Editors: Mohamed Kaâniche, mohamed.kaaniche@laas.fr | Aad van Moorsel, aad.vanmoorsel@ncl.ac.uk

It's Time for Trustworthy Systems

Gernot Heiser, Toby Murray, and Gerwin Klein | NICTA

The time for truly trustworthy systems, backed by machine-checked formal proof and analysis, has arrived. Over the past few decades, advances in formal verification and analysis technologies mean that these tools can now scale sufficiently to cover the entire software trusted computing base of appropriately designed real-world systems.

We base this claim on our experience with the formal verification and analysis of the seL4 microkernel.[1] A microkernel is a minimal OS kernel; seL4 weighs in at under 10,000 lines of code. The microkernel is also the most critical trusted component in any system built on it. It lets us build well-performing systems with millions of lines of legacy code, while reducing the trusted code base to the same order of 10,000 lines of code[2] that we've already demonstrated we can formally verify.

For this to work, the microkernel must be able to effectively isolate trusted from untrusted code (see Figure 1), spatially and temporally. The high-level properties needed for this are integrity, confidentiality, and predictable *worst-case execution time* (WCET). Depending on the deployment context, the focus might shift between safety or security, and additional properties will be of interest (see Figure 2).

Providing properties such as integrity and confidentiality has been OS kernels' primary function for a long time. The exciting part is that we now can fully formalize and prove these properties of real system implementations at the C code level. Even more interesting is that we can now use them to drastically reduce the effort of proving whole systems' security goals. Of course, that was the idea all along: provide strong enforcement mechanisms that let us construct and conceptually reason about the system more easily. The shift is from conceptual to formal.

Our progress on the deep formal analysis of the seL4 microkernel has been enabled by our previous proof of functional correctness.[1] This proof showed that seL4's C implementation conforms to an abstract functional specification of its behavior. This proof was the first of its kind, with the relatively high cost of roughly 25 person-years. Although this proof is important in its own right, its true power is in reducing the effort for further analysis. We can now build on this result when reasoning about behaviors of seL4 because we can reduce reasoning about the implementation to reasoning about the specification. In our experience, this means an order of magnitude less effort.

Proving Security

Our experience after the functional-correctness proof demonstrates that proving whole-system

Copublished by the IEEE Computer and Reliability Societies

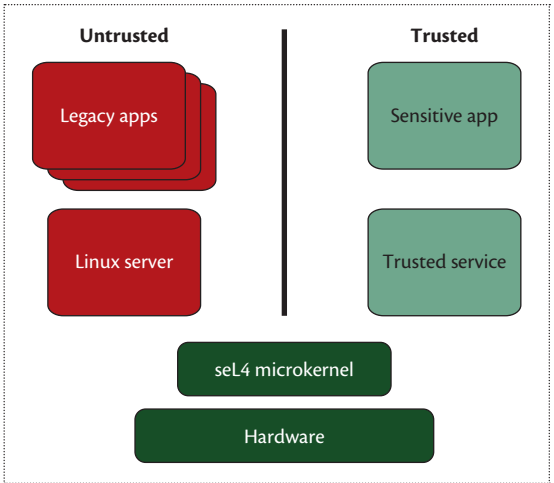

Figure 1. The separation provided by the seL4 microkernel. This separation lets us build well-performing systems with millions of lines of legacy code, while reducing the trusted code base to a manageable level.

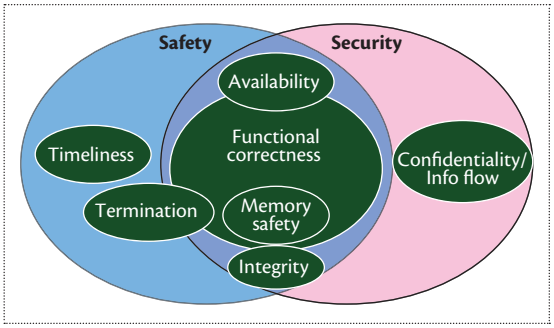

Figure 2. Properties of trustworthy systems. To effectively isolate trusted from untrusted code, the microkernel requires integrity, confidentiality, and predictable worst-case execution time. Depending on the deployment context, the focus might shift between safety or security, and some of the other listed properties will be of interest.

security properties is now tractable at a reasonable cost. We have since completed a proof of integrity enforcement for seL4 in less than 10 person-months,[3] and we're completing the dual proof of confidentiality enforcement. These proofs combine into the classic security property of noninterference. We carefully constructed our confidentiality and noninterference formulations to be preserved by the formal refinement statement that embodies functional correctness. Therefore, we can continue to exploit the functional-correctness proof's benefits.

