There is now a way to mathematically prove that the software governing critical safety and security systems in aircraft and motor vehicles is free of a large class of errors - long before the plane takes off or the car's engine starts.

All thanks to researchers at NICTA, Australia’s Information and Communications Technology (ICT) Research Centre of Excellence, who have completed the world’s first formal machine-checked proof of a general-purpose operating system kernel.

The Secure Embedded L4 (seL4) microkernel itself, which has been designed for real-world use, has potential applications in defence and other safety and security industries where the flawless operation of complex embedded systems is of critical importance.

'It is hard to comment on this achievement without resorting to clichés,' said Prof Lawrence Paulson at Cambridge University’s Computer Laboratory. 'Proving the correctness of 7,500 lines of C code in an operating system's kernel is a unique achievement, which should eventually lead to software that meets currently unimaginable standards of reliability.'

'Formal proofs for specific properties have been conducted for smaller kernels, but what we have done is a general, functional correctness proof, which has never before been achieved for real-world, high-performance software of this complexity or size,' explained NICTA principal researcher Dr Gerwin Klein, who leads NICTA's formal verification research team.

The proof also shows that many kinds of common attacks will not work on the seL4 kernel. For instance, the microkernel is impervious to buffer overflows, a common form of software attack where hackers take control of programs by injecting malicious code. 'Our seL4 kernel cannot be subverted by this kind of attack,' said Dr Klein.

The outcome is the result of four years' research by Dr Klein's team of 12 NICTA researchers, NICTA/UNSW PhD students and UNSW contributed staff. They successfully verified the C code and proved more than 10,000 intermediate theorems in more than 200,000 lines of formal proof. The proof is machine checked using the interactive theorem-proving program Isabelle. It is one of the largest machine-checked proofs ever done.

To reach this milestone, the NICTA team invented new techniques in formal machine-checked proofs, made advances in the mathematical understanding of real-world programming languages and developed new methodologies for rapid prototyping of operating system kernels.

'This work goes beyond the usual checks for the absence of certain specific errors,' Prof Paulson said. 'Instead, it verifies full compliance with the system specification. The project has yielded not only a verified microkernel but a body of techniques that can be used to develop other verified software.'

NICTA will shortly transfer its intellectual property to NICTA spin-out company Open Kernel Labs, whose embedded hypervisor software - also based on NICTA research - is in millions of consumer devices worldwide.

A scientific paper describing this research will appear in the 22nd ACM Symposium on Operating Systems Principles (SOSP) here. Further details about NICTA's L4 verified research project can be found here.
The L4.verified project
A Formally Correct Operating System Kernel


In current software practice it is widely accepted that software will always have problems and that we will just have to live with the fact that it may crash at the worst possible moment: You might be on a deadline. Or, much scarier, you might be on a plane and there's a problem with the board computer.

Now think what we constantly want from software: more features, better performance, cheaper prices. And we want it everywhere: in mobile phones, cars, planes, critical infrastructure, defense systems.

What do we get? Mobile phones that can be hacked by SMS. Cars that have more software problems than mechanical ones. Planes where computer problems have lead to serious incidents. Computer viruses spreading through critical infrastructure control systems and defense systems. And we think "See, it happens to everybody."

It does not have to be that way. Imagine your company is commissioning a new vending software. Imagine you write down in a contract precisely what the software is supposed to do. And then — it does. Always. And the developers can prove it to you — with an actual mathematical machine-checked proof.

Of course, the issue of software security and reliability is bigger than just the software itself and involves more than developers making implementation mistakes. In the contract, you might have said something you didn't mean (if you are in a relationship, you might have come across that problem). Or you might have meant something you didn't say and the proof is therefore based on assumptions that don't apply to your situation. Or you haven't thought of everything you need (ever went shopping?). In these cases, there will still be problems, but at least you know where the problem is not: with the developers. Eliminating the whole issue of implementation mistakes would be a huge step towards more reliable and more secure systems.

Sounds like science fiction?

The L4.verified project demonstrates that such contracts and proofs can be done for real-world software. Software of limited size, but real and critical.

We chose an operating system kernel to demonstrate this: seL4. It is a small, 3rd generation high-performance microkernel with about 8,700 lines of C code. Such microkernels are the critical core component of modern embedded systems architectures. They are the piece of software that has the most privileged access to hardware and regulates access to that hardware for the rest of the system. If you have a modern smart-phone, your phone might be running a microkernel quite similar to seL4: OKL4 from Open Kernel Labs.

We prove that seL4 implements its contract: an abstract, mathematical specification of what it is supposed to do.
The Proof

This page defines in high-level language what precisely we are proving, what we assume, and what the proof implies. It is aimed at an audience with a technical background, but does not assume any expertise in formal verification.

