# seL4 in Australia: From Research to Real-World Trustworthy Systems

GERNOT HEISER, GERWIN KLEIN, and JUNE ANDRONICK, CSIRO's Data61 and UNSW Sydney

**ACM Reference Format:**
Gernot Heiser, Gerwin Klein, and June Andronick. 2020. seL4 in Australia: From Research to Real-World Trustworthy Systems. *Comm. ACM* 63, 4 (April 2020), 5 pages. https://doi.org/10.1145/3378426

## 1 ABSTRACT

The seL4 microkernel was the first operating system proved functionally correct to the source-code level. We provide an overview of its development since, in terms of extending verification, evolving the kernel to support wider classes of real-world systems, discuss some of the frameworks that allow extending verification guarantees to other critical system components, as well as real-world deployments.

## 2 INTRODUCTION

Ten years ago, the functional correctness proof of the seL4 microkernel marked the first time a complete operating-system (OS) kernel had been verified to the source-code level [4]. This means there was a machine-checked proof that the implementation in the C language satisfied the kernel's specification, expressed in mathematical logic.

Much has happened since then: We have extended the verification to show the kernel enforces desired security and safety properties, we have removed the need to trust the compiler, and we verified implementations for processor architectures other than the original Arm v6. We used experience from deploying seL4 in a number of real-world systems to evolve the kernel and its proofs to support a broader class of use cases, and we have made significant progress toward extending the assurance to systemwide properties. We will provide a brief overview of these developments, as well as ongoing research.

## 3 WHAT IS SEL4?

An operating system consists of a *kernel*, the software that runs in the privileged mode of the hardware, and system services that are programs running in the unprivileged hardware mode (user mode). A microkernel is the irreducible part of the kernel: It only contains code that must run privileged. The kernel is the most dangerous part of the system: if anything goes wrong here, there is nothing to protect the rest of the system. Microkernels minimize the kernel to maximize the chance of getting it right.

To make this work, the microkernel must be very general in what it provides—it must allow constructing arbitrary system functionality on top. As such, microkernels strive to be *policy free*, and provide only general mechanisms that can be used to implement arbitrary policies. For example, the microkernel does not have a notion of a security policy, and it does not provide a file system (with its implied policies of access control and management of persistent storage), it does not provide a process model. Instead it provides primitive notions of address spaces, threads of execution and low-level communication primitives (referred to as IPC).

Minimality and policy freedom combined with high performance have been the defining design principle of the L4 family of microkernels, which go back to the mid 1990s. seL4, developed 2004–2009, takes policy freedom to a new level: It does not even manage physical memory; the kernel has no heap, and userlevel managers must

Authors' address: Gernot Heiser; Gerwin Klein; June Andronick, CSIRO's Data61, UNSW Sydney.

provide the kernel with memory to store metadata, such as page tables and thread control blocks. Kernel memory thus becomes subject to memory-management policies defined at the user level. An implication is that userlevel memory partitioning extends into the kernel, making it easier to reason about isolation in an seL4-based system.

Besides that, seL4 continues the trademark performance focus of the L4 microkernel family, outperforming any other OS in terms of message-passing cost, despite its strong assurance story. This is summarized in our principle "security is no excuse for poor performance."

## 4 VERIFICATION: MORE PROPERTIES, MORE PLATFORMS

The original functional-correctness proof was between an operational model of the kernel and the C source of its implementation. It applied to 32-bit Arm v6 processors. We later reverified it for the Arm v7 architecture (a minor change). More importantly, we developed a translation validation tool chain, which used a formalization of the Arm instruction-set architecture (ISA) developed at Cambridge to prove the executable binary produced by the GCC compiler and linker have the same semantics as the C code, that is, the compilation did not introduce bugs. This took the compiler, as well as our assumptions on C semantics, out of the trusted computing base (TCB).

We also proved the kernel model had desirable isolation properties, specifically that in a suitably configured system, seL4 can enforce confidentiality, integrity and availability using capability-based access control, although the confidentiality notion does not cover timing channels. This implies that seL4 enables the construction of systems that are secure in a well-defined sense.

