Proof Engineering Considered Essential

Gerwin Klein

NICTA[*] and UNSW, Sydney, Australia

{first-name.last-name}@nicta.com.au

Abstract. In this talk, I will give an overview of the various formal verification projects around the evolving seL4 microkernel, and discuss our experience in large-scale proof engineering and maintenance.
In particular, the presentation will draw a picture of what these verifications mean and how they fit together into a whole. Among these are a number of firsts: the first code-level functional correctness proof of a general-purpose OS kernel, the first non-interference proof for such a kernel at the code-level, the first binary-level functional verification of systems code of this complexity, and the first sound worst-case execution-time profile for a protected-mode operating system kernel.
Taken together, these projects produced proof artefacts on the order of 400,000 lines of Isabelle/HOL proof scripts. This order of magnitude brings engineering aspects to proofs that we so far mostly associate with software and code. In the second part of the talk, I will report on our experience in proof engineering methods and tools, and pose a number of research questions that we think will be important to solve for the wider scale practical application of such formal methods in industry.

1 The seL4 Verification

This extended abstract contains a brief summary of the seL4 verification and proof engineering aspects. A more extensive in-depth overview has appeared previously [13].

The seL4 kernel is a 3rd generation microkernel in the L4 family [17]. The purpose of such microkernels is to form the core of the trusted computing base of any larger-scale system on top. They provide basic operating system (OS) mechanisms such as virtual memory, synchronous and asynchronous messages, interrupt handling, and in the case of seL4, capability-based access control. The idea is that, using these mechanisms, one can isolate software components in time and space from each other, and therefore not only enable verification of such components in isolation and in higher-level programming models, but even forego the formal verification of entire components in a system altogether, and focus on a small number of trusted critical components instead, without sacrificing assurance in the critical properties of the overall system [3]. This general idea is not new. For instance, it can be found for simpler separation kernels in the

[*] NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program

MILS setting [2]. For modern systems, some of the untrusted components will be an entire monolithic guest OS such as Linux. That is, the microkernel is used not only for separation, but also as a full virtualisation platform or hypervisor.

This setting provides the motivation for the formal verification of such kernels: clearly, they are at the centre of trust for the overall system — if the microkernel misbehaves, no predictions can be made about the security or safety of the overall system running on it. At the same time, microkernels are small: roughly on the order of 10,000 lines of C code. The seL4 verification shows that this is now within reach of full formal code-level verification of functional and non-functional properties, and with a level of effort that is within a factor of 2–5 of normal high quality (but not high assurance) software development in this domain. With further research in proof engineering, automation, and code and proof synthesis, we think this factor can be brought down to industrially interesting levels, and in specific cases, can even be made cheaper than standard software development.

Apart from its scale, two main requirements set the verification of seL4 apart from other software verification projects: a) the verification is at the code level (and more recently even at the binary level), and b) it was a strict requirement of the project not to sacrifice critical runtime performance for ease of verification.

The second requirement is crucial for the real-world applicability of the result. Especially in microkernels, context switching and message passing performance is paramount for the usability of the system, because these will become the most frequently run operation not just of the kernel, but of the entire system. The mere idea of the first generation of microkernels has famously been criticised for being prohibitive for system performance and therefore ultimately unusable [24]. Time has shown this argument wrong. The second generation of microkernels have demonstrated context switching and message passing performance on par with standard procedure calls as used in monolithic kernels [17]. The third generation has added strong access control to the mix without sacrificing this performance. Such microkernels now power billions of mobile devices, and therefore arguably have more widespread application than most (or maybe all) standard monolithic kernels. All this rests on the performance of a few critical operations of such kernels, and it is no wonder that the field seems obsessed with these numbers. Using simplifications, abstractions or verification mechanisms that lead to one or two orders of magnitude slow-down would be unacceptable.

The first requirement — code-level verification instead of verification on high-level models or manual abstractions — was important to achieve a higher degree of assurance in the first place, and later turned out to be indispensable for maintaining the verification of an evolving code base. The various separate verification projects around seL4 took place over a period of almost a decade, but they fully integrate and provide machine-checked theorems over the same code base (except the worst-case execution-time (WCET) analysis, which uses different techniques). Whenever the code or design of the kernel changes, which happens regularly, it is trivial and automatic to check which parts of the verification break and need to be updated. This would be next to impossible if there was a manual abstraction step involved from the artefact the machine runs to the artefact the proof is concerned with. It has often been observed that even light-weight application of formal methods brings significant benefit early in the life cycle of a project. Our experience shows that strong benefits can be sustained throughout the much longer maintenance phase of software systems. As I will show in the talk, maintaining proofs together with code is not without cost, but at least in the area of critical high-assurance systems changes can now be made with strong confidence, and without paying the cost of full expensive re-certification.

