OS Verification — Now!

Harvey Tuch Gerwin Klein Gernot Heiser
National ICT Australia
University of New South Wales
(harvey.tuch|gerwin.klein|gernot)@nicta.com.au

Abstract

Hard, machine-supported formal verification of software is at a turning point. Recent years have seen theorem proving tools maturing with a number of successful, real-life applications. At the same time, small high-performance OS kernels, which can drastically reduce the size of the trusted computing base, have become more popular. We argue that the combination of those two trends makes it feasible, and desirable, to formally verify production-quality operating systems — now.

1 Introduction

There is increasing pressure on providing a high degree of assurance of a computer system's security and functionality. This pressure stems from the deployment of computer systems in life- and mission-critical scenarios, and the need to protect computing and communication infrastructure against attack. This calls for end-to-end guarantees of systems functionality, from applications down to hardware.

While security certification is increasingly required at higher system levels, the operating system is generally trusted to be secure. This clearly presents a weak link in the armour, given the size and complexity of modern operating systems.

However, there is a renewed tendency towards smaller operating system kernels which could help here. This is mainly motivated by two increasingly popular scenarios:

Trusted applications and legacy software The general trend towards standard APIs and COTS technology (e.g. Linux) is even reaching safety- and security-critical embedded systems. Similarly, emerging applications on personal computers and home/mobile electronics require digital rights management and strong protection of cryptographic keys in electronic commerce. In both cases it is necessary to run large legacy systems alongside highly critical components to provide desired functionality, without the former being able to interfere with the latter. This requirement is met by de-privileging the legacy system and using a small kernel or monitor to securely switch between the trusted and untrusted subsystems, as in L4Linux, and processor manufacturers are moving towards hardware support for such partitioning (ARM TrustZone and Intel LaGrande).

Secure and efficient multiplexing of hardware This scenario partitions a system into isolated, de-privileged peer subsystems, typically several copies of the same or different full-blown operating systems. The partitioning may be based on full virtualisation (as in VMWare), or para-virtualisation, as in Xen and Denali. The underlying privileged *virtual-machine monitor* or *hypervisor* is typically of much smaller size than the operating systems running in individual partitions.

Both scenarios require an abstraction layer of software far smaller than a traditional monolithic OS kernel. For the rest of this paper we refer to this layer simply as the *kernel*, since the distinction between hypervisor, microkernel and protection-domain management software is not of relevance here.

The reduction in size, compared to traditional approaches, already goes a long way towards making the kernel more trustworthy. Standard methods for establishing the trustworthiness of software, such as testing and code review (while they inherently cannot guarantee absence of faults) work better on a smaller code base.

Recently, algorithmic techniques, like static analysis and model checking, have achieved impressive results in bug hunting in kernel software [8]. However, they cannot provide confidence in full functional correctness, nor can they give hard security guarantees.

The only real solution to establishing trustworthiness is formal verification, *proving* the implementation correct. This has, until recently, been considered an intractable proposition — the OS layer was too large and complex for poorly scaling formal methods. In this paper we argue that, owing to the combination of improvements in formal methods and the trend towards smaller kernels, full formal verification of real-life kernels is now within reach.

In the next section we give an overview of formal verification and its application to kernels. In Section 3 we examine the challenges encountered and experience gained in a pilot project that successfully applied formal verification to the L4 microkernel utilising the Isabelle theorem prover.

2 Formal verification

Formal verification is about producing strict mathematical proofs of the correctness of a system. But what does this mean? From the formal-methods point of view, it means that a formal *model* of the *system* behaves in a manner that is consistent with a formal *specification* of the *requirements*. This leaves a significant semantic gap between the formal verification and the user's view of correctness [6]. The user (e.g. application programmer) views the system as "correct" if the behaviour of its object code on the target hardware is consistent with the user's interpretation of the (usually informally specified) API. Bridging this semantic gap is called *formalisation*. This is shown schematically in Fig. 1.

