Feature

seL4: Operating Systems With the Reliability of Mathematics

Gernot Heiser
UNSW Sydney
Sydney, NSW 2052, Australia

A COMPUTER SYSTEM can only be as reliable as its operating system (OS), since the OS controls the system's resources and is responsible for enforcing security. If the OS fails, be it by a random crash or as the result of a targeted attack, the whole system fails.

Mainstream OSes consist of tens of millions of lines of code [1]. Given the software-engineering rule-of-thumb estimates 1–3 bugs per 1,000 lines of code [2], and even (optimistically) assuming that code quality has improved in the past 20 years, this implies that these systems have literally thousands of faults, and it is therefore unsurprising that they fail frequently: MITRE's vulnerability list shows about 10 critical exploits per year for Linux [3].

Safety-critical systems, where reliability is paramount, generally use OSes based on a *microkernel*: the OS "kernel," the part of the OS that executes in the privileged mode of the hardware, is reduced to a minimum (typically tens of thousands of source lines of code, kSLOC), and most OS functionality is implemented in processes executing in unprivileged mode (aka. user mode), running side-by-side with application functionality, as shown in Figure 1.

Such an approach limits the damage caused by faults or exploits, as the kernel isolates user-mode processes from each other—OS services and applications alike. In other words, the microkernel approach dramatically reduces the attack surface, although without eliminating it completely, as demonstrated by critical vulnerabilities of commercial microkernel-based OSes QNX, INTEGRITY, and Lynx [4].

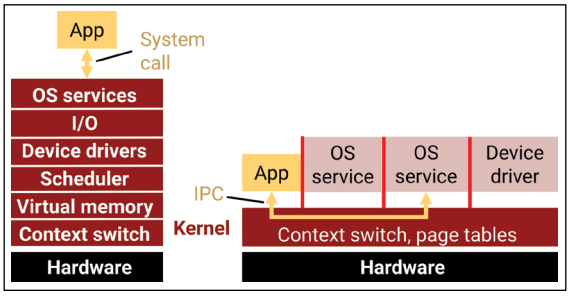

Figure 1. Monolithic versus microkernel structure. The typical code size of a monolithic kernel is three orders of magnitude larger than that of a microkernel.

However, the small size of a well-designed microkernel allows going a (big) step further: *formally verifying* the correctness of the kernel's implementation, that is, developing a mathematical proof that the implementation adheres to a specification of the kernel's functionality. The first kernel so verified is the *seL4 microkernel* [5].

Digital Object Identifier 10.1109/MRL.2026.3680074
date of current version: XX XXX XXXX.

While seL4 has been extensively covered in the academic literature, in this article, we will examine it specifically from the reliability angle, as well as look at the practicalities of deploying seL4 and the seL4-based LionsOS system that greatly eases adoption. But first, we will look at what seL4 is (and is not), and what its formal verification achieves.

seL4 microkernel

seL4 comprises about 10–12 kSLOC, making it one of the smallest microkernels that provide sufficient functionality to build arbitrary systems on top. At the same time, seL4 sets the benchmark for microkernel performance [6], important for real-world use. It is also open source, together with all proofs, and supported by an open-source community that keeps contributing functionality on top of seL4 as well as tools that ease its use.

The small size of the kernel was made possible by strict adherence to the *microkernel minimality principle* [7], which states that *the kernel should include no functionality that can be implemented in user mode*. So, instead of the services normally expected from an OS, seL4 only provides minimal, *policy-free* mechanisms for controlling hardware resources: a secure *protected procedure call* (PPC) mechanism (to invoke services provided by a server process), a semaphore-like synchronization mechanism, an address-space abstraction that is just a thin wrapper around hardware-defined page tables, an execution abstraction (threads), and a mechanism for providing bounded access to execution time for a thread, plus primitives for handling hardware exceptions and interrupts [8].

