From a proven correct microkernel to trustworthy large systems

June Andronick

NICTA*, UNSW

june.andronick@nicta.com.au

Abstract. The seL4 microkernel was the world's first general-purpose operating system kernel with a formal, machine-checked proof of correctness. The next big step in the challenge of building truly trustworthy systems is to provide a framework for developing secure systems on top of seL4. This paper first gives an overview of seL4's correctness proof, together with its main implications and assumptions, and then describes our approach to provide formal security guarantees for large, complex systems.

1 Introduction

The work presented here aims to tackle the general challenge of building truly trustworthy systems. The motivation is classic: software is ubiquitous and in use in systems that are more and more critical. This issue being well-accepted does not prevent the observation [4] that we routinely trust systems which again and again demonstrate their lack of trustworthiness.

The approach taken here follows the idea [10] of minimising the amount of code that need to be trusted, known as the *trusted computing base* (TCB), i.e. the part of the system that can potentially bypass security. What is added here is then to *prove* that this TCB can actually be trusted, *prove* that it is implemented in such a way that it does not bypass security. And by proving, we mean providing a formal, mathematical proof.

The first step in taking up this challenge has been to concentrate on the unavoidable part of the TCB: the operating system's core, its kernel. The kernel of a system is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The idea of minimising the TCB applied to kernels led to the concept of *microkernels*. A microkernel, as opposed to the more traditional *monolithic* design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping hardware mechanisms and needing to run in privileged mode. All OS services

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. A well-designed high-performance microkernel, such as the various representatives of the L4 microkernel family [8], consists of the order of 10 000 lines of code, making the trustworthiness problem more tractable. The L4.verified project produced, in August 2009, the world's first general-purpose microkernel whose functional correctness has been formally proved: seL4 [6]. Section 2 gives an overview of this proof, its assumptions, results and implications, and overall effort.

Given this trustworthy foundation, we are now looking at designing and building large, complex systems, for which formal guarantees can be provided about their safety, security and reliability. Our vision, together with our on-going and future work, are described in Section 3.

2 A proven correct OS kernel

The challenges in providing a "formally proven correct, general-purpose microkernel" are multiple, but all mainly come down to building a system that is both verifiable and suitable for real use. From the formal verification point of view, complexity is the enemy. From the kernel point of view, performance is the target. These two diverging objectives have been met by designing and implementing a new kernel, from scratch, with two teams working together and in parallel on formalisation and optimisations.

This kernel, called seL4, is a microkernel of the L4 family designed for practical deployment in embedded systems with high trustworthiness requirements. As a microkernel, seL4 provides a minimal number of services to applications: abstractions for virtual address spaces, threads, inter-process communication (IPC). One of seL4's key differentiators is its fine-grained access control, enforced using the hardware's memory management unit (MMU). All memory, devices, and microkernel-provided services require an associated *capability* [3], i.e. an access right, to utilise them. The set of capabilities a component possesses determines what a component can directly access.

The formal verification work aimed at proving the kernel's *functional correctness*, i.e. proving that the kernel's implementation is correct with respect to a formal specification of its expected behaviour. Formally, we are showing *refinement*: all possible behaviours of the C implementation are a subset of the behaviours of the abstract specification. For this, we use interactive, machine-assisted and machine-checked proof, namely the theorem prover Isabelle/HOL [9].

In practice, this was done is several steps, as shown in Figure 1. First, increasingly complete prototypes of the kernel were developed in the functional language Haskell. On one hand, low-level design evaluation was enabled by a realistic execution environment that is binary-compatible with the real kernel. On the other hand, the Haskell prototype could be automatically translated in the theorem prover as the formal design specification, where the refinement to the abstract specification could be started. This first refinement step represents a proof that the design is correct with respect to the specification. Since the Haskell prototype

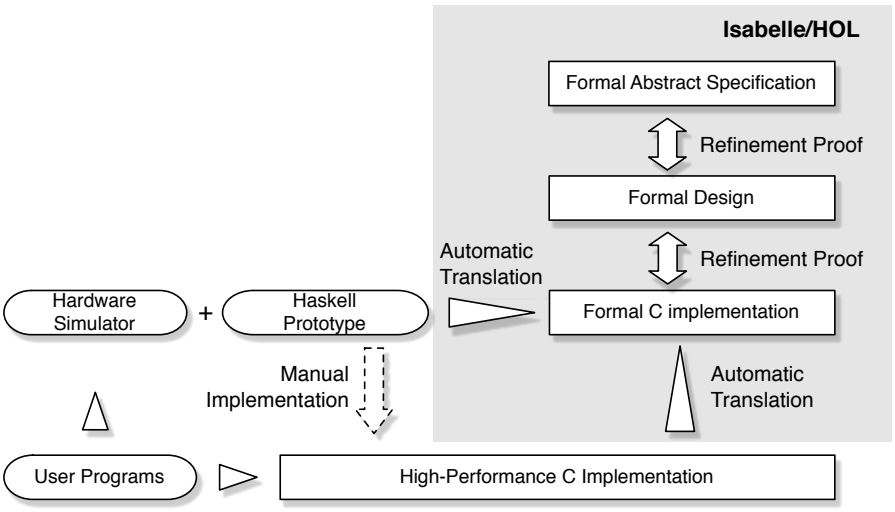

Fig. 1. The seL4 design process and refinement layers

did not satisfy our high-performance requirement, we then *manually* translated it into high-performance C code, giving opportunities for micro-optimisations. The C code was then translated into Isabelle, using a very precise and faithful formal semantics for a large subset of the C programming language [6,11,12]. A second refinement step then proved that the C code, translated in Isabelle, was correct with respect to the formal design [13].

The refinement being transitive, the two refinement steps give us a formal proof that the C implementation of seL4 refines its formal specification.

The main assumptions of the proof are correctness of the C compiler and linker, assembly code, hardware, and boot code. The verification target was the ARM11 uniprocessor version of seL4 (there is also an unverified x86 port of seL4). Under those assumptions, the functional correctness proof also gives us mathematical proof that the seL4 kernel is free of buffer overflows, NULL pointer dereferences, memory leaks, and undefined execution. The verification revealed around 460 bugs, both on the design and implementation. The total effort amounted to 2.5 person years (py) to develop the kernel and 20 py for the verification, including 9 py invested in formal language frameworks, proof tools, proof automation, theorem prover extensions and libraries. More details about the assumptions, implications and effort can be found in [6,5].

The overall key benefit of a functional correctness proof is that proofs about the C implementation of the kernel can now be reduced to proofs about the specification for properties preserved by refinement. The correspondence established by the refinement proof ensures that all Hoare logic properties of the abstract model also hold for the refined model. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties

can be), our refinement guarantees that the same property holds for the kernel source code.

3 Trustworthy, large systems

The L4.verified project has demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification. Although verifying programs with sizes approaching 10 000 lines of code is a significant improvement in what formal methods were previously able to verify with reasonable effort, it still represents a significant limit on the verification of modern software systems, frequently consisting of millions of lines of code.

Our vision to verify such large and complex systems comes from the observation [1] that not all software in a large system necessarily contributes to a given property of interest. For instance, the user interface of a medical device might represent a large amount of code, and ideally, the safe delivery of medicine should not have to rely on it. Similarly, the entertainment system implementation in a car should not have any impact on the safety of the braking system.

The idea is thus again to minimise the TCB, minimise the amount of code which the desirable property relies on, to a size where formally verifying its exact behaviour is still possible. Formally proving that the property holds for the overall system then consists in proving that it holds for the trusted components, modelled by their expected behaviours, and proving that the untrusted parts are isolated, i.e. that nothing needs to be verified about them. The key here is to use seL4's access control mechanisms to enforce this isolation between the trusted and the untrusted parts: what untrusted components can access is determined by the set of capabilities they hold. Careful choice of initial capabilities distribution can thus isolate large parts of software to exclude them from the TCB.

Our approach is to develop methodologies and tools that enable developers to systematically (*i*) isolate the software parts that are not critical to a targeted property, and prove that nothing more needs to be verified about them for the specific property; and (*ii*) formally prove that the remaining critical parts satisfy the targeted property.

More precisely, Figure 2 illustrates the different steps our approach proposes. First, the architecture of the system defines the components needed for the system, and the capabilities they need to hold. This initial capabilities distribution defines the partition between trusted and untrusted components, with respect to a desired property for the system. We have defined a capability distribution language, called capDL [7], with a formal semantics that enables us to formally describe what the exact initial distribution is expected to be.

The next step is to prove that, given this initial capability distribution and the identified partition between trusted and untrusted components, the targeted property holds on the entire system. To avoid having to reason on the complex, detailed and low-level capDL description, we first abstract the architecture description into a simpler, high level security architecture. The aim is to have

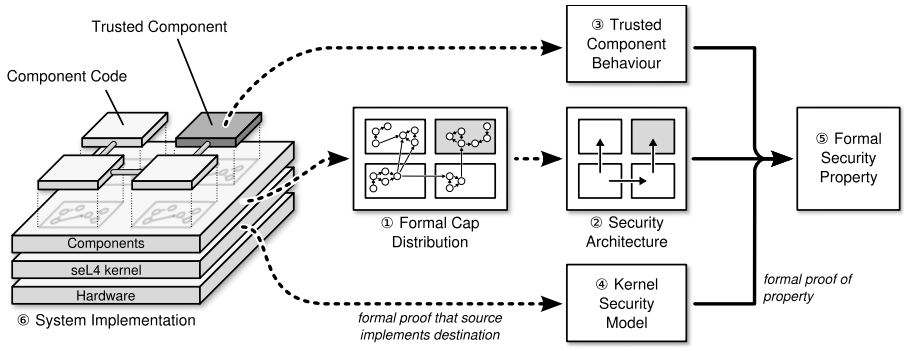

Fig. 2. Full-system verification approach for seL4-based system

the abstraction done automatically, together with a formal proof of refinement. The property is then proved at this abstract level. The trusted components' behaviour is modelled as the sequence of kernel instructions they are expected to perform. At this abstract level, the kernel instructions are described in a high level security model of the kernel. The untrusted components' behaviour is modelled as any instruction authorized by the set of capabilities they hold. The concurrent execution of all components is modelled as all possible interleavings of instructions from any component in the system.

The proof of the property implicitly validates the identified partition between trusted and untrusted components: if the proof succeeds, it means that the property indeed does not depend on the untrusted components' behaviours, and that they will be correctly implemented by any concrete program code. In some cases, the property may not be proved, revealing some issues in the design that need to be fixed.

Inspired by seL4's successful "design for verification" process, we believe that the design and implementation of the components should be done in parallel in an iterative process. Although the implementation of the untrusted components is not constrained, the proof does depend on the trusted components' behaviour. Therefore, for the property to hold not only on the abstract level but on the actual implementation, the trusted components' code has to be shown correct with respect to the expected formal behaviour used for the proof. This would follow and use the refinement approach and framework developed for seL4 verification. Similarly, we need to prove that the initial boot code leads to a state satisfying the expected formal initial capability distribution. This is ongoing work. Finally, we need to prove that the kernel's code refines its security model used to model the trusted components instructions. Building on existing seL4 refinement layers (Figure 1), this comes down to adding a layer on the top of the stack and proving that the formal abstract specification refines the security model (with additional work to prove that seL4's access control mechanism indeed ensures isolation). All of this is ongoing work.

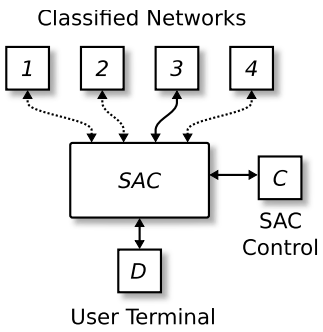

Fig. 3. The SAC routes between a user's terminal and 1 of n classified networks.

The main gain in this vision is that formal guarantees can be made for a large complex system's *implementation*, while ignoring the identified large untrusted components, leaving only the trusted components to be formally verified.

The first steps of the approach have been demonstrated on a concrete example system, namely a multilevel secure access controller (SAC) aiming to isolate networked services of different classification levels, as illustrated in Figure 3. In this case study the user only needs to access one network at a time, and selects the network through a web interface provided by the SAC on a control network interface. The property the SAC must ensure is that all data from one network is isolated from each of the other networks. While we assume that the user's terminal is trusted to not leak previously received information back to another network, we otherwise assume that all networks connected to the SAC are malicious and will collude. The SAC is representative of systems with simple requirements, but involving large, complex components, here a secure web interface, network card drivers, a TCP/IP stack for the web server, and IP routing code, any one individually consisting of tens of thousands of lines of non-trivial code.

The architecture that has been designed for the SAC is represented in Figure 4, where the user's terminal is connected to NIC-D, while the SAC is controlled through a web interface provided on NIC-C, and for simplicity of explanation, we assume that the SAC only needs to multiplex two classified networks, NIC-A and NIC-B. The system's security architecture has been designed to minimise the TCB to a single trusted component (in addition to the underlying kernel): the router manager. The router manager is the only component with simultaneous access to both NIC-A and NIC-B. The aim is that it does never use those accesses (capabilities) to access NIC-A or NIC-B, but only holds them to grant one or the other to an untrusted router component in charge of routing between one network and the user terminal. Another untrusted component, the SAC controller, provides a web interface to the control network on NIC-C. When the SAC needs

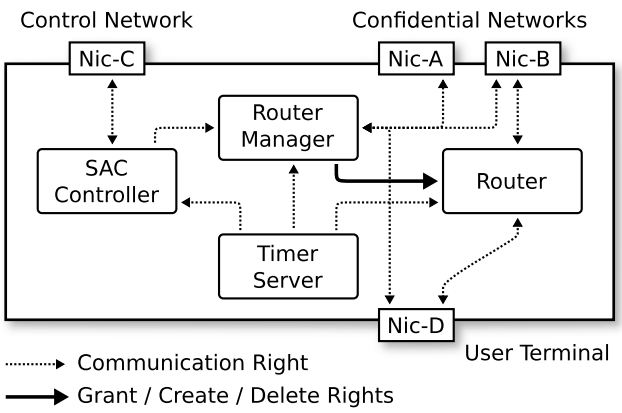

Fig. 4. High-level component breakdown of the SAC design. The router manager is the only trusted component in the system, as no other component has simultaneous access to both NIC-A and NIC-B.

to switch between networks, the SAC controller informs the router manager, which deletes the running router component and sanitises the hardware registers and buffers of NIC-D (to prevent any residual information from inadvertently being stored in it). The router manager then recreates the router, and grant it access to NIC-D and either NIC-A or NIC-B as required. This allows the router to switch between NIC-A and NIC-B without being capable of leaking data between the two.

We therefore only need to trust the router manager's implementation (approximately 1500 lines of code) not to violate the isolating security policy, but can ignore the two other large untrusted components, that we implement as Linux instances, comprising millions lines of code. At least this is what we expect, we now have to prove it.

For this case study, we first formalised the low level design in capDL, leading to a detailed description of the initial capability distribution in terms of kernel objects, as shown in Figure 5. Then we manually abstracted this design into an abstract security architecture between high level components, as the one in Figure 4. Doing this step automatically, together with a proof of refinement, is part of our future work. Finally we have formally shown that with this security architecture, information cannot flow from one back-end network to another. Details of the proof can be found in [2].

What remains to be done for this case study is to prove that (1) the router manager's code refines its formal behaviour used for the proof; (2) the booting code leads to the state illustrated in Figure 5. We also need to prove the kernel's security model refinement to the code, which in this case would also involve extending our existing functional correctness proof to the x86 version of seL4 used for the case study.