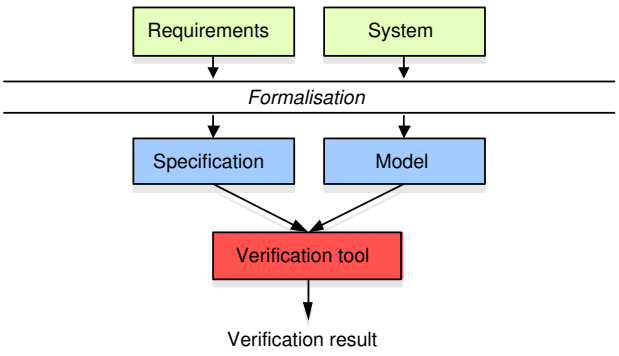

Figure 1: Formal verification process

Verification technology At present there are two main verification techniques in use: model checking and theorem proving. *Model checking* works on a model of the system that is typically reduced to what is relevant to the specific properties of interest. The model checker then exhaustively explores the model's reachable state space to determine whether the properties hold. This approach is only feasible for systems with a moderately-sized state space, which implies dramatic simplification. As a consequence, model checking is unsuitable for establishing a kernel's full compliance with its API. Instead it is typically used to establish very specific safety or liveness properties. Furthermore, the formalisation step from system to model is quite large, commonly done manually and therefore error prone. Hence, model checking usually does not give guarantees about the actual system. Model checking has been applied to the OS layer [17] and has shown utility here as a means of bug discovery in code involving concurrency. However, claims of implementation verification are disputable due to the manual abstraction step. Tools like SLAM [2] can operate directly on the kernel source code and automatically find safe approximations of system behaviour. However, they can only verify relatively simple properties, such as the correct sequencing of operations on a mutex — necessary but not sufficient for correct system behaviour.

The *theorem proving* approach involves describing the intended properties of the system and its model in a formal logic, and then deriving a mathematical proof showing that the model satisfies these properties. The size of the state space is not a problem, as mathematical proofs can deal with large or even infinite state spaces. This makes theorem proving applicable to more complex models and full functional correctness.

Contrary to model checking, theorem proving is usually not an automatic procedure, but requires human interaction. While modern theorem provers remove some of the tedium from the proof process by providing rewriting, decision procedures, automated search tactics, etc, it is ultimately the user who guides the proof, provides the structure, or comes up with a suitably strong induction statement. While this is often seen as a drawback of theorem proving, we consider it its greatest strength: It ensures that verification does not only tell you *that* a system is correct, but also *why* it is correct. Proofs are developed interactively with this technique but can be checked automatically for validity once derived, making the size and complexity of the proof irrelevant to soundness.

Verifying kernels What do the models and specifications look like in kernel verification?

Clearly a kernel needs to implement its API, so the specification is typically a formalisation of this API. This is created by a manual process with a potential for misstatement, as APIs tend to be specified informally or at best semi-formally using natural languages, and are typically incomplete and sometimes inconsistent. It is then desirable to utilise a formalism such that the correspondence between the informal and formal specification is relatively easy to see even for OS developers who are no experts in formal methods.

The kernel model is ideally the kernel executing on the hardware. In reality it is preferable to take advantage of the abstraction provided by the programming language in which the kernel is implemented, so the model becomes the kernel's source-level implementation. This introduces a reliance on the correctness of the compiler and linker (in addition to the hardware, boot-loader and firmware).

Some criticisms are commonly voiced when considering OS verification. *Is there any point if we have to rely on compiler and hardware correctness?* With source-level verification, compiler and hardware correctness have become orthogonal issues — when we have the required formal semantics for the language and hardware, verification of these system components can be attempted independently to that of the OS. Both hardware