What we prove

Formal proofs can be tricky. They prove exactly what you have stated, not necessarily what you mean or what you want.

Our proof statement in high-level natural language is the following:

**The C code of the seL4 microkernel correctly implements the behaviour described in its abstract specification and nothing more.**

This doesn't sound like much, but it is a very strong formal statement, called functional correctness. A cynic might say: proofs like this only show that every fault in the specification has been precisely implemented in C. This is true, the above statement does not exclude this problem. But: at some point you have to say what you want (the "abstract specification") and that is what you get.

The idea is that it is much easier and quicker to say in the specification what you want, because the language used to express it is more powerful (formal, higher-order logic in this case) and because you can leave out detail you do not care about: the specification only needs to say what the software does, not how it is done.

There is another reason implementation proofs like this give you a strong return in terms of software trustworthiness: they imply the absence of large classes of common programming errors. More on this below.

And finally, for the experts, proofs like this are valuable, because they enable you to prove further things about the software much more quickly and easily: for a very large class of properties, it is now enough to prove that the property holds on the specification. These additional proofs are a further way to make sure the specification states what you mean, not just what you say. Our correctness proof then guarantees that the same property also holds for the C implementation of the kernel without any further work to be done.

This is just the beginning. We plan to make good use of this last point in the future and we also plan to make use of it for proving the correctness of applications running on top of the seL4 microkernel. Without a verified OS kernel, verified applications stand on shaky ground.

What we assume

With a proof in formal logic, it is important to understand what its basic assumptions are, because this is where fault can still occur. Our proof about the seL4 microkernel goes down to the level of the C programming language.

- **Compiler/linker:** we assume that the compiler has translated this specific program (seL4) correctly into ARM11 assembly code (according to our formalisation of C and the C standard) and that the linker lays out this code correctly in memory.
- **Assembly:** the seL4 kernel, like all operating system kernels, contains some assembly code, about 600 lines in our case. For seL4, this concerns mainly
entry to and exit from the kernel, as well as direct hardware accesses. For the proof, we assume this code is correct.

- **Hardware**: we assume the hardware works correctly.
- **Hardware management**: the proof tries to make only the most minimal assumptions on the underlying hardware. It abstracts from cache consistency, cache colouring and TLB (translation lookaside buffer) management. The proof assumes these functions are implemented correctly in the assembly layer mentioned above and that the hardware works as advertised. The proof also assumes that especially these three hardware management functions do not have any effect on the behaviour of the C program. This is only true if they are used correctly.
- **Boot code**: the proof currently is about the operation of the kernel after it has been loaded correctly into memory and brought into a consistent, minimal initial state. This leaves out about 1,200 lines of the code base that a kernel programmer would usually consider to be part of the kernel.
- **Virtual memory**: virtual memory is the hardware mechanism that the kernel uses to protect itself from user programs and user programs from each other. Virtual memory makes life complicated for the proof, because it can change how memory is laid out and how the kernel accesses it. While our proof does talk about the concept of virtual memory and does verify the virtual memory subsystem of the kernel that is offered to user programs, the execution model of the kernel itself and its own access to memory is not up to our usual standards of reasoning from first principles like the rest of our proof. Instead we assume a certain standard behaviour of memory, and prove the necessary conditions that ensure these assumptions are true or at the very least we informally make sure they are implied by what we have proved so far. The thing is: you have to trust us that we got all necessary conditions and that we got them right. Our machine-checked proof doesn't force us to be complete at this point. In short, in this part of the proof, unlike the other parts, there is potential for human error. If you only accept end-to-end machine-checked proof as true and don't trust humans, you might therefore count virtual memory for kernel execution into the assumptions section.

What do these assumptions mean?

The proof assumptions mean that there may be faults remaining in the kernel that could be classified as implementation faults below the level of C. They will not be faults that are properly visible on the level of the C
programming language (which is our main claim of correctness), but they could still be serious faults that make the sel4 kernel misbehave.

We have made these assumptions to fit into the carefully limited scope and the limited resources of a major research project. These specific assumptions are not a limitation of the general formal verification approach. In theory, it is possible to eliminate all of them: there are successful verification projects showing the correctness of even optimising C compilers; there are at least two prominent research groups that have demonstrated successful formal verification of assembly code and low-level hardware management functions; we have ourselves proved an earlier version of the boot code correct down to the level of a precise, executable specification; and we have a separate formalisation of ARM11 virtual memory from first principles. There are still significant scientific challenges in unifying all of these into a single framework, but it is clear at this point that it can be done.

With all the purity and strength of mathematical proof it is easy to get carried away. There is a fundamental theoretical limit of formal verification: there will always be some bottom level of assumptions about the physical world left and these assumptions have to be validated by other means. Mathematical proof is proof as long as it talks about formal concepts. It is when assumptions connect to reality where things may still fail. Albert Einstein is quoted as saying "As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality." For instance, if the hardware is faulty, or if a cosmic ray randomly changes memory, the correctness predictions of our proof do not apply. Neither do any traditional tests or verification methods help against cosmic rays, but it is important to keep in mind that even with mathematical proof, there are no absolute guarantees about the physical world.