Each of these proofs maps the seL4 protection state to a corresponding abstract access control policy. We then define the security properties' integrity and confidentiality against this policy. Integrity limits what the currently running thread can modify; confidentiality limits what it can read or infer.

Because seL4 implements a dynamic capability-based access control mechanism, its protection state can evolve. So, we also proved *authority confinement*, which means that, for well-formed policies, the protection state remains consistent with the policy. In other words, the policy places an upper bound on the authority in the system.

Compared to integrity and authority confinement, confidentiality is harder to reason about. Unlike modifying information, reading information isn't directly observable in the system state. To determine whether the kernel might have read a private system state on behalf of a subject, we must consider all other counterfactual executions in which this private state differs. If the execution results are the same in all cases, we can conclude that this private state doesn't influence the execution and so is never read. The difference from earlier proofs is that we must consider multiple executions instead of analyzing one execution at a time.

Even though confidentiality is harder to prove, completing the confidentiality proofs for almost all of the seL4 kernel took only under 14 person-months. Out of more than 1,250 lemmas, only 13 remained unproved at the time of writing.

The next step is to pull kernel-level properties up to system-level properties. This is precisely what noninterference is good for. We can use it to separate trusted code from legacy code. Also, for suitably designed systems, we can reduce our analysis from the millions of lines of code for the entire system to just the thousands of lines that constitute its trusted components. Noninterference tells us that we can analyze the trusted components independently of the rest of the system because they're isolated.

We expect to see full proofs of noninterference for seL4-based systems in the next one to two years, based on the noninterference theorem for seL4.

What about Safety?

At the OS level, security and safety reduce mostly to the same key properties. However, as Figure 2 indicates, many safety-critical systems have an extra requirement: timeliness. Medical implants, industrial robots, and vehicles are hard real-time systems; they must react to an external event within a strictly bounded time. Yet they also contain untrustworthy legacy code. Think of a medical implant that must communicate with the external world via a wireless network for monitoring and maintenance. The network drivers and protocol stack likely comprise tens of thousands of lines of code and can't be trusted, requiring the isolation provided by the microkernel. This means that the microkernel must hand control to the life-support functions quickly enough, no matter what the legacy code was doing when some critical sensor raised an interrupt.

Although the system can interrupt the legacy code at any time, that code might have invoked an arbitrary microkernel call to obtain a service. The implication is that the uninterruptible execution time of all kernel calls must be strictly bounded.

Such strict WCET bounds are reasonably easy to establish for classic real-time OSs, but such systems don't provide isolation

and therefore aren't suitable for this class of *multicriticality systems*. seL4 has treaded new territory here as well, being the first OS kernel providing strong isolation that has undergone complete WCET analysis. This analysis is sound in that the results are guaranteed to be upper bounds on the latencies that can be observed in real execution. However, even on a highly complex processor, such as an ARM11 or Cortex-A8, the degree of pessimism is moderate. This is because the observed latencies (which present a lower bound on WCET) are within a factor of five of the upper bounds. Most of this pessimism is the unavoidable result of underspecified hardware operation.[4]

How Can I Attack a System with Proof?

There are limits to what formal security and safety proofs can achieve. Such fundamental limits also apply to other methods such as testing, but with the strength of mathematical proof, it's easy to get carried away.

Maybe counterintuitively, the possibility of mistakes in the proof itself is a nonissue in practice. The proof is machine-checked. Although soundness defects can't be fundamentally ruled out, they can be made arbitrarily unlikely.

