Proof Engineering Challenges for Large-Scale Verification

Gerwin Klein

NICTA[⋆] and UNSW, Sydney, Australia

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

In this extended abstract I summarise challenges for proof engineering that we encountered in the formal verification of the seL4 microkernel [7], and its subsequent proofs of integrity [12], non-interference [10], and binary correctness [11]. I focus on problems where there is scope for automation using AI and machine-learning techniques. For more background on the seL4 verification, and an analysis of the effort spent on it, see previous work [6].

The seL4 kernel is a 3rd generation microkernel in the L4 family [9]. Such kernels 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, enabling separate compositional verification of trusted components as well as proof that no such correctness is required of untrusted components, because the kernel and its policy configuration already sufficiently constrain their behaviour [2].

The verification of seL4 was not a large project by industrial software development standards, but it was sizeable for an academic formal verification project. The functional correctness proof of seL4 took roughly 12 person years, the overall initial project, including tool building, libraries, and research in scalable proof techniques, usable semantics of the C programming language, etc. took about 25 person years; for a more precise analysis see [6]. This effort later paid off in the proof of high-level security properties: they were much easier to show, because they could now be established on an abstract specification instead of directly on the code. Integrity cost less than 8 person months, non-interference less than 21 person months, and updates to the kernel to add a separation scheduler cost another 21 person months, including updates to all existing proofs. Automatic binary verification for functional correctness then extended these properties down to the low-level semantics of ARMv6 machine instructions.

The largest of these proofs, the initial functional correctness verification produced about 200,000 lines of Isabelle/HOL proof scripts [7] with a team of on average 12 people over 4 years (about 7 full-time equivalent).

During the subsequent proofs, the seL4 kernel evolved. While there were no C-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

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

portions of the kernel. Some of these changes were motivated by the security proofs, for instance to simplify them, or to add the scheduler with separation properties. Other changes were motivated by applications the group was building.

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 [8], the Verisoft project [1] that addressed a whole system stack, and the four colour theorem [4].

There is little research on managing formal verification on this scale and the experience in our verification was that this scale makes a significant difference to how proofs are developed and maintained [3]. The key difference to smaller proofs is that no single person at any time understands the whole proof, or even the whole code base in detail. Only the fact the proof is machine-checked gives us confidence in the soundness of the overall result. Of course, we are not the first to recognise the issue of scale for proofs. All of the other large-scale verification projects mentioned previously make note of it, as did previous hardware verifications [5].

While many of these issues are similar to traditional software engineering, there are differences that could be exploited to increase the productivity of the verification engineer and to increase the scale at which such proofs can be applied.

In particular, large-scale proofs have the following two properties that make them more amenable to automation and assistance by AI and machine learning techniques than traditional code development:

- it is cheap and easy to check if an existing proof is correct
- in a large-scale proof there are often a large number of analogous or similar cases and proof fragments

The first property is interesting for proof refactoring: while in code refactoring it is important to be semantics preserving, in proof refactoring, there is an easy check if the refactored proof still works.

This can be exploited in techniques that deal with the second property. Assuming we were able to automatically find a similarity between a current proof goal and some previous proof fragment, a proof suggestion based on this fragment does not necessarily have to be correct. It will be checked by the prover anyway, and it may even be useful to the verification engineer in its incorrect form, because she may be able to adjust it. Specific areas where such techniques might be useful are

- finding analogies within one proof with a large number of cases, and suggesting proofs to the engineer, or optimising proof search based on previous cases;
- finding and exploiting similarity between recurring proof fragments distributed over different proofs, in particular proofs of other verification engineers;
- suggesting and automating lemma extraction for recurring proof fragments;
- automatically generalising and re-proving lemmas whose statement was needlessly specific to achieve better re-use;
- suggesting, e.g. via auto-completion, high-level proof structure that has been extracted from previous proofs with similar statements.

It is crucial for such tools not to get into the way of normal proof interaction if they are to be useful to the verification engineer. Code completion and similar techniques from traditional code-based integrated development environments that are suggestive rather than prescriptive are directly applicable.

Isabelle's PIDE interface [13] is already making good progress in this direction, for instance providing automatic feedback on counter examples or searching for proofs in the background while the engineer goes about her work. Such techniques benefit immensely from the increasing number of cores on desktop machines, because multiple separate analyses distribute trivially over them.

With more sophisticated suggestion and analysis tools becoming available, there should not only be room for significant improvements in productivity for large-scale proofs, and but also the possibility of making such proofs more accessible to new team members. In our experience, learning Isabelle is easy. Finding your way in a large proof and code base is much more time consuming.

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. Andronick, J., Greenaway, D., Elphinstone, K.: Towards proving security in the presence of large untrusted components. In : 5th SSV, USENIX (2010)
3. Bourke, T., Daum, M., Klein, G., Kolanski, R.: Challenges and experiences in managing large-scale proofs. In : Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management, Springer (2012)
4. Gonthier, G.: Formal proof — the four-color theorem. Notices of the American Mathematical Society **55**(11) (2008) 1382–1393
5. 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
6. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Trans. Comp. Syst. **32**(1) (2014) 2:1–2:70
7. 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, ACM (2009) 207–220
8. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In : 33rd POPL, ACM (2006) 42–54
9. Liedtke, J.: Towards real microkernels. CACM **39**(9) (1996) 70–77
10. 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, (2013) 415–429
11. Sewell, T., Myreen, M., Klein, G.: Translation validation for a verified OS kernel. In: PLDI, ACM (2013) 471–481
12. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 enforces integrity. In : 2nd ITP. Volume 6898 of LNCS., Springer (2011) 325–340
13. Wenzel, M.: Isabelle/jEdit - a prover IDE within the PIDE framework. In: Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management. Volume 7362 of LNCS., Springer (2012) 468–471