Furthermore, we performed a sound and complete worst-case execution time (WCET) analysis of the kernel for specific Arm v6 processors, the first published case of such an analysis for a protected-mode OS. This means it is possible to reason about the timeliness of hard real-time systems built on seL4. [5] offers a detailed description of these verification achievements.

We have since ported and verified the kernel on other architectures: The functional correctness proof for the 64-bit x86 architecture was completed in 2018, while proof of functional correctness, as well as translation correctness for 64-bit RISC-V processors, is scheduled to complete in early 2020. We verified kernel extensions for hardware-supported virtualization on Arm v7 in 2017, making seL4 a fully verified hypervisor.

## 5 EVOLVING THE KERNEL

While based on more than a decade of experience with earlier L4 kernels [3], seL4's system model differs from these in a number of ways. The most obvious is its radical separation of policy and mechanism, by delegating all spatial resource management to user level. Experience with this model led to a number of incremental changes, which resulted in re-verifying the evolved kernel.

However, the original kernel's most significant shortcoming was its handling of time, which it had inherited from its L4 ancestry—a highly simplistic and somewhat unprincipled scheduling model. We had earlier flagged this as the last outstanding significant kernel-design issue [3].

We solved this problem by introducing *scheduling contexts*, which extend capability-based access control to time [7]. Other than concurrent work at Washington University, this represents the first capability model of time that is suitable for hard real-time systems and mixed-criticality systems, and it achieves this without compromising seL4's trademark superior performance. The implementation of this revised resource-management model is presently undergoing verification, which we expect to complete in early 2020.

## 6 SECURITY BY ARCHITECTURE: BUILDING TRUSTWORTHY SYSTEMS

The seL4 mechanisms are powerful, but also low-level; this is the cost of taking policy freedom further than ever before. A simple (if not trivial) system, consisting of one process invoking another, already consists of about 50 seL4 objects that userlevel code must manage. Such a system is depicted in the gray-background box on the LHS