The fundamental limits of formal reasoning are instead the assumptions the proof rests on and the gap between the formal property and our human understanding of it. Attacking these is much more fruitful. For instance, a usual assumption is hardware correctness. A viable attack would be to make the hardware fail in interesting ways—for instance by overheating—revealing information and thereby violating confidentiality. Likewise, running the system on hardware that doesn't match the proof's assumptions might also cause the system to fail. These assumptions also embody the system's assumed context of use. The proof handily identifies these assumptions and gives us the defense method: ensure the system is deployed under the right operating conditions.

A more subtle attack is to exploit hardware details beneath the lowest abstraction layer in the verification. If you can exploit the mismatch between reality and model, you might be able to find a side channel. For instance, functional models don't talk about hardware timing, so confidentiality proofs don't guarantee the absence of covert timing channels. The defense against these must be with traditional analyses. In this particular case, you could use the kernel's WCET profile.

You can also attack the understanding of the specification. Although the specification is easier to understand than the code, it's still by no means simple. You might find a behavior in the specification that is surprising to system designers. The defense against this is to ensure that your system's security goal is crisp and easy to understand. Then prove this goal on the basis of the specification, shifting understanding from complex to simple.

Although understanding these limitations is necessary, formal analysis provides an immense benefit because validating and defending assumptions is much easier than analyzing code. If you deploy formal analysis correctly, whole classes of problems disappear. If you're looking for buffer-overflow attacks in seL4, you're wasting your time. If you're trying to escalate privilege or corrupt another application without authorization, you won't succeed. The proof has checked all possible behaviors.

Security proofs don't need to end at the internal-policy level—for example, which component or actor can talk to whom. Instead, the proof can and should go up to the system's actual high-level security goal—for example, that no information can be exchanged between

two high-level networks. Even with this level of assurance, there's no magic bullet. You still need orthogonal methods to ensure that you're building the right system with the right requirements, and not the wrong system correctly.

We expect full proofs of security or safety for suitably architected systems within the next one to two years. We know how to architect and analyze (some) secure systems with minimal trusted code bases, we know how to complete kernel-level security proofs, and we know how to build trustworthy user components. The next challenges are to compose these parts into one final proof of a system-wide security goal and to decrease the cost of verification through code synthesis and stronger automation.

Although the initial investment into the functional-correctness proof of seL4 was high, this proof keeps paying off as we prove further properties on top of it. In our experience, these proofs get easier each year.

The seL4 kernel is just the first system with such a comprehensive suite of strong high-level properties. The excuse that machine-checked proofs of safety and security are infeasible doesn't apply anymore. The age in which truly security- and safety-critical systems should be fielded without proof is ending. ■

Acknowledgments

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.

References

1. G. Klein et al., "seL4: Formal Verification of an Operating System Kernel," *Comm. ACM*, vol. 53, no. 6, 2010, pp. 107–115.
2. J. Andronick, D. Greenaway, and K. Elphinstone, "Towards Proving Security in the Presence of Large Untrusted Components," *Proc. 5th Int'l Workshop Systems Software Verification*, Usenix Assoc., 2010; http://static.usenix.org/events/ssv10/tech/full_papers/Andronick.pdf.
3. T. Sewell et al., "seL4 Enforces Integrity," *Proc. 2nd Int'l Conf. Interactive Theorem Proving*, LNCS 6898, Springer, 2011, pp. 325–340.
4. B. Blackham, Y. Shi, and G. Heiser, "Improving Interrupt Response Time in a Verifiable Protected Microkernel," *Proc. 7th EuroSys Conf.*, 2012; www.ssrg.nicta.com.au/publications/papers/Blackham_SH_12.pdf.

Gernot Heiser leads the Trustworthy Systems project at NICTA and is Scientia Professor and John Lions Chair of Operating Systems at the School of Computer Science and Engineering at the University of New South Wales. Contact him at gernot@nicta.com.au.

Toby Murray is a researcher in the Software Systems Research Group at NICTA and a conjoint lecturer in the School of Computer Science and Engineering at the University of New South Wales. Contact him at toby.murray@nicta.com.au.

Gerwin Klein is a research leader at NICTA and a conjoint associate professor in the School of Computer Science and Engineering at the University of New South Wales. Contact him at gerwin.klein@nicta.com.au.