# Correct OS kernel? Proof? Done!

Gerwin Klein

NICTA[1] and University of New South Wales, Sydney, Australia

gerwin.klein@nicta.com.au

Two years ago Gernot Heiser demanded in this venue *Your System is Secure? Prove it!* [5] He also mentioned the L4.verified [3] project at NICTA which is doing just that. This proof is now completed and in this article I'm showing what we have proved and what that means for security.

## The seL4 microkernel: Correct!

The basic idea goes back to the 1970s: since then people have been trying to formally verify operating systems [4, 10]. It's the obvious place to start when you are serious about meaningful assurance for critical systems. The idea for formal verification is that programs are just mathematics in the end. And if you want to show beyond doubt that something is true in mathematics, you prove it. If you want to be sure that the proof is right, you do it fully formally so that it can be machine-checked.

It was clear early on that this is possible in principle, but enthusiasm ebbed off after an initial flurry of activity around the late '70s and early '80s. Mathematical semantics for real programming languages were not developed far enough, machine support for theorem proving was only starting to appear, and the whole problem seemed infeasible for any real program of interesting size. Full formal program verification was like controlled fusion power: about 30 years of research in the future.

In contrast to controlled fusion, 30 years later things have changed. With the formal verification of the seL4 microkernel we have reached an important milestone: the first commercially viable microkernel, formally verified all the way down to its low-level C implementation [8]. The proof is machine-checked from first principles in the theorem prover Isabelle/HOL [2], and it was an order of magnitude cheaper to build than a traditional software certification.

seL4 is a small microkernel in the L4 family [1]: 8,700 lines of C and 600 lines of assembly. It is not Linux with millions of lines of code. Instead, it provides the basic mechanisms to build an OS: threads, message passing, interrupts, virtual memory, and strong access control with capabilities. Network, file systems, and device drivers are implemented in user space in microkernel systems and it has been shown that this can be achieved with high performance.

Our proof does precisely what the dream of the '70s was: we define formally in an abstract specification what it means for the kernel to be correct. We describe what it does for each input (trap instruction, interrupt, etc), but not necessarily how it is done. Then we prove mathematically that the C implementation always correctly implements this specification.

The proof goes down to the level of the C implementation and assumes the correctness of things below that level: compiler, linker, assembly code, hardware, low-level cache management and TLB. We also assume correctness of the boot code. This is still a long list, but each proof has to stop somewhere. This is what we picked. With more resources, it is possible to eliminate almost all of the assumptions above. There are, for instance, a number of recent research projects on verified optimising C compilers.

We did not only prove the code correct, we also did extensive design proofs and validation.

[1]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.

## What does this mean for security?

In a sense, functional correctness is one of the strongest properties you can prove about a system: you have a precise formal prediction of how the kernel behaves in all possible situations for all possible inputs. If you are interested in a more specific property and you can express this property in Hoare logic, it is now enough to work with this formal prediction, with the specification. This is orders of magnitude easier than direct proofs on the implementation.

Does this mean that we have proved seL4 is secure? Not yet, really. We proved that seL4 is functionally correct. *Secure* would first need a formal definition, which in itself is a wide field and depends on what you want to use the kernel for. Taken seriously, security is a whole-system question, including the system's human components. Nevertheless, there are many different formal security properties of the kernel such as access control, confidentiality, integrity, and availability that one might be interested in. We do have a high-level model for seL4 access control with a nice confinement theorem about it, and we are currently busy connecting up this model with the proven specification. Other properties, for instance secrecy, do not necessarily connect that easily, but we will be looking into that as well in the future.

Even without proving specific security properties on top, a functional correctness proof already has interesting implications for security. If you sit back a minute and think about what it means that we can always predict what the system is going to do, then a number of things come to mind. If the assumptions from above are true, then in seL4 we will have:

- **no code injection attacks.** If we always know precisely what the system does, and if the spec doesn't explicitly allow it, then we can't have any foreign code executing as part of seL4. Ever.