and compiler verification are currently active areas of research. It should be noted that the gap between formal model and implementation will always exist, even in the presence of a verified processor, since real hardware is a physical realisation of some model and its correct operation is beyond the scope of formal verification [6] — one cannot prove the absence of manufacturing defects for example. The aim of OS verification is to significantly reduce the larger gap between user requirements and implementation and hence gain increased confidence in system correctness. *Even if the kernel is verified, what has been gained when user-level applications such as file-systems are not?* In the first scenario described in the introduction, the question is really that of what do we need to verify to be able to claim the *trusted* applications are correct. The kernel provides the basic abstraction over the underlying hardware necessary to enforce the boundary between trusted and untrusted applications and allows the behaviour of untrusted applications to be abstracted away or ignored when verifying the trusted code. Trusted applications may also have some redeeming characteristics when it comes to verification — they should be relatively small in a well-designed TCB and may take advantage of higher-level languages. For a hypervisor no additional work remains after OS verification — if the correct resource management and isolation is provided at the OS level then there is no possibility of faulty or malicious code executing in one partition from influencing or knowing about another.

Proof-based OS verification has been tried in the past [13, 20]. The rudimentary tools available at the time meant that the proofs had to end at the design level; full implementation verification was not feasible. The verification of Kit [4] down to object code demonstrated the feasibility of this approach to kernel verification, although on a system that is far simpler than any real-life OS kernel in use in secure systems today. There is little published work from the past 10–15 years on this topic, and we believe it is time to reconsider this approach.

3 Challenges and Experiences

Since the early attempts at kernel verification there have been dramatic improvements in the power of available theorem proving tools. Proof assistants like ACL2, Coq, PVS, HOL and Isabelle have been used in a number of successful verifications, ranging from mathematics and logics to microprocessors [5], compilers [3], and full programming platforms like JavaCard [18].

This has led to a significant reduction in the cost of formal verification, and a lowering of the feasibility threshold. At the same time the potential benefits have increased, given e.g. the increased deployment of embedded systems in life- or mission-critical situations, and the huge stakes created by the need to protect IP rights valued in the billions.

Consequently, we feel that the time is right to tackle, once again, the formal verification of OS kernels. We therefore decided about a year ago to attempt a verification of a real kernel. We are among several current efforts with this goal, notably VFiasco [9], VeriSoft [19] and Coyotos [15]. We target the L4 microkernel in our work as it is one of the smallest and best performing general-purpose kernels around, is deployed industrially and its design and implementation is well understood in our lab.

As this is clearly a high-risk project, we first embarked on a pilot project in the form of a constructive feasibility study. Its aim was three-fold: (i) to formalise the L4 API, (ii) to gain experience by going though a full verification cycle of a (small) portion of actual kernel code, and (iii) to develop a project plan for a verification of the full kernel. An informal aim was to explore and bridge the culture gap between kernel hackers and theorists, groups which have been known to eye each other with significant suspicion.

The formalisation of the API was performed using the B Method [1], as there existed a significant amount of experience with this approach among our student population. While L4 has an unusually detailed and very mature (informal) specification of its API [12], it came as no surprise to us to find that it was incomplete and ambiguous in many places, and inconsistent in a few. Furthermore it was sometimes necessary to extract the intended and expected kernel behaviour from the designers themselves and, occasionally, the source code.

In spite of those challenges, this part of the project turned out not overly difficult, and was done by a final-year undergraduate student. The result was a formal API specification that is mostly complete, describing the architecture-independent system calls in the IPC and threads subsystem of L4. Non-determinism was used in places where the current API was not clear on specific behaviour, and optimisations present at the API level that contributed significant complexity yet only provided disputable performance gains were omitted. The remaining subsystem (virtual memory) was formalised separately in the verification part of the project described below. The B specification consists of about 2000 lines of code.

The full verification was performed on the most complex subsystem, the one dealing with mapping of pages between address spaces and the revocation of such mappings, corresponding to approximately 5% of the kernel source code. We formalised a significant part of this API section and derived a verified implementation based on the existing implementation but with a subset of its functionality. Its implementation consists of the page tables, the *mapping database* (used to keep track of mappings for revocation purposes), and the code for lookup and

manipulation of those data structures.

Since our view of the system is that of execution on an abstract machine corresponding to the implementation language, the lowest-level model must rely on the formal semantics of the source code language and hardware. The L4 kernel is written in a mixture of a (mostly-C-like) subset of C++ with some assembler code. While the complete formal semantics of systems languages is an active area of research [9, 14], a complete semantics is not required. For our purpose it sufficed to have a semantics for the language subset actually used in the verified code. The code derived during this work was based on the data structures and algorithms in the existing implementation, but we had the freedom to make changes to remain in a safe subset of C++. Such changes are acceptable as long as they have no significant performance impact.

Semantics for the assembler code could be derived from the hardware model. This was not tackled in our pilot project, as the slice was implemented without resorting to assembler (our work is based on ARM processors, which feature hardware-loaded TLBs). We did, however, formalise some aspects of the hardware, such as the format of page table entries. In principle, processor manufacturers could provide their descriptions of the ISA level in a HDL to facilitate this, in practise this rarely happens. Instead one typically uses ISA reference manuals as a basis for formalisation. Hardware models of commercial microprocessors such as x86 and ARM [7] are available. While these are presently somewhat incomplete for kernel verification purposes, they should be extendable without major problems.

We use higher-order logic (HOL) as our language for system modelling, specification and refinement, specifically the instantiation of HOL in the theorem prover Isabelle. HOL is an expressive logic with standard mathematical notation. Terms in the logic are typed, and HOL can directly be used as simple functional programming language. HOL is consequently unesoteric for programmers with a computer science background. We are using this functional language to describe the behaviour of the kernel at an abstract level. This description is then *refined* inside the prover into a program written in a standard, imperative, C-like language. In a refinement some part the state space is made more concrete, substitutions for operations for the new state space are described and proven to simulate the abstract operations. For example, an abstract albeit simplistic view of the page table for an address space is a function mapping virtual pages to page table entries. Refinement of this would replace the function with a page table data structure such as a multi-level page table and the corresponding insertion and lookup procedures.

The abstract description is at the level of a reference manual and relatively easy to understand. This is the level we use for analysing the behaviour of the system and for proving additional simple safety properties, such as the requirement that the same virtual address can never be translated to two different physical addresses. The abstract model is operational, essentially a state machine. This is close to the intuition that systems implementors have of kernel behaviour as an extended hardware machine, and has an associated well-understood hierarchical refinement methodology. An operational model for kernel behaviour in HOL then helps minimising the gap between requirements and specification. At the end of the refinement process stands a formally verified imperative program. A purely syntactic translation then transforms this program into ANSI C. A detailed description of this process can be found elsewhere [11, 16].

We found Isabelle suitable for the task. It is mature enough for use in large-scale projects and well-documented, with a reasonably easy-to-use interface. Being actively developed as an open source tool, we are able to extend it and (working with the developers) to fix problems should they arise.

During the process of formalising the VM subsystem we discovered several places in the existing semi-formal description and reference manual where significant ambiguities existed, and some inconsistencies with implementation behaviour. The ordering of internal operations in the system calls responsible for establishing and revoking VM mappings, *map, grant* and *unmap*, was underspecified, leading to problems when describing a formal semantics. A potential security problem could result from one of the inconsistencies found.

An interesting experience was that the expected culture clash between kernel hackers and formal methods people was a non-issue. The first author of this paper is a junior PhD student with significant kernel design and implementation experience. He obtained the necessary formal methods background within two months to the degree where he could productively perform proofs in Isabelle. It took about the same time for all participants to gain an appreciation of the other side's challenges. This is one of the reasons that we believe that the full verification of L4 is achievable.

However, we are convinced that some important requirements must be met for such a project to have a chance. It is essential that some of the participants have significant experience with formal methods and a good understanding of what is feasible and what is not, and how best to approach it. On the other hand, it is essential that some of the participants have a good understanding of the kernel's design and implementation, the trade-offs underlying various design decisions, and the factors that determine the kernel's performance. It must

be possible to change the implementation if needed, and that requires a good understanding of changes that can be done without undermining performance.

4 Looking Ahead

The challenges for formal verification at the kernel level relate to performance, size, and the level of abstraction. Runtime performance of the verified code is one of the highest priorities in operating systems, particularly in the case of a microkernel or virtual-machine monitor, which is invoked frequently. Software verification has traditionally not focused on this aspect — getting it verified was hard enough. Size is a limiting factor as well. Even a small microkernel like L4 measures about 10,000 lines of code. Larger systems have been verified before, but only on an abstract description, not on the implementation level. Compared to application code, the level of abstraction is lower for kernel code. Features like direct hardware access, pointer arithmetic and embedded assembly code are not usually the subject of mainstream verification research.

Another practically important issue is ensuring that verified code remains maintainable. In principle, every change to the implementation might invalidate the verification. The extent to which this occurs will depend on the nature of the change. Hand optimisation of the IPC path, for example, may require less work to reestablish correctness than changes to system call semantics, since in the optimisation case higher-level abstraction proofs remain valid. The fact that the proofs are machine-checked makes it easy to determine which proofs are broken by the change, and techniques such as a careful, layered proof structure and improved automation for simple changes help to make this problem easier to handle. Whether this is enough remains an open question.

We believe that full formal specification of the kernel API prior to kernel implementation is desirable. The benefits of having a complete, consistent and unambiguous reference for kernel implementors, users and verifiers is clear, and the effort required is modest when compared with either implementation or verification.

The investment for the virtual memory part of the pilot project was about 1.5 person years. All specifications and proofs together run to about 14,000 lines of proof scripts. This is significantly more than the effort invested in the virtual memory subsystem in the first place, but it includes exploration of alternatives, determining the right methodology, formalising and proving correct a general refinement technique, as well as documentation and publications.

We estimate that the full verification of L4 will take about 20 person years, including verification tool development. This sounds a lot, but must be seen in relation to the cost of developing the kernel in the first place, and the potential benefits of verification. The present kernel [12] was written by a three-person team over a period of 8–12 months, with significant improvements since. Furthermore, for most of the developers it was the third in a series of similar kernels they had written, which meant that when starting they had a considerable amount of experience. A realistic estimate of the cost of developing a high-performance implementation of L4 is probably at least 5–10 person years.

Under those circumstances, the full verification no longer seems prohibitive, and we argue that it is, in fact, highly desirable. The kernel is the lowest and most critical part of any software stack, and any assurances on system behaviour are built on sand as long as the kernel is not shown to behave as expected. Furthermore, formal verification puts pressure on kernel designers to simplify their systems, which has obvious benefits for maintainability and robustness even when not yet formally verified.

There is a saying that the problem with engineers is that they cheat in order to get results, the problem with mathematicians is that they work on toy problems in order to get results, and the problem with program verifiers is that they cheat on toy problems in order to get results. We are ready to tackle the real problem without cheating.

Acknowledgements

We would like to thank all those who contributed to the part of the L4 kernel verification pilot project which we report on here — Kevin Elphinstone, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Ken Robinson and Adam Wiggins.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN'01, Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122, 2001.

[3] S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. COCV'03*, Electronic Notes in Theoretical Computer Science, pages 33–50, 2003.

[4] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[5] B. C. Brock, W. A. Hunt, Jr., and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., 1994.

[6] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[7] A. Fox. Formal specification and verification of ARM6. In D. Basin and B. Wolff, editors, *TPHOLs '03*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.

[8] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.

[10] In G. Klein, editor, *Proc. NICTA FM Workshop on OS Verification*. Technical Report 0401005T-1, National ICT Australia, 2004.

[11] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[12] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual Version X.2*. University of Karlsruhe, Oct 2001. `http://l4ka.org/projects/version4/l4-x2.pdf`.

[13] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1980.

[14] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[15] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In Klein [10], pages 1–19.

[16] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In Klein [10], pages 73–97.

[17] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *HotOS-VI*, 1997.

[18] VerifiCard project. `http://verificard.org`, 2005.

[19] VeriSoft project. `http://www.verisoft.de`, 2005.

[20] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.