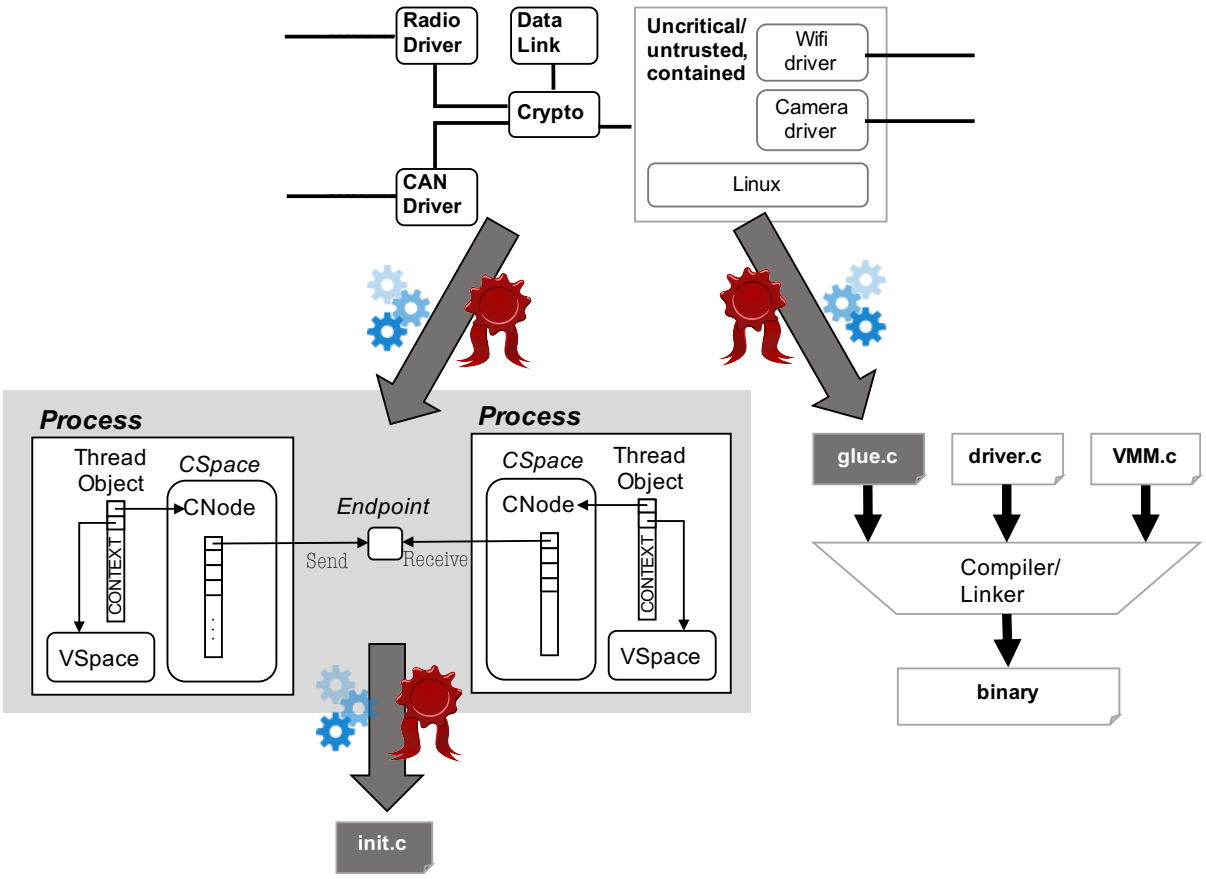


Fig. 1. Architecture enforcement. Parts in gray are generated with assurance.

of Fig 1. It shows two processes, each represented by a thread object that contains scheduling parameters and capabilities to other objects: a Cspace made up of Cnodes that store capabilities to objects the process has the right to access, and a Vspace, which describes the process's memory map (a thin abstraction over page tables). The two processes each have a capability to an Endpoint (communication port) object, with Send right for the one process, and Receive right for the other, which allows the processes to communicate (in one direction).

Building practical systems calls for a higher abstraction level (which will inevitably introduce policy and thus might be domain-dependent). Developers think in terms of architectures represented by boxes, representing data/active entities, and lines connecting them, representing communication. Such an abstraction is presented by our CAmkES component framework, whose boxes are processes with well-defined and seL4-enforced interfaces, an example is shown at the top of Fig 1. CAmkES allows presenting security policies architecturally: The security policy is observed if we can guarantee that communication can only happen where it is explicitly allowed by the architecture, that is, along lines connecting boxes.

We support this by a number of certified, automatic transformations. The developer specifies the architecture in the CAmkES architecture-description language (ADL). The first transformation (box arrow on the right) generates from the ADL the communication "glue" code, that is, the seL4 system call invocations that provide the

communication indicated by the lines in the architecture specification. The generated C code is compiled and linked together with the code implementing the functionality of the boxes to create the system image.

A second language—CapDL—describes the capability distribution in a system. We generate the low-level representation of the system by compiling the CAmkES ADL spec into CapDL (top left box arrow). The CapDL representation is the low-level representation (gray box) discussed earlier. For clarity, we only show a small part of the CapDL representation of the architecture at the top (corresponding to just two of the boxes and one connecting line).

The architecture compilation comes with a proof procedure, which ensures the CapDL spec only allows the interactions specified at the ADL level.

Finally, our CapDL tool chain generates from that representation the system initialization code that produces the respective seL4 objects and assigns capabilities (bottom left box arrow). This CapDL initializer is currently verified at the model level, with code verification under way. Together these tools guarantee the userlevel initialization code boots the system into a state that is correctly represented by the CAmkES specification, and thus allows reasoning about security properties at that architectural level [6].

CAmkES components can in principle be implemented in any programming language. Security/safety-critical, that is, *trusted*, components should be verified to ensure their trustworthiness. This is possible for C programs, as demonstrated by the verification of seL4, but easier and more cost effective in higher-level languages, especially functional languages that guarantee type and memory safety. We employ two such languages—CakeML and Cogent.