- **no buffer overflows.** This is mainly a classic vector for code injection, but buffer overflows may also inject unwanted data and influence kernel behaviour that way. We prove that all array accesses are within bounds and we prove that all pointer accesses are well typed, even if they go via casts to void and arcane address arithmetic. Buffer overflows are not going to happen.

- **no NULL pointer access.** Few things crash a kernel more nicely or are easier to exploit: see for instance a recent bug in the Linux kernel believed to affect all versions since May 2001 [9]. The bug allows local privilege escalation and execution of arbitrary code in kernel mode, and it's a classic NULL pointer dereference. We have these as direct proof obligation for every pointer access in the system. None of them occur in seL4.

- **no ill-typed pointer access.** Even though the kernel code deliberately breaks C type safety for efficiency at some points, in order to predict that the system behaves according to specification, we have to prove that circumventing the type system is safe at all these points. We cannot get unsafe execution from any of these operations.

- **no memory leaks.** There are no memory leaks in seL4 and there is never any memory freed that is still in use. This not purely a consequence of the proof itself. Much of the design of seL4 was focussed on explicit memory management and it is one of the kernel's more innovative features. Users may run out of memory, but the kernel never will. The kernel also provides an availability property for users (this one we have not yet explicitly proved): if you have set up your memory resources correctly, other users will not be able to starve you of that memory or of the kernel memory resources necessary to back your meta-data. This is by far not true for other kernels, even in the L4 family.

- **no non-termination.** We have proved that all kernel calls terminate. This means the kernel will never suddenly freeze and not return from a system call. This does not mean that the whole system will never freeze, you can still write bad device drivers and bad applications, but if you set it up right, you can always stay in control of run-away processes.

- **no arithmetic or other exceptions.** The C standard defines a long list of things that can go wrong and that you should not be doing: shifting machine words by a too-large amount, dividing by zero, etc. Our framework makes it a specific obligation for us to prove that these do not occur. We have solved all of them. We're also taking special care with overflowing integer arithmetic.

- **no unchecked user arguments.** All user input is checked and validated. If the kernel receives garbage or malicious packages it will respond with the specified error messages, not with crashes. Of course, it is still possible to shoot yourself in the foot. For instance, if you have enough privileges, the kernel happily allows a thread to kill itself. It will never allow anything to crash the kernel, though.

Many of these are general security traits that are good to have for any kind of system. We have also proved a large number of properties that are specific to this kernel. We have proved them about the kernel design and specification. With functional correctness, we know they are true about the code as well. Some examples are:

- **Aligned objects.** Two simple low-level invariants of the kernel are: all objects are aligned to their size, and no two objects overlap in memory. This makes comparing memory regions for objects very simple and efficient.
- **Wellformed data structures.** Lists, doubly linked, singly linked, with and without additional information, are a pet topic of formal verification. These data structures also occur in seL4 and we proved the usual properties: lists are not circular when they shouldn't be, back pointers point to the right nodes, insertion, deletion etc, works as expected and doesn't introduce any garbage.
- **Algorithmic invariants.** Many optimisations rely on certain properties being always true, so specific checks can be left out or can be replaced by other, more efficient checks. A simple example is that the distinguished idle thread is always in thread state *idle* and therefore can never be blocked or otherwise waiting for IO. This can be used to remove checks in the code paths that deal with the idle thread. If the state invariant wasn't true, not having explicit cases for other thread states would easily lead to kernel crashes.
- **Correct book-keeping.** This one was much more interesting and consists of a large collection of individual properties. The seL4 kernel has an explicit user-visible concept of keeping track of memory, who has access to it, who access was delegated to and what needs to be done if a privileged process wants to revoke access from a whole collection of delegates. It is the central mechanism for re-using memory in seL4. The data structure that backs this concept is correspondingly complex and its implications reach into almost all aspects of the kernel. For instance, we proved that if a live object exists anywhere in memory, then there exists a node (an explicit capability actually) in this data structure that covers the object. And if such a capability exists, then it exists in the proper place in the data structure and has the right relationship towards parents, siblings and descendants within. Also, if an object is live (may be mentioned in other objects anywhere in the system) then the object itself together with that capability must have recorded enough information to reach all objects that refer to it (directly or indirectly). Together with a whole host of further invariants, these properties allow the kernel code to reduce the complex, system-global test whether a region of memory is mentioned anywhere else in the system to a quick, local pointer comparison that takes next to no time to execute.

We have proved about 80 such invariants on the low-level design such that they directly transfer to the data structures used in the C program.

The key condition in all this is *if the assumptions above are true*. To attack any of these properties, this is where you would have to look. What the proof really does is take 7,500 lines of C code out of the equation and reduce possible attacks and the human analysis necessary to guard against them to the remaining bits. It is not an absolute guarantee or a silver bullet, but it is definitely a big deal.

## What about type safe languages and static analysis?

A frequent question is whether the same couldn't be achieved by full coverage testing, by using a type safe language, or by static analysis.

It is almost customary in verification papers to bash testing as not sufficient to show the absence of bugs and therefore being useless. I'm not going to do that here. Used right and in combination with other techniques, testing is an effective method to get reliable software. After all, planes do not fall out of the sky all the time because of implementation errors. There have been software-related incidents, but to the best of my knowledge planes still are the safest mode of transportation available. People *can* build reliable software. Testing is just very hard and very expensive to get complete. There are no easy measures for it. For example, if you have the very simple fragment of C code `if (x < y) z = x/y else z = y/x` for `x`, `y`, and `z` being `int` and you test with `x=4,y=2` and `x=8,y=16`, you have full code coverage, every line is executed at least once, every branch of every condition is taken at least once, and you still have two bugs remaining. Of course, any human tester will immediately spot that you should test for something like `x=0,y=-1` and `x=-1,y=0`, but for bigger non-trivial programs there is no easy way to find these cases and it is pretty much infeasible to be sure you have all of them. Humans are good at ingenuity and creativity, they are not so good at repetitive, complex, high-detail tasks. Especially not under pressure. So anything that can reduce the burden should be used. Our style of formal verification on the other hand is very good at completeness. It's what it's all about. It will force you to work through all the relevant cases and it will tell you what the relevant cases are. And because humans are bad at repetitive tasks, we have machine assistance and machine checking of the proofs.

As I said above, the verification takes the C implementation out of the picture. You now only have to test the models and the specification against your expectations. The C model can be reused for any verification, so that cost amortises quickly. Testing the specification was a big part of our development and design process and is a lot easier than testing the implementation.

Similarly, type safe languages as used in Singularity [7] for instance are good. They help. They prevent you from doing lots of stupid things right from the start. If you can, you should use them. But they will usually not safe you from the effects of a NULL pointer dereference, for instance. Sure, an unexpected NULL pointer access will be checked, caught, and reported as an exception. But in an OS, then what? You may fail gracefully instead of catastrophically, but even that may be hard to do if you have progressed way after argument checking and have already changed parts of the state. The difference is that seL4 will just not access NULL pointers. Period.

Type safe languages often require a complex runtime of 'dirty' code that needs to be trusted. Singularity's trusted code base is larger than the whole seL4 kernel, so there is no real win in terms of easier verification. And despite Singularity eliminating the need for context switches, you still pay overhead for runtime checks. As a result, L4 kernels are still faster. [6]

Static analysis can do even more than most type safe languages. Some parts of the security-relevant properties that are implications and by-products of our proof are covered in theory by a number of static analysis tools. The advantage and the problem with static analysis is that it is automatic. It cannot be safe and complete at the same time, otherwise it would solve the halting problem. So if it is safe, then it will have false alarms and instances of *maybe correct* instead of *definitely correct.* The functional correctness property we proved is way too hard for static analysis. Even the by-products are still too hard. Some instances of pointer dereferences in the code are safe for deep reasons of the underlying algorithm. You would have to add redundant explicit checks into the code to make them go through with static analysis. This is precisely what you don't want for a high-performance OS kernel. Humans, constructing an interactive, machine-checked proof, on the other hand, have no problem solving the halting problem and conducting a fully precise analysis.