There are two other assumptions that we do not include in the list above, because they are of a different kind:

- We assume the axioms of higher-order logic are logically consistent.
- We assume our proof is correct.

The first is a fundamental question of formal logic. If this is not true, mathematics in general has a much bigger problem than one verified OS kernel. The second is more interesting, but equally unlikely to be false. A typical conversation about the topic goes like this:

Q: So you say this OS kernel is correct. How do you know?
A: We proved it in formal logic.
Q: How do you know your proof is correct?
A: We use a computer program, a so-called theorem prover to construct and check the proof.
Q: How do you know the theorem prover is correct? Have you verified it?
A: We have not formally verified the theorem prover (you could, but note the potential for an infinite conversation if we said yes here). It is a so-called LCF-system that constrains any correctness-critical problems to a very small part of the prover. We have strong confidence in this small core.
Q: Is there a way to check the proof independently?
A: There is. You can generate a gigantic text file that contains no proof search, nothing complicated, only the axioms and proof steps of the most basic, fundamental kind. You can write a small, independent proof-checking program for that. We haven't done so, but we could.

At the end of the conversation, we are three steps removed from the original problem. Software bugs in the last step may have an impact on the first, but they need to go through all the other steps first and pass by the eyes of the
human verification engineer who spends most of her working time looking at precisely this proof. There is no absolute guarantee that the proof is correct, but when you come right down to it, humans are good at creating proofs, computers are very good at checking them. It is an easy problem for computers. If you are worried about the proof, be worried about the assumptions in the first part. They are much more likely to cause problems.

What the proof implies

We mentioned above that our formal proof of functional correctness implies the absence of whole classes of common programming errors. Provided our assumptions above are true, some of these common errors are:

- **Buffer overflows**: buffer overflows are a classic security attack against operating systems, trying to make the software crash or even to inject malicious code into the cycle. We have proved that no such attack can be successful on seL4.
- **Null pointer dereferences**: null pointer dereferences are another common issue in the C programming language. In applications they tend to lead to strange error messages and lost data. In operating systems they will usually crash the whole system. They do not occur in seL4.
- **Pointer errors in general**: in C it is possible to accidentally use a pointer to the wrong type of data. This is a common programming error. It does not happen in the seL4 kernel.
- **Memory leaks**: memory leaks occur when memory is requested, but never given back. The other direction is even worse: memory could be given back, even though it is still in use. Neither of these can happen in seL4.
- **Arithmetic overflows and exceptions**: humans and mathematics usually have a concept of numbers that can be arbitrarily big. Machines do not, they need to fit them into memory, usually into 32 or 64 bits worth of storage. Machines also generate exceptions when you attempt to do things that are undefined like dividing by zero. In the OS, such exceptions would typically crash the machine. This does not occur in seL4.

We could go on, but it would become a bit boring... There are other techniques that can also be used to find some of these errors. Here, the absence of such bugs is just a useful by-product of the proof. To be able to complete our proof of functional correctness, we also prove a large number of so-called invariants: properties that we know to always be true when the kernel runs. To normal people these will not be exciting, but to experts and kernel programmers they give an immense amount of useful information. They give you the reasons why and how data structures work, why it is OK to optimise and leave out certain checks (because you know they will be always be true anyway), and why the code always executes in a defined and safe manner.

As mentioned above, this is just the beginning. Formal verification has a lot more to offer. Check back for the next research project.

---

**The L4.verified project**

**Creating Trustworthy Software**
The L4.verified project is providing a mathematical, machine-checked proof of the functional correctness of the seL4 microkernel with respect to a high level, formal description of its expected behaviour. The aim is to produce a truly trustworthy, high-performance operating system kernel.

**Approach**

The figure above shows the formal refinement approach L4.verified is taking. The bottom level of the verification and of the refinement picture is a high-performance C and assembly implementation of the seL4 microkernel. The next level up, the low-level design, is a detailed, executable specification of the intended behaviour of the C implementation. This executable specification is derived automatically from a prototype of the kernel, developed in the seL4 project. The prototype was written in the high-level, functional programming language Haskell. The next refinement layer up is the high-level design, an abstract, operational specification of the kernel. It contains less detail and is not directly executable any more, but it still precisely captures the intended kernel behaviour. Finally, on the top-most level, there is an abstract access control model of seL4 that captures how capabilities (the kernel's access control mechanism) are distributed in the system. To the right of this access control model, the picture shows one of the security properties that seL4 can enforce: isolation of security domains.

All proofs in the project, shown as green arrows in the picture, are machine-checked in the interactive theorem prover Isabelle/HOL. The arrow between access control model and high-level design is dotted, because this proof is not part of the L4.verified project and is left for future work.

For a more in-depth overview of the project, please check the publications page. A good start into to the general area is "Operating System Verification -- An Overview", a high-level overview of the verification project can be found in the SOSP'09 paper "seL4: Formal Verification of an OS Kernel".

**Status**
The project has successfully completed all of the specification artefacts shown in the picture and all of the proofs denoted by solid arrows: the security proof, the refinement towards the low-level design, and the refinement down to the C implementation.

Outcomes
The project has achieved a number of significant research outcomes. Some highlights are:

- An implementation correctness proof between low-level design and C implementation for a code base of similar size or complexity has never been achieved before.
- The implementation correctness proof between high-level and low-level design alone already makes the seL4 kernel, to our knowledge, the best, and most deeply analysed high-performance microkernel in the world.
- In producing the seL4 design and implementation together with the seL4 project, we have developed a methodology and tool set for rapid-prototyping of small OS kernels. This methodology has a very short turn-around time for trying out new features and integrates formal specification and verification directly into the development cycle. Developing and testing new kernel features in this method takes on the order of hours and days instead of weeks and months. Developing a new high-assurance kernel-design would not have been possible without it in the available time.
- For the C implementation correctness proof, we developed a very precise formalisation of the C programming language and its memory model. The formalisation includes low-level and unsafe C programming constructs like pointer arithmetic and pointer cast. Harvey Tuch, the main investigator and PhD student on this part, recently won the CISRA PhD award for this model which allows us to reason abstractly and efficiently about these C features that are usually outside the scope of verification projects, but are required for microkernel performance optimisations.

Commercialisation
The research outcomes of the project are set to be commercialised in the NICTA spinout company Open Kernel Labs. Stay tuned for the first commercially available, fully formally verified microkernel.
What does a formal proof look like?

This document shows a small example of a formal, machine-checked proof.

The example we pick is the proof for a theorem of standard mathematics from Freek Wiedijk’s compilation *The Seventeen Provers of the World* [2]. We show that the size of the diagonal of a square of side 1 is not rational, i.e., that it cannot be written as the division of an integer by another integer. Remember that according to Pythagoras, the size of the diagonal $d$ of a 1 by 1 square is $\sqrt{2}$, because $d^2 = 1^2 + 1^2$.

A mathematician might phrase this theorem as follows.

**Theorem 1** $\sqrt{2}$ is irrational: there are no integers $a$ and $b$ such that $\sqrt{2} = \frac{a}{b}$, where the fraction $\frac{a}{b}$ is irreducible.

A fraction is irreducible if it is in its simplest form (e.g. $2/4$ and $50/100$ both represent the same irreducible fraction $1/2$). An irreducible fraction $\frac{a}{b}$ is defined by the fact that $a$ and $b$ are relatively prime, or coprime, i.e., that there is no number diving both $a$ and $b$ other than 1 (denoted $gcd(a, b) = 1$, where $gcd$ stands for the greater common divisor).

**Informal proof**

Several different proofs exist for this theorem. One of the easiest one is using the concept of proof by contraposition: we prove that $\sqrt{2}$ is irrational by proving that if it was rational, then we would have a contradiction.

So we first assume that $\sqrt{2}$ is rational. This means that we have two integers $a$ and $b$ such that $\sqrt{2} = \frac{a}{b}$, where the fraction $\frac{a}{b}$ is irreducible. We therefore have that $a^2 = 2b^2$. Now, by definition of even numbers, this gives that $a^2$ is even, and this implies that $a$ itself is even (as the square of an odd number is odd). So there exists a $k$ such that $a = 2k$. If we replace $a$ by $2k$ in our original equation, we obtain $4k^2 = 2b^2$, which implies that $b^2 = 2k^2$, meaning that $b^2$, and therefore $b$ itself are also even.

Here we reach the contradiction because we have $a$ and $b$ both even, which contradicts the fact that $\frac{a}{b}$ is irreducible ($2$ divides both $a$ and $b$ so $gcd(a, b) \neq 1$). Our first assumption was thus false: $\sqrt{2}$ cannot be rational.

**Formal proof in Isabelle/HOL**

The fully formal, machine-checked proof of the same theorem in the prover Isabelle/HOL [1] is a bit longer with more detail, but follows the same line of reasoning.

We first prove a more general lemma stating that the square root of any prime number is irrational, and use it to prove that in particular the square root of the prime number 2 is irrational.

In the machine version we write $\sqrt{q}$ for $\sqrt{q}$ and we explicitly convert between natural numbers and real number using the function $real$. The proof is a formal statement followed by a sequence of commands to the theorem prover that are phrