A Formally Verified OS Kernel. Now What?

Gerwin Klein

[1] NICTA*, Australia

[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

gerwin.klein@nicta.com.au

Abstract. Last year, the L4.verified project produced a formal, machine-checked Isabelle/HOL proof that the C code of the seL4 OS microkernel correctly implements its abstract implementation. In my presentation I will summarise the proof together with its main implications and assumptions, I will describe in which kinds of systems this formally verified kernel can be used for gaining assurance on overall system security, and I will explore further future research directions that open up with a formally verified OS kernel.

1 A Formally Verified OS Kernel

Last year, we reported on the full formal verification of the seL4 microkernel from a high-level model down to very low-level C code [5].

To build a truly trustworthy system, one needs to start at the operating system (OS) and the most critical part of the OS is its *kernel*. The kernel 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 kernel is a mandatory part of a system's *trusted computing base* (TCB)—the part of the system that can bypass security [9]. Minimising this TCB is the core concept behind *microkernels*, an idea that goes back 40 years.

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 are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. Previous implementations of microkernels resulted in communication overheads that made them unattractive compared to monolithic kernels. Modern design and implementation techniques have managed to reduced this overhead to very competitive limits.

A microkernel makes the trustworthiness problem more tractable. A well-designed high-performance microkernel, such as the various representatives of

* 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

the L4 microkernel family, consists of the order of 10,000 lines of code (10 kloc). We have demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification.

The approach we used was interactive, machine-assisted and machine-checked proof. Specifically, we used the theorem prover Isabelle/HOL [8]. Formally, our correctness statement is classic refinement: all possible behaviours of the C implementation are already contained in the behaviours of the abstract specification. The C code of the seL4 kernel is directly and automatically translated into Isabelle/HOL. The correctness theorem connects our abstract Isabelle/HOL specification of kernel behaviour with the C code. The main assumptions of the proof are correctness of the C compiler and linker, assembly code, hardware, and boot code.

In my presentation I will give an overview of this proof and reflect on some of the lessons learned in this project. For instance, one question with any verification is: what does the specification say, what level of detail does it contain, and why?

The current abstract specification of the seL4 kernel was written with the following goals:

- abstract from data structures and implementation, aim to specify what, not how;
- have enough detail for functional correctness proofs of applications on top of seL4;
- have enough detail for users to understand results, error conditions and error codes

Especially the latter two goals mean that the abstraction level of this specification is at points lower than one might wish for a specification. For instance, to be able to show functional correctness of code on top of seL4, we do not want to use nondeterminism too liberally in the specification. If the kernel is allowed to do either A or B nondeterministically for a certain system call, then user programs will always need to handle both cases, even though in the implementation it will usually be possible (and desired) to predict which case is taken. We have therefore usually tried to include enough data in the abstract specification layer to precisely predict kernel behaviour. This draws a small number of rather arcane implementation details to a high specification level. In hindsight it is precisely these cases that users of the kernel now find hard to understand and that are candidates for future evolvement and simplification of the seL4 API. Some of these were eliminated in early API iterations (precisely to achieve an nicer specification), but others were deemed too hard or necessary at the time.

An area where nondeterminism is more reasonable is the pure indication of failure due to implementation restrictions not visible at an abstract level. In seL4 this would mean the system call either fully succeeds or has no effect and returns with an error code. This is significantly simpler to handle for the user than two different behaviours with state change. On the other hand, this means for user programs that such system calls will always need error handling code. Hence, there is still a trade-off to be made between predictive precision and elegant abstraction.

Another way to achieve more abstraction is to focus on specific aspects of the system. We have previously published different versions of a high-level access control model of seL4 [3,2]. These models abstract from message passing, memory content, and nearly all other behaviour of the system. They only record which objects have access to which capabilities. This in turn determines which system calls a thread may perform and which objects it may access. If system calls and capabilities are additionally tagged with information flow attributes, it becomes possible to reason about the authorised information flow behaviour of a system without knowing its full functional behaviour.

Another potential abstraction of the kernel may be purely its message passing behaviour, possibly ignoring or disabling shared memory. This could be used to implement abstract communication formalisms such as the pi calculus.

2 What Now?

This section gives an overview of our current work on top of the seL4 kernel.

The main, long-term aim of this work is to produce code-level formal security proofs for large systems on the order of 1 million lines of code or more. The key idea of microkernel-based designs is to drastically reduce the trusted computing base of a system. If the system architecture explicitly takes security boundaries into account, as for example in MILS (Multiple Independent Levels of Security or Safety) systems [1], and these security boundaries are adequately enforced by the underlying OS kernel, it should be possible to reduce the amount of code that needs to be formally verified to a manageable size of a few thousand or maybe only a few hundred lines of code.

For an initial case study, we have confirmed that this is possible and feasible. The case study is a small network switching device that gives a front-end terminal access to either of two back-end networks. The job of the switch is to make sure that no data travels between the back-end networks through the switch. It is Ok for data to travel through the terminal. In this case study the user and the front-end terminal are trusted to do the right thing.

A simple Linux-based switch with web interface and user authentication will at least include the Linux kernel, a web server, and a full network stack. This already brings us into the area of millions of lines of code.

With the seL4 kernel as the bottom layer and the right architecture, we managed to isolate the security critical part into one small component with less than 2,000 lines of code. The component starts and stops Linux instances that do the actual authentication and network routing/switching work. Security is achieved by using seL4's fine-grained access control mechanism to restrict the Linux instances sufficiently such that they do not need to be trusted for the information flow property above to be true. 2,000 lines of code are small enough to be formally verified, and we have already formalised and verified the access control aspect of this trusted component as well as of the entire system. This proof shows that the modelled high-level behaviour of the trusted component achieves the security property and it shows that this component is indeed the

only trusted component of the system — all other components are assumed to try anything in their power to subvert the system.

We conducted the above proof in Isabelle/HOL. It took about 1 month of work for one person relatively new to Isabelle. We also tried the same proof in the SPIN model checker. The same person, reasonably experienced in SPIN, took about half a week to achieve the same result. This is encouraging, because it means that at least the high-level problem could be solved automatically and even the interactive proof was not very time consuming. On the other hand, even for this small system, we found it necessary to simplify and change the model to avoid state explosion in the model checker. This does not necessarily mean that SPIN would not be able to solve larger systems (our security model may just not be very suitable for model checking), but it does indicate that the process of connecting this high-level model to code will not be straight-forward.

The initial Isabelle model tried to be a faithful abstraction of the access control content of the actual system with a straightforward path to code refinement for the trusted component. The SPIN model needed custom abstractions and simplifications that (we think) were justified, but that do not have a straightforward refinement path to code any more. An interesting direction for future work will be to see if such simplifications can be automated and justified automatically by Isabelle/HOL proof.

To connect this high-level model and proof to code, the following challenges are still open: we need to show that the seL4 kernel actually implements the access control model we used in the proof and we need to show that the C code of the trusted component implements the access control behaviour that we modelled.

3 What Else?

The previous section gave a broad outline of our current research agenda. Of course, there are further useful and interesting things to be done with a formally verified OS kernel. In this section, I explore two of them.

Verified Compiler

The arguably strongest assumption of our functional correctness proof is trusting the compiler to translate seL4 correctly. We are using standard gcc for the ARM platform with optimisations. While it is almost certain that there are bugs in this version of gcc, there is sufficient hope that this particular program (seL4) does not expose them, because the seL4 implementation is sticking to well-tested parts of the C language.

Nonetheless, it would be appealing to use a verified compiler on the source code of seL4, even if it is not as strongly optimising as gcc. The obvious candidate is the CompCert project by Leroy et al [6]: an optimising C compiler, formally verified in Coq, with ARM as one possible back-end. We have not yet spent serious effort trying to use this compiler on the seL4 source, but initial investigations

show that there will be a number of challenges. The main ones are: different theorem provers, different memory models, and different C language subsets. Firstly, our proof is in Isabelle/HOL, the CompCert proof is in Coq. While it should in principle be possible to transform the proof in one system into the other, in practice both proofs are very large and an automatic transformation will quickly run into technical difficulties. Secondly, while the memory models for C used in either project are morally very close, they are technically different and it is not immediately clear how the correctness statement in one translates into the other. And finally, the C language subset used in seL4 contains features that are not strictly supported by the C standard, but are nonetheless used for kernel programming. Not all of these are covered by the CompCert compiler which is targeted more at user programs. All of these can be overcome in principle, but they will require significant effort. Even if the formalisms were not adjusted to fully fit together and the connection would therefore remain informal, one would arguably still achieve a higher level of trustworthiness by running the seL4 source through a verified compiler.

A different, similarly promising approach is the one of Slind et al [7], where the compilation process happens directly in the theorem prover. For our case the approach would probably have to be fully re-implemented in Isabelle/HOL and targeted to the SIMPL intermediate language [10] that is also the target of our parser from C into Isabelle. Performing the compilation and its correctness proof fully inside the theorem prover would have the advantage that it neatly removes the translation step from C to Isabelle from the trusted computing base. The concrete C program would merely remain a convenient syntactic representation. The actual translation would take the same formal artefact in Isabelle/HOL as source that our kernel correctness proof has as a basis. Depending on how far one would drive this approach, one could add custom compiler optimisations to particular parts of the kernel and the final product could eventually be the stream of bytes that makes up the boot image that is transferred to hardware. This would reduce the trusted part to an accurate assembly level model of the target platform — something which has already been achieved in the HOL4 prover [4].

High-Level Language Runtimes

While C is still the most widely used language in embedded and OS programming, formal C verification is far for pleasant. There are various higher-level, type safe programming languages that offer themselves more readily to formal verification. The price is not necessarily just performance, but usually also a language runtime system (memory allocation, garbage collection, etc) that is larger and arguably more complex than the whole microkernel.

This should not stop us from implementing these language runtimes on top of seL4. For one, just because they are bigger and more complex, they are not necessarily harder to verify themselves. And, even if we do not verify the language runtime, we still get the benefit of one well-tested and often-reused component that could be isolated from other parts of the system using seL4 mechanisms

and that will still provide us with the productivity and reasoning benefits of a high-level language.

Two language runtimes specifically come to mind: Haskell and JVM. Haskell (or ML), because we have made good experiences in the past with verifying large programs (e.g. the seL4 kernel) that make use of a significant subset of Haskell language features and JVM, because the JVM is arguably the most well-studied and well-formalised mainstream programming languages currently in use. The JVM has existing implementations that are certified to the highest level of Common Criteria. This means it is clear in principle how a fully verified JVM implementation and runtime can be produced.

We have conducted initial successful experiments with a Haskell runtime on top of seL4. It executes normal compiled Haskell programs that do not make use of higher-level OS services (which seL4 does not provide). It also provides a direct Haskell interface to all relevant seL4 system calls. Producing a formally verified Haskell runtime still poses a significant research challenge, but seL4 offers itself as the perfect basis for it. Given the abstract seL4 specification, it is one that could be taken on by a third party.

Acknowledgements The security proof mentioned above was conducted almost entirely by David Greenaway with minor contributions from June Andronick, Xin Gao, 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. C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Paul MN, Oct 2008.
2. A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th WS Syst. Softw. Verification SSV'09*, volume 254 of *ENTCS*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.
3. D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008 — Verified Softw.: Theories, Tools & Experiments*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, Oct 2008. Springer.
4. A. Fox and M. O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *Proc. 23rd Int. Confón Interactive Theorem Proving (ITP'10)*, LNCS. Springer, 2010. To appear.
5. 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.

6. X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54, New York, NY, USA, 2006. ACM.
7. M. O. Myreen, K. Slind, and M. J. C. Gordon. Extensible proof-producing compilation. In O. de Moor and M. I. Schwartzbach, editors, *Proc. 18th International Conference on Compiler Construction (CC'09)*, volume 5501 of *LNCS*, pages 2–16. Springer, 2009.
8. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
9. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.
10. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.