## So, did you find any bugs?

We didn't test the kernel extensively before verification started, but we did quite a bit of debugging initially and we did use it for student projects internally for more than 6 months and ported it to a different architecture. After initial debugging, these activities found 16 bugs in this internal alpha-release. After that, the kernel ran just fine for everything the students wanted to do.

We also ran a static analysis tool on the code before verification. We found two bugs (counted in the above 16) and got hundreds of false positives.

Formal verification then found 144 more bugs in the C code, in total 160. This means even though the code appeared to be running just fine for normal application, there were an order of magnitude more bugs lurking in there than the 16 found initially. They were mostly but not exclusively in rarely used features. As mentioned, we also did proofs on the design and the specification level. This was to 95% before the code was written, and we fixed about 150 issues in each. That means, in total we have discovered roughly 460 issues in kernel code, design and specification.

None of the bugs found in the C verification stage were deep in the sense that the corresponding algorithm was flawed. These were already caught in the design validation phase. Some examples are missing checks on user supplied input or subtle side effects in the middle of an operation breaking global invariants. The bugs discovered in the C code were mainly typos, misreading the specification, or failing to update all relevant code parts for specification changes. Even though their cause was often simple, understandable human error, their effect in almost all cases was sufficient to crash the kernel or create security vulnerabilities. Other more interesting bugs found during the C implementation proof were missing exception case checking, and different interpretations of default values in the code. For example, the interrupt controller on ARM returns 0xFF to signal that no interrupt is active which is used correctly in most parts of the code, but in one place the check was against NULL instead.

## Do I need a team of PhDs for this?

Formal verification is thought of as high-effort, expensive, and needing a large team of highly qualified experts. Our project shows that things are by far not as bad, especially compared to other high-assurance methods. After the industry rule of thumb of $10k/loc, Common Criteria EAL6 certification of seL4 would have cost about $87 Million.

If we overestimate our effort with 30 person years and if we overestimate our fully loaded salary with $200k/year per person, we get $6M spent. Even if you take into account that CC takes more than just providing evidence that the design and code works, our proof provides higher assurance than what EAL7 officially requires (EAL7 requires only formal design proofs, no formal implementation proofs). And it does that for an order of magnitude less money.

So yes, it is still considerable effort, but it is in the same order of magnitude of normal, good quality design and implementation, not in the prohibitively expensive class any more.

Do you need a team of PhDs for this? We didn't. Most of the verification engineers in the project did not have a PhD. Some, but not all where PhD students. Many were never involved in theorem proving before. They were university graduates and they are certainly very smart people, but they learned machine-checked theorem proving on the job. This attests that modern theorem proving tools like Isabelle/HOL are mature enough to be used, even if they can undoubtedly still be improved. You will probably want at least one expert with previous experience in the team and you will definitely want domain experts like OS designers in the team, but formal foundational verification on real code with about 10,000, maybe 50,000 loc, for full functional correctness is definitely possible today with today's technology and tools. It's fun, too. More people should get into it.

## References

[1] L4 microkernel. http://l4hq.org.

[2] The Isabelle theorem prover. http://isabelle.in.tum.de/, 2009.

[3] The L4.verified project. http://ertos.nicta.com.au/research/l4.verified/, 2009.

[4] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, pages 329–334, New York, NY, USA, Jun 1979.

[5] G. Heiser. Your system is secure? prove it! *USENIX ;login:*, 32(6):35–38, Dec 2007.

[6] G. Heiser. Q: What is the difference between a microkernel? http://www.ok-labs.com/blog/entry/singularity-vs-l4-microkernel-performance/, 2009.

[7] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM Operat. Syst. Rev.*, 41(2):37–49, 2007.

[8] 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*, Big Sky, MT, USA, Oct 2009. ACM.

[9] T. Ormandy and J. Tinnes. Linux null pointer dereference due to incorrect proto_ops initializations. http://www.cr0.org/misc/CVE-2009-2692.txt, 2009.

[10] G. J. Popek, M. Kampe, C. S. Kline, and E. Walton. UCLA data secure Unix. In *AFIPS Conference Proceedings: 1979 National Computer Conference (1979 NCC)*, pages 355–364. AFIPS Press, 1979.