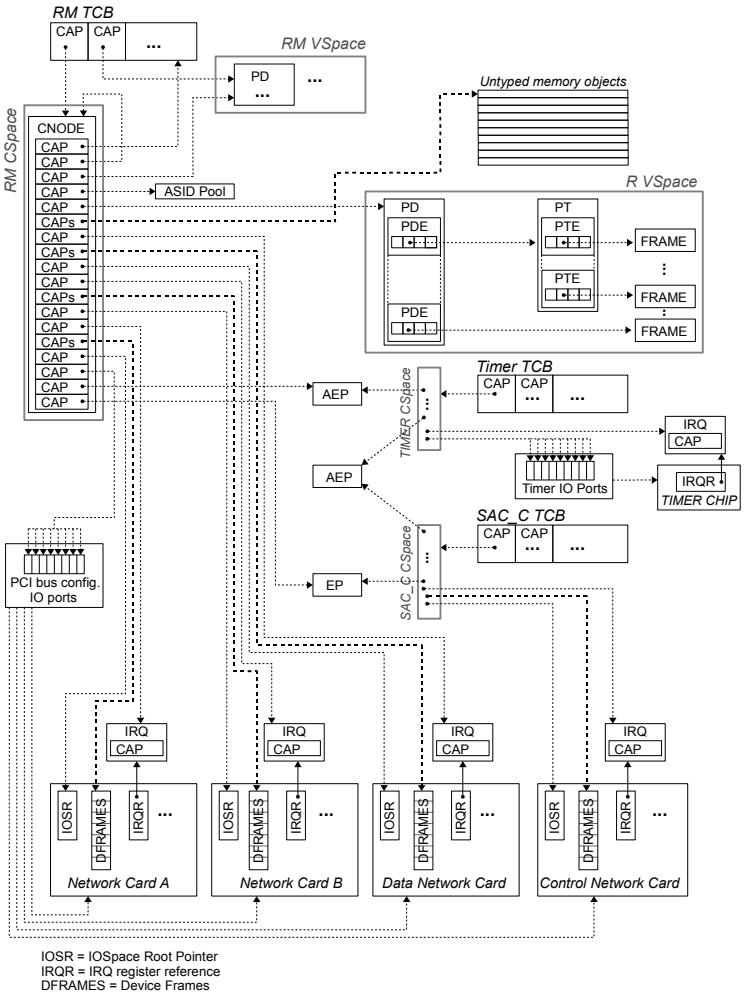

Fig. 5. Low-level Design

This case study illustrates our vision of how large software systems consisting of millions of lines of code can still have formal guarantees about certain targeted properties. This is achieved by building upon the access control guarantees provided by the verified seL4 microkernel and using it to isolate components such that their implementation need not be reasoned about.

Acknowledgments The proof of the SAC mentioned above was conducted almost entirely by David Greenaway with minor contributions from Xin Gao, Gerwin Klein, and myself. The following people have contributed to the verification and/or

design and implementation of seL4 (in alphabetical order): June Andronick, Timothy Bourke, Andrew Boyton, David Cock, Jeremy Dawson, Philip Derrin Dhammika Elkaduwe, Kevin Elphinstone, Kai Engelhardt, Gernot Heiser, Gerwin Klein, Rafal Kolanski, Jia Meng, Catherine Menon, Michael Norrish, Thomas Sewell, David Tsai, Harvey Tuch, and Simon Winwood.

References

1. J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.
2. J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.
3. J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.
4. G. Heiser, J. Andronick, K. Elphinstone, G. Klein, I. Kuz, and L. Ryzhyk. The road to trustworthy systems. In *5th WS Scalable Trusted Comput.*, Chicago, IL, USA, Oct 2010.
5. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *CACM*, 53(6):107–115, Jun 2010.
6. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
7. I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, New Delhi, India, Aug 2010. To appear.
8. J. Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.
9. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.
11. H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug 2008.
12. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007.
13. S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 500–515, Munich, Germany, Aug 2009. Springer.