This small size is crucial for verification, as verification effort grows quadratically with the size of the specification [9], a strong incentive to keep things small and simple. Modularity can mitigate some of this quadratic growth: all other things being equal, verifying a module with specification size of n will have complexity $O(n^2)$, while breaking this into k equal-sized modules could theoretically reduce it to $O(k(n/k)^2)$ or $1/k$ of the single-module cost. This is, of course, highly optimistic, as the total size of the specification of the k-module system would almost certainly be larger than the single module. More importantly, a truly minimal microkernel, such as seL4, does not have much internal modularity as its functions are highly interdependent.

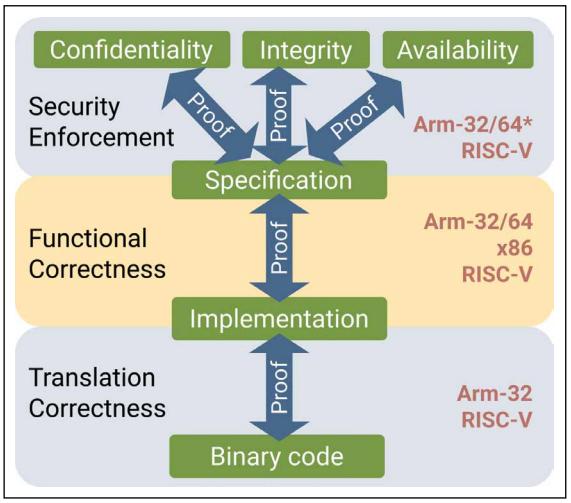

Figure 2. seL4 proofs across architectures. *The confidentiality proof for Arm-64 is expected to be completed by mid-2026.

Figure 2 shows the various proofs about seL4. The core is the proof of functional correctness: The kernel's C code implements its specification. This represents a very strong notion of freedom from implementation bugs (subject to a number of assumptions that include correctness of the small amount of assembly code, correctness of the boot code, and correct understanding of the operation of the hardware) [10]. It is a very powerful property, as it allows reasoning about the kernel by only looking at the specification, not the implementation.

But functional correctness does not guarantee that the specification is a "good" one, that is, it has the right properties. While it is never possible to prove that a specification has all the properties one expects from it—this "expectation" is inherently informal and thus not formally verifiable—it is possible to get closer by proving specific properties about the specification. This was done by proving that seL4 can enforce (formal specifications of) the "CIA triad": confidentiality, integrity, and availability [11]. These proofs compose with functional correctness, meaning that they apply to the kernel's implementation.

Having a correct C implementation still requires trusting the C compiler, which is likely buggy (we use GCC, itself 100s of kSLOC), and there is no guarantee that the compiler assumes the same semantics of the C language as the proofs—a real problem as the C semantics are ambiguous. The seL4 proofs,

therefore, take the compiler out of the trusted computing base (TCB): a separate *translation validation toolchain* proves that the binary code generated by the compiler has the same semantics as what the functional correctness proof assumes about the C code [11]. This guarantees that the executable binary behaves according to the specification.

In addition, seL4 has undergone a sound and complete analysis of its *worst case execution time* (WCET) [12], making it the only protected-mode OS kernel that is known to have such a sound analysis of its timing behavior. Originally performed for (now mostly obsolete) Armv6 processors, and having become stale because Arm discontinued publishing instruction latencies, we have recently re-established the analysis for open-source, 64-bit RISC-V cores.

