p. 29–43.

[17] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 653–669.

[18] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., and Parno, B. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 99–115.

[19] Hance, T., Zhou, Y., Lattuada, A., Achermann, R., Conway, A., Stutsman, R., Zellweger, G., Hawblitzel, C., Howell, J., and Parno, B. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) [To Appear]* (July 2023).

[20] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 1–17.

[21] Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., and Dreyer, D. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (2015), S. K. Rajamani and D. Walker, Eds., ACM, pp. 637–650.

[22] Kantee, A. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels.* PhD thesis, Aalto University, 2012.

[23] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[24] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang. 7*, OOPSLA1 (2023), 286–315.

[25] Li, J., Lattuada, A., Zhou, Y., Cameron, J., Howell, J., Parno, B., and Hawblitzel, C. Linear types for large-scale systems verification. *Proc. ACM Program. Lang. 6*, OOPSLA1 (2022), 1–28.

[26] Li, S., Li, X., Gu, R., Nieh, J., and Hui, J. Z. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (2021), M. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 3953–3970.

[27] Li, S., Li, X., Gu, R., Nieh, J., and Hui, J. Z. A secure and formally

verified Linux KVM hypervisor. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021* (2021), IEEE, pp. 1782–1799.

[28] Lorch, J. R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J. R., and Zhao, X. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 197–210.

[29] Matsakis, N. D., and Klock, II, F. S. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (New York, NY, USA, 2014), HILT '14, ACM, pp. 103–104.

[30] Murray, T. C., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., and Klein, G. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), IEEE Computer Society, pp. 415–429.

[31] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling symbolic evaluation for automated verification of systems code with serval. In *SOSP* (2019), ACM, pp. 225–242.

[32] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017* (2017), ACM, pp. 252–269.

[33] Nikita Koval, Dmitry Khalanskiy, and Dan Alistarh. A formally-verified framework for fair synchronization in kotlin coroutines. *CoRR* (2021).

[34] Pirelli, S., Valentukonytė, A., Argyraki, K., and Candea, G. Automated verification of network function binaries. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 585–600.

[35] Tao, R., Yao, J., Li, X., Li, S., Nieh, J., and Gu, R. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 866–881.

[36] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (apr 2009), 76–85.

[37] Yang, J., and Hawblitzel, C. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010* (2010), B. G. Zorn and A. Aiken, Eds., ACM, pp. 99–110.