CakeML [8] is an ML dialect which comes with a certifying compiler as well as a runtime system that has been proved functionally correct. It is a fully fledged functional language suitable for implementing a wide class of applications, and its semantics supports verification of programs in the HOL4 and Isabelle proof assistants. We are using CakeML to build verified components of application code, as well as in the CapDL initializer.

Cogent is a simpler language, aimed at implementing systems code [1]. For performance predictability it is non-managed (has no garbage collector), yet the type system ensures memory safety, similar to Rust. The language is intentionally Turing-incomplete, which means it is supplemented by a library of (manually-verified) abstract data types. The Cogent compiler produces C code, together with an Isabelle embedding of the program semantics and a proof certificate that the C code has the same semantics. We have used Cogent to implement file systems, simple device drivers, as well as protocol layers.

## 7 REAL-WORLD DEPLOYMENTS

We open sourced seL4 in 2014 under the GPL v2 license. This led to the kernel being designed into a number of security- and safety-critical systems. Among them are autonomous aerial and land vehicles under the DARPA HACMS program, where together with project partners we performed an *incremental cyber retrofit*, re-architecting easily compromised real-world systems into something that resisted attacks by professional penetration testers [6].

Other uses are a secure communication device that has been security-evaluated and approved for defense use up to *secret*, and a military cross-domain device able to securely connect to networks of different classifications (which is undergoing security evaluation). A number of safety-critical systems are under development, including medical devices, autonomous passenger cars, and industrial control systems.

## 8 PRESENT ACTIVITIES

In order to scale up community engagement we are setting up an *seL4 Foundation*, similar to those for the Linux and RISC-V communities. We expect the setting-up process to be completed some time in 2020. The aims of the Foundation include a broad-based membership, commercial sponsorship, and an acceleration of the provision of

components and tools for seL4 developers. This complements our engineering efforts aimed at providing generic frameworks and infrastructure, to lower the bar of adopting seL4 for building trustworthy systems in a wide class of application domains.

Ongoing research activities include incorporating and verifying *time protection* mechanisms for principled prevention of timing channels [2]. We are also working on verifying the multicore version of seL4, a challenging task due to the fact that for performance reasons the kernel is rich in data races. Finally, we are working on our ultimate goal: Proving security or safety of a complete, real-world system. This will require a combination of all the building blocks discussed here, as well as significant further innovation in formal analysis and proof techniques.

## 9 SYSTEMS VERIFICATION AROUND THE WORLD

seL4 was the first general-purpose operating system (or any significant system component) whose implementation was proved correct. It triggered a wealth of projects on verifying systems code but is still unique in not compromising on performance and being designed for real-world use. This has just been recognized by the ACM SIGOPS Hall of Fame award.

Some other high-profile verification projects include work at MIT, on proving integrity properties of file systems in the presence of crashes, and most recently tackling some of the concurrency issues in file systems—verifying concurrent systems is extremely difficult and will take time to fully solve.

The CertiKOS kernel at Yale showed that formal OS kernel verification is possible for multicore processors in principle, at least in a simpler, more restricted hypervisor/separation kernel setting that sacrificed performance for easier verification. The main verification technique used there does not directly extend to the more complex seL4 kernel, but some of the underlying ideas can be potentially applied.

A group at the University of Washington is specializing on fully automated verification of systems code, such as file systems and simple OS kernels, and is exploring the limits of what automated verification can achieve.

Researchers at Microsoft demonstrated in the IronClad and IronFleet systems that verified OS-level properties can be lifted up to an entire distributed system and its protocols, albeit still assuming correctness of network drivers and communication. Nevertheless, this work shows our goal of systemwide properties on a verified kernel can be achieved in principle.

## REFERENCES


[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016.

[2] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM.

[3] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29, April 2016.

[4] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[5] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[6] Gerwin Klein, June Andronick, Ihor Kuz, Toby Murray, Gernot Heiser, and Matthew Fernandez. Formally verified software in the real world. *Communications of the ACM*, 61:68–77, October 2018.

[7] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

[8] Yong Kiam Tan, Magnus Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, February 2019.