Note, however, that the seL4 verification currently only applies to a single-core configuration. Verification of a multikernel [13] configuration is in progress. Also, like any OS, verified or not, seL4 is at the mercy of the underlying hardware. While both Arm and RISCV ISA specifications are now formalized, and seL4's translation validation links to these formal ISA specs, there is no guarantee that the processor implementations are correct with respect to the ISA spec. If the hardware is buggy or can be made to misbehave by physical manipulation, including irradiation, overclocking, or over-exercising as in Rowhammer, all bets are off. Also, the ISA (intentionally) abstracts over timing, meaning that the verification cannot preclude information leakage through timing channels, but work on addressing this is in progress [14].

seL4's verification mostly used *interactive theorem proving* (ITP), a labor-intensive, mostly manual process. The effort for the initial proof of functional correctness was 11 person-years (not counting investment in reusable frameworks) and consisted of 200,000 lines of proof script to verify 8,500 lines of the C code [11]; by now, the proof base has grown to well over a million lines. This effort, while not higher than that of traditionally engineered high-assurance code (which provides *no guarantees*), can only be justified for such a highly reusable artifact.

How seL4 helps reliability

seL4's formal verification takes the kernel out of the TCB: You do not have to *trust* what is proved, you only have to trust mathematics (besides the proof assumptions and the underlying hardware).

Among others, the verification implies that seL4 cannot be affected by any of the usual OS vulnerabilities: There can be no stack overflows, no control-flow attacks (such as "return-oriented programming"), no arrays accessed out of bounds, no integrity violations of kernel data, no kernel exceptions, crashes or other undefined behavior—the kernel is a rock-solid foundation for the system built on top.

In addition, seL4 uses *capabilities* [17] for access control. This means that access is determined per-process and at object granularity (e.g., individual page frames and individual communication channels between processes), which supports fine-grained tracking of information flow (see Figure 3).

An *object capability* or short *capability* is an unforgeable token that provides *prima facie* evidence of the right to access an object. The capability thus implies two things: a reference to an object (it is an opaque pointer) and a (possibly empty) set of rights on that object—think of it as a key to the object.

In a capability system, the only way of performing an operation on an object is by invoking the capability. For an operating system that usually means invoking a system call with the capability as the first argument, with further arguments specifying the desired operation, and any parameters for the operation. Capabilities present a fine-grained, per-object (e.g. file, page) protection model.

This is in contrast to the widely used access-control lists[15] used in all mainstream operating systems, where objects list the active agents (typically represented as user and group IDs) that can access the object—implying that each process with the respective ID has the same rights.

Capabilities are usually made un-forgeable by making them kernel objects, meaning that user code can only refer to them indirectly—similar to file descriptors in Unix. This means that operations on capabilities themselves, such as creating copies or handing capabilities to other processes, require a system call. The alternative approach is to protect capabilities by hardware, using tagged memory; CHERI capabilities[16] are a recent example. Such capabilities can be copied like normal data, simplifying distribution and delegation, at the cost of revocation being hard (requiring a memory scan).

Note that Linux "capabilities" are not object capabilities: they only limit the system calls a process is allowed to invoke, do not provide fine-grained access control and do not support delegation.

Figure 3. Explainer: capabilities.

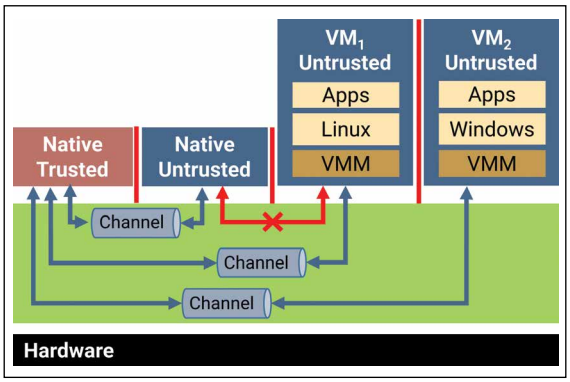

Figure 4. Fine-grained access control with capabilities.

This is a critical enabler of seL4's proof of confidentiality enforcement (Figure 2).

Capability-based access control also means that seL4 does not use ambient authority and is therefore not prone to *confused deputy* attacks [18] that inherently affect OSes using access-control lists (ACLs), where a privileged service can be tricked into violating system integrity. All mainstream OSes, all other verified kernels, and all commercial embedded OSes (with the sole exception of the L4Re microkernel, formerly called Fiasco [19], and the OS built on it) use ACLs and are therefore vulnerable.

Figure 4 gives an example. The untrusted components (shown in blue) have channels to communicate with the trusted component (in red color), but no others. Any communication between untrusted components is therefore only possible via the trusted component (which could be acting as a firewall).

The figure also shows the use of virtual machines (VMs) with seL4; these look like native components to the rest of the system. Typically, VMs are used to support legacy software that is tied to another OS, such as Linux. Different VMs are isolated from each other as much as native components are—the virtualization functionality is provided by a per-VM *virtual machine monitor* (VMM). The VMM, therefore, cannot break isolation and is not part of the system's TCB [20].

Memory management

Another unique aspect of seL4 is its policy-free approach to resource management, specifically physical memory. Other than allocating memory for its own global static data and its strictly bounded stack, the kernel does not manage memory at all.

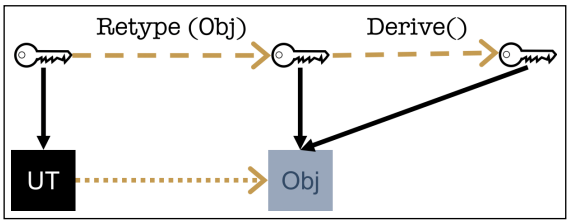

Figure 5. Retyping Untyped (UT) memory to a kernel object type (Obj) and deriving capabilities.

Instead, it hands (capabilities to) all free memory to the initial user process (called the "Init Task"), which is then in charge of all dynamic memory management; this is why it is also referred to as the *global resource manager*.

Free memory is called *Untyped*, and to put it to any use, it has to be *retyped* into a specific object type known to the kernel. Figure 5 shows the process: Invoking `Retype()` on an Untyped cap (i.e., a cap referencing Untyped memory) will create a new cap, referring to the same memory, but that memory now has the specified object type. From the new cap, another cap (with the same or lesser rights) can be derived. Derivation allows controlled delegation of access rights by handing a derived cap to another entity.

seL4 capabilities are revocable: a capability can be invalidated by its owner at any time, meaning that all the rights conveyed by it are revoked; if the cap was delegated, this revokes the delegation. Invalidating the original object cap (resulting from retyping) will also revert the memory to its original Untyped state.

Retyping is the mechanism for providing memory to the kernel. For example, creating an address space for a user process requires page tables; these are created by retyping free memory into *page-table objects*. Similarly, creating a thread requires a thread control block in the kernel to hold the thread state; this is achieved by retyping memory into a *thread control object*. Other object types are for representing communication channels, semaphores, scheduling budgets, architecture-specific state for interrupt handling, and for storing capabilities themselves. Finally, there are *frame* objects; these are the only objects used for storing user rather than kernel data. A frame cap can be inserted into a page-table object, which establishes a mapping from a virtual memory page to the physical memory referenced by the cap.

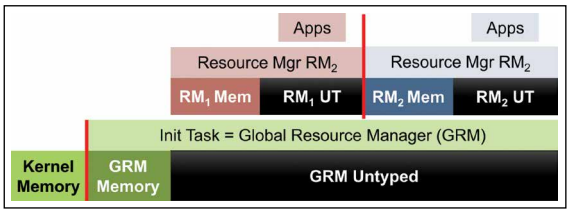

Figure 6. User-level memory management and delegation.

The user-delegated memory management approach implies that the kernel requires no heap and can therefore not run out of memory. This is at the heart of availability-enforcement in seL4: to get the kernel to consume memory, a user-mode process has to provide that memory to the kernel explicitly, from its own resources. It is therefore impossible to mount a denial-of-service attack against another part of the system, at least for the memory resource (see below for time).

The user-level management of memory, together with the controlled delegation of privileges enabled by capabilities, makes it possible to create largely autonomous subsystems with strictly limited interactions. As shown in Figure 6, the global resource manager can partition the free memory and hand it to separate processes, making them independent resource managers that autonomously manage their part of the resources. Such subsystems are strongly isolated: they can only interact with each other if the global manager has explicitly set up shared channels. And the global manager can remove a subsystem by invalidating all capabilities delegated to it.

Finally, seL4 has support for *mixed-criticality real-time systems* (MCS), that is, systems where safety-critical hard real-time tasks co-exist with less critical ones. In many such systems, it is not feasible to simply give the critical tasks the highest priority. For example, a critical control loop may execute every 100 ms and require 20 ms of processing time. It is undesirable to have the control loop monopolize the processor for that long, as this may lead to losing interrupts or network packets. Instead, less critical tasks should be able to preempt the control loop for strictly bounded periods.

seL4 provides the mechanisms for this bounded preemption: a task can be given a strict bound on its CPU usage, for example, an Ethernet driver might be allowed to consume 2 μs of processor time every 10 μs, limiting its CPU bandwidth to 20%, which guarantees sufficient time for the control loop to meet its deadline. Together with the kernel's WCET analysis and a mechanism to charge server time to the client that is requesting the service, this allows configuring scheduling parameters to guarantee timeliness of critical tasks, that is, ensuring temporal availability and integrity (note that this MCS version of seL4 [21] is currently unverified, but its verification is expected to be completed in 2027).

Is it ready to use?

Yes! seL4 has been used to protect autonomous air vehicles for years [22], [23]; DARPA even offered a "steal this drone" at the 2021 DEF CON conference [24]—no one succeeded in compromising the seL4-based system. seL4 is meanwhile deployed in national security systems.

Civilian deployments include commercial electric cars manufactured by NIO, starting with the ONVO L60 vehicle launched in 2024 [25] and has been rolled out on various model refreshes. MEP has built seL4 into their SureVoice aircraft–ground communication system [26].

However, in practice, it is not easy to use seL4. This is a direct consequence of it being a pure microkernel: seL4 provides no application-oriented services whatsoever, no file system, no networking abstraction, not even device drivers for low-level I/O functionality. This must all be built on top, using the primitive hardware-near abstractions the kernel provides—seL4 is "the assembly language of OSes."

Furthermore, the experience of the first 10+ years of seL4 has shown that developing a good, flexible, and performant system design on seL4, even in the embedded domain, requires deep expertise that takes a long time to acquire and is therefore quite rare. Most successful deployments to date were either developed in close collaboration with seL4's creators or by companies hiring multiple graduates from UNSW. This challenge is clearly inhibiting seL4's uptake.

Fortunately, this is changing rapidly, with several OS frameworks recently becoming available. Some are in-house developments supporting wider services and products, such as NIO's SkyOS controlling their cars [25], Neutrality's Atoll hypervisor aimed at supporting secure virtual private network services [27], and MIT Lincoln Labs' Magnetite OS for space satellites [28].

Others are generally available. Kry10 OS is a commercial product for embedded and cyber–physical

systems [29]. UNSW's LionsOS is an open-source OS also for embedded and cyber–physical systems [30] that, among others, runs the seL4 Foundation's website and is being deployed in third-party products [31], while Sculpt OS is an open-source, general-purpose OS developed by Genode [32].

We will have a closer look at LionsOS because of its target domain of security- and safety-critical embedded systems, while also being open source, enabling a detailed look at its design.

LionsOS: Designed for reliability

The LionsOS design is a direct consequence of its focus on reliability. It is a strict application of the principle of *separation of concerns*, resulting in a highly modular architecture, where each module is a separate user-mode process isolated by seL4.

The design takes the *KISS principle* [33] to the extreme, by adopting *use-case specific policies*: a core duty of an OS is to manage system resources, including memory, CPU time, and network bandwidth, according to some resource policy. In an embedded system, the resource policy is dictated by the use case and does not suddenly change (except as a result of a system compromise, which LionsOS is explicitly designed to prevent).

The LionsOS approach is to hard-code these policies. Resulting from the separation of concerns, a policy can usually be implemented in a single, simple module that can be targeted to the requirements of the use case. LionsOS achieves use-case diversity by keeping these policy implementations simple and easy to rewrite. This leads to a "Lego[1]-set approach" to system construction: The designer composes the system from a set of building blocks, choosing from different, interface-compatible implementations of a policy module. The resulting system has a static architecture, where modules and their permitted interactions are defined at system configuration time. Among others, this frees the system designer from explicitly dealing with capabilities.

The benefits of this approach of "radical simplicity" become most evident when looking at device drivers, known to be the most complex and error-prone part of an OS [34]. A highly performant driver for the Ethernet controller of an i.MX8 system-on-chip for LionsOS has 569 lines of code, compared to 4,775 lines of the Linux driver for the same controller, and the LionsOS driver was written by a second-year student [35]. The `sel4.systems` website runs on LionsOS using this and other native drivers, all written from scratch in C or Rust.

While the simple driver model makes it easier to develop drivers than for other OSes, it is impractical to require all drivers to be rewritten. Therefore, LionsOS supports the reuse of unmodified Linux drivers by encapsulating them in a VM (usually one VM per driver). Obviously, such a reused driver has a large TCB and should not be used for safety-critical devices and will also come with performance overheads. However, the kernel's spatial and temporal isolation mechanisms, including using an IOMMU to restrict the device's access to physical memory, ensure that the driver VM cannot compromise the rest of the system.

Figure 7 shows a reference design of a point-of-sale system built on LionsOS. Each of the boxes inside the LionsOS box is a separate, seL4-protected module. The system has native drivers for the serial port, timer device, Ethernet, and I^2C bus, and reuses a Linux graphics driver inside a VM. It uses a network file system (NFS) for accessing the backend store via Ethernet; Ethernet is also used for remote access to the device. The business logic is implemented in Python. OS policies are mostly embedded in the *virtualizer* (Virt) modules that share a single device between multiple clients and implement traffic-shaping policies.

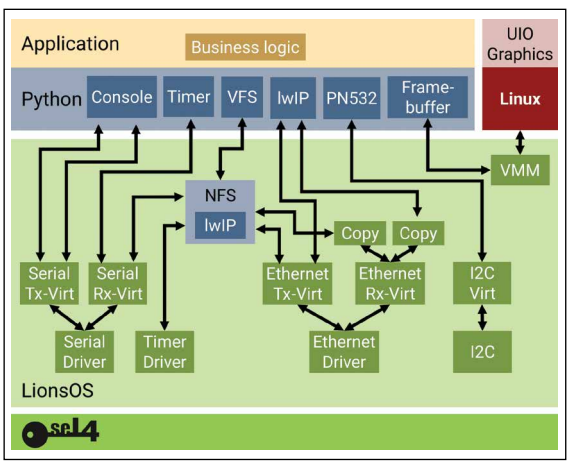

Figure 7. Architecture of a LionsOS-based point-of-sale system. Green LionsOS components are written from scratch; the NFS client and lwIP network stack are ported from other systems.

[1]Registered Trademark.

Like the drivers, most components in this system are small, with fewer than 500 lines of code each. This includes the virtualizers, which are responsible for keeping the clients of a device separated. Initial explorations show that formal verification of such small and relatively simple components is feasible using automated verification tools, given that seL4 enforces the interfaces, so components can be verified in isolation.

seL4, due to its comprehensive formal verification and open-source nature, is the most solid foundation for reliable systems, and it is already deployed in critical defense and civilian systems, including mass-produced electric cars.

Currently, verification for the widely used Arm-64 architecture does not yet cover confidentiality, but this is expected to be completed mid-year. Verification currently also does not cover the mechanisms for controlling the time resource (important for MCS) and is restricted to single-core configurations—removing those limitations is in progress [36].

Usability used to be the biggest barrier to uptake of seL4, but that is now addressed with deployment-ready seL4-based OSs, especially Kry10 OS and LionsOS. The latter is undergoing comprehensive formal verification itself, expected to be completed in 2027. ■

Acknowledgments

seL4 is maintained by the seL4 Foundation, which is supported by its members. It is furthermore supported by an open-source community that contributes functionality and frameworks that make the kernel easier to use. seL4 verification is supported by the UK's National Cyber Security Centre and DARPA under the PROVERS program contract FA 8750-24-9-1000, which also supports LionsOS development. Verification of LionsOS is performed under the PISTIs-V project, funded by the German agency *Agentur für Innovation in der Cybersicherheit GmbH* (Cyberagentur) under *the Ecosystem Formally Verifiable IT-Provable Cybersecurity* (EVIT) Program; this program also funds verification of the multikernel version of seL4.

References

[1] Steven Vaughan-Nichols. (Jul. 2025). *Linux's remarkable journey from one dev's Hobby to 40 Million Lines of Code-and Counting*. [Online]. Available: https://www.zdnet.com/article/linuxs-remarkable-journey-from-one-devs-hobby-to-40-million-lines-of-code-and-counting/

[2] P. Mohagheghi et al., "An empirical study of software reuse vs. defect-density and stability," in *Proc. Int. Conf. Softw. Eng.*, May 2004, pp. 282–291.

[3] MITRE Corporation. (2026). *Linux Kernel: Security Vulnerbilities*. [Online]. Available: https://www.cvedetails.com/vendor/33/Linux.html

[4] MITRE Corporation. (2026). *CVE Details*. [Online]. Available: https://www.cvedetails.com/vulnerability-list.php

[5] G. Klein et al., "seL4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010.

[6] Z. Mi et al., "SkyBridge: Fast and secure inter-process communication for microkernels," in *Proc. EuroSys Conf.*, Dresden, DE, USA, Mar. 2019, pp. 1–15.

[7] J. Liedtke, "On μ-kernel construction," in *Proc. ACM Symp. Operating Syst. Princ.*, Copper Mountain, CO, USA, Dec. 1995, pp. 237–250.

[8] Gernot Heiser. (May 2020). *The seL4 Microkernel—An Introduction. Sel4 Foundation Whitepaper*. [Online]. Available: https://trustworthy.systems/publications/papers/Heiser_20:sel4wp.abstract.pml

[9] D. Matichuk et al., "Empirical study towards a leading indicator for cost of formal software verification," in *Proc. Int. Conf. Softw. Eng.*, Firenze, Italy, Feb. 2015, pp. 722–732.

[10] seL4 Foundation. What the Proofs Assume. (2025). *Explanation of the Assumptions of seL4's Verification*. [Online]. Available: https://sel4.systems/Verification/assumptions.html

[11] G. Klein et al., "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[12] T. Sewell, F. Kam, and G. Heiser, "High-assurance timing analysis for a high-assurance real-time OS," *Real-Time Syst.*, vol. 53, pp. 812–853, Sep. 2017.

[13] A. Baumann et al., "The multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM Symp. Operating Syst. Princ.*, Big Sky, MT, USA, Oct. 2009, pp. 29–44.

[14] Q. Ge et al., "Time protection: The missing OS abstraction," in *Proc. EuroSys Conf.*, Dresden, Germany, Mar. 2019, pp. 1–17.

[15] B. W. Lampson, "Protection,"in *Proc. 5th Princeton Symp. Inf. Sci. Syst.*, Mar. 1971, pp. 437–443.

[16] J. Woodruff et al., "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. Int. Symp. Comput. Archit.*, 2014, pp. 457–468.

[17] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, pp. 143–155, Mar. 1966.

[18] N. Hardy, "The confused deputy (or why capabilities might have been invented)," *ACM Operating Syst. Rev.*, vol. 22, no. 4, pp. 36–38, 1988.

[19] A. Lackorzyński, and A. Warg, "Taming subsystems: capabilities as universal resource access control in L4," in *Proc. Workshop Isolation Integr. Embedded Syst.*, Nuremburg, DE, USA, Mar. 2009, pp. 25–30.

[20] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proc. EuroSys Conf.*, Paris, France, Apr. 2010, pp. 209–222. ACM.

[21] A. Lyons et al., "Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time," in *Proc. EuroSys Conf.*, Porto, Portugal, Apr. 2018, pp. 1–16.

[22] D. Cofer et al., "A formal approach to constructing secure air vehicle software," *Computer*, vol. 51, no. 11, pp. 14–23, Nov. 2018.

[23] D. Cofer et al., "Cyberassured systems engineering at scale," *IEEE Secur. Privacy*, vol. 20, no. 3, pp. 52–64, May 2022.

[24] D. Cofer et al.. (Aug. 2021). *Steal This Drone: DEF CON 29 Aerospace Village Activity*. [Online]. Available: https://loonwerks.com/publications/cofer2021defcon.html

[25] Ning Qu. (2024). *seL4 in Software-Defined Vehicles: Vision, Roadmap, and Impact at NIO, September 2024. Talk at the seL4 Summit*. [Online]. Available: https://sel4.systems/Summit/2024/abstracts2024.html#a-software-defined

[26] MEP. (Sep. 2025). *SureVoice Solid*. [Online]. Available: https://www.mep-info.com/products/surevoice-solid

[27] D. Cock, M. Mirmont, and S. L. Blond. (Sep. 2024). *The Neutrality Atoll Hypervisor and the seL4 multikernel, September 2024. Talk at the seL4 Summit*. [Online]. Available: https://sel4.systems/Summit/2024/abstracts2024.html#a-neutrality

[28] J. Furgala and S. Jero. (Sep. 2024). *Porting NASA's Core Flight System to Magnetite on seL4, September 2025. Talk at the seL4 Summit*. [Online]. Available: https://sel4.systems/Summit/2025/abstracts2025.html#a-porting-nasa

[29] (2024). *Kry10 Unlimited Inc. A Secure OS for Software Defined Machines*. [Online]. Available: https://www.kry10.com/#platform.

[30] UNSW Sydney. (Sep. 2024). *LionsOS-Fast, Secure, Adaptable!*. [Online]. Available: https://trustworthy.systems/projects/LionsOS

[31] ExploitChance. (Jan. 2026). *Announcement on LinkedIn*. [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7421806392665264128/

[32] A. Boettcher and S. Sumpf. (2025). *Sculpt OS-a Dynamic General-Purpose OS Powered by Genode on seL4, 2025. Talk at the seL4 Summit*. [Online]. Available: https://sel4.systems/Summit/2025/abstracts2025.html#a-sculpt-os

[33] Wikipedia. (2001). *KISS Principle*. [Online]. Available: https://en.wikipedia.org/wiki/KISS_principle

[34] A. Chou et al., "An empirical study of operating systems errors," in *Proc. ACM Symp. Operating Syst. Princ.*, Lake Louise, Alta, CA, USA, Oct. 2001, pp. 73–88.

[35] G. Heiser et al., "Fast, secure, adaptable: LionsOS design, implementation and performance," 2025, *arXiv:2501.06234*.

[36] seL4 Foundation. (2025). *Development Roadmap*. [Online]. Available: https://sel4.systems/Verification/assumptions.html

Gernot Heiser is a scientia (distinguished) professor and the John Lions chair of operating systems at UNSW Sydney, Sydney, Australia. His research focuses on provably secure operating systems and their use in safety- and security-critical applications. He is a Fellow of IEEE.

■ Direct questions and comments about this article to Gernot Heiser, UNSW Sydney, Sydney, NSW 2052, Australia; gernot@unsw.edu.au.