The talk will describe the current state of the formal verification of the seL4 kernel, which is conducted almost exclusively in the LCF-style [10] interactive proof assistant Isabelle/HOL [20]. The exceptions are binary verification, which uses a mix of Isabelle, HOL4 [23] and automatic SMT solvers, and the WCET analysis, which uses the Chronos tool, manual proof and model checking for the elimination of infeasible paths.

In particular, the verification contains the following proofs:

- functional correctness [14] between an abstract higher-order logic specification of seL4 and its C code semantics, including the verification of a high-performance message-passing code path [13];
- functional correctness between the C code semantics and the binary of the seL4 kernel after compilation and linking [21], based on the well-validate Cambridge ARM semantics [7];
- the security property *integrity* [22], which roughly says that the kernel will not let user code change data without explicit write permission;
- the security property *non-interference* [19,18], which includes *confidentiality* and together with integrity provides isolation, which implies *availability* and *spacial separation*;
- correct user-level system initialisation on top of the kernel [5], according to static system descriptions in the capability distribution language capDL [15], with a formal connection to the security theorems mentioned above [13];
- a sound binary-level WCET profile obtained by static analysis [4], which is one of the key ingredients to providing temporal isolation.

Verification can never be absolute; it must always make fundamental assumptions. In this work we verify the kernel with high-level security properties down to the binary level, but we still assume correctness of TLB and cache flushing operations as well as the correctness of machine interface functions implemented in handwritten assembly. Of course, we also assume hardware correctness. We give details on the precise assumptions of this verification and how they can be reduced even further elsewhere [13].

The initial functional correctness verification of seL4 took 12 person years of work for the proof itself, and another 12-13 person years for developing tools, libraries, and frameworks. Together, these produced about 200,000 lines of Isabelle/HOL proof scripts [14].

The subsequent verification projects on security and system properties on top of this functional correctness proof were drastically cheaper, for instance less than 8 person months for the proof of integrity, and about 2 person years for the proof of non-interference [13]. During these subsequent projects, the seL4 kernel evolved.

While there were no code-level defects to fix in the verified code base, changes included performance improvements, API simplifications, additional features, and occasional fixes to parts of the non-verified code base of seL4, such as the initialisation and assembly portions of the kernel. Some of these changes were motivated by security proofs, for instance to simplify them, or to add a scheduler with separation properties. Other changes were motivated by applications the group was building on top of the kernel, such as a high-performance data base [11].

This additional work increased the overall proof size to roughly 400,000 lines of Isabelle proof script. Other projects of similar order of magnitude include the verified compiler CompCert [16], the Verisoft project [1] that addressed a whole system stack, and the four colour theorem [8,9].

While projects of this size clearly are not yet mainstream, and may not become mainstream for academia, we should expect an increase in scale from academic to industrial proofs similar to the increase in scale from academic to industrial software projects. There is little research on managing proofs and formal verification on this scale, even though we can expect verification artefacts to be one or two orders of magnitude larger than the corresponding code artefacts. Of course, we are not the first to recognise the issue of scale for proofs. All of the other large-scale verification projects above make note of it, as did previous hardware verifications [12].

We define a large scale proof as one that no single person can fully understand in detail at any one time. Only collaboration and tool support make it possible to conduct and check such proofs with confidence.

Many of the issues faced in such verification projects are similar to those in software engineering: there is the matter of merely browsing, understanding, and finding intermediate facts in a large code or proof base; there are dependencies between lemmas, definitions, theories, and other proof artefacts that are similar to dependencies between classes, objects, modules, and functions; there is the issue of refactoring existing proofs either for better maintainability or readability, or even for more generality and additional purposes; and there are questions of architecture, design, and modularity in proofs as well as code. Some of the proof structure often mirrors the corresponding code structure, other parts do not necessarily have to do so. For large scale proofs, we also see issues of project management, cost and effort estimation, and team communication. These again have similarities with software engineering, but also have their unique challenges.

Based on our experience in the verification projects mentioned above, the following research questions would be interesting and beneficial to solve.

1. What are the fundamental differences and similarities between proof engineering and software engineering?
2. Can we estimate time and effort for a specific proof up front, and with which confidence? Related questions are: can we predict the size of the proof artefacts a project will produce? Are they related to effort? Can we predict the complexity or difficulty of a proof given artefacts that are available early in the project life cycle, such as initial specifications and/or code prototypes?

3. Which technical tools known from traditional software development could make an even higher impact on proof engineering? Emerging prover IDEs [25] for instance can provide more semantic information than typical programming IDEs, and refactoring tools can be more aggressive than their code counterparts because the result is easily checked.
4. Are there more fundamental ways in which proof irrelevance, formal abstraction, and modularity can be exploited for the management of large scale proofs?
5. Can concepts such as code complexity or technical debt be transferred to proofs in a useful way?
6. Are the fundamental aspects of proof library design that are different to software libraries? What are the proof and specification patterns?
7. Empirical software engineering has identified a number of "laws" that statistically apply to the development of large software projects [6]. Which of these continue to hold for proofs? Are there new specific correlations that hold for large scale proofs?

Some of these questions do already receive some attention, but not yet to the degree required for making significant broader progress in this area.

This is clearly just a subjective subset of research question in this space. As software engineering has done for code development, we think that addressing such questions for large-scale proofs will have a positive impact not only on the industrial feasibility of large verification projects, but also on the every-day development of smaller proofs.

References

1. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A., Tsyban, A.: Balancing the load — leveraging a semantics stack for systems verification. JAR: Special Issue Operat. Syst. Verification **42, Numbers 2–4** (2009) 389–454
2. Alves-Foss, J., Oman, P.W., Taylor, C., Harrison, S.: The MILS architecture for high-assurance embedded systems. Int. J. Emb. Syst. **2** (2006) 239–247
3. Andronick, J., Greenaway, D., Elphinstone, K.: Towards proving security in the presence of large untrusted components. In Klein, G., Huuck, R., Schlich, B., eds.: 5th SSV, Vancouver, Canada, USENIX (Oct 2010)
4. Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., Heiser, G.: Timing analysis of a protected operating system kernel. In: 32nd RTSS, Vienna, Austria (Nov 2011) 339–348
5. Boyton, A., Andronick, J., Bannister, C., Fernandez, M., Gao, X., Greenaway, D., Klein, G., Lewis, C., Sewell, T.: Formally verified system initialisation. In Lindsay Groves, Jing Sun, ed.: 15th ICFEM, Queenstown, New Zealand, Springer (Oct 2013) 70–85
6. Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Pearson, Addison Wesley (2003)
7. Fox, A., Myreen, M.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Kaufmann, M., Paulson, L.C., eds.: 1st ITP. Volume 6172 of LNCS., Edinburgh, UK, Springer (Jul 2010) 243–258

8. Gonthier, G.: A computer-checked proof of the four colour theorem. http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf (2005)
9. Gonthier, G.: Formal proof — the four-color theorem. Notices of the American Mathematical Society **55**(11) (2008) 1382–1393
10. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF. Volume 78 of LNCS. Springer (1979)
11. Heiser, G., Le Sueur, E., Danis, A., Budzynowski, A., Salomie, T.I., Alonso, G.: RapiLog: Reducing system complexity through verification. In: EuroSys, Prague, Czech Republic (Apr 2013) 323–336
12. Kaivola, R., Kohatsu, K.: Proof engineering in the large: Formal verification of pentium® 4 floating-point divider. In: Correct Hardware Design and Verification Methods, Springer (2001) 196–211
13. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems (TOCS) **32**(1) (Feb 2014) 2:1–2:70
14. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP, Big Sky, MT, USA, ACM (Oct 2009) 207–220
15. Kuz, I., Klein, G., Lewis, C., Walker, A.: capDL: A language for describing capability-based systems. In: 1st APSys, New Delhi, India (Aug 2010) 31–36
16. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In Morrisett, J.G., Jones, S.L.P., eds.: 33rd POPL, Charleston, SC, USA, ACM (2006) 42–54
17. Liedtke, J.: Towards real microkernels. CACM **39**(9) (Sep 1996) 70–77
18. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: from general purpose to a proof of information flow enforcement. In: IEEE Symp. Security & Privacy, San Francisco, CA (May 2013) 415–429
19. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, ed.: The Second International Conference on Certified Programs and Proofs, Kyoto, Springer (Dec 2012) 126–142
20. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
21. Sewell, T., Myreen, M., Klein, G.: Translation validation for a verified OS kernel. In: PLDI, Seattle, Washington, USA, ACM (Jun 2013) 471–481
22. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F., eds.: 2nd ITP. Volume 6898 of LNCS., Nijmegen, The Netherlands, Springer (Aug 2011) 325–340
23. Slind, K., Norrish, M.: A brief overview of HOL4. In Otmane Ait Mohamed, Csar Muoz and Sofine Tahar, ed.: Theorem Proving in Higher Order Logics, 20th International Conference, Montral, Canada, Springer (Aug 2008) 28–32
24. Tannenbaum, A., Torwalds, L.: LINUX is obsolete. Discussion on comp.os.minix, https://groups.google.com/forum/#!topic/comp.os.minix/wlhw16QWltI (1992)
25. Wenzel, M.: Isabelle/jEdit - a prover IDE within the PIDE framework. In Jeuring, J., Campbell, J.A., Carette, J., Reis, G.D., Sojka, P., Wenzel, M., Sorge, V., eds.: Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management. Volume 7362 of LNCS., Springer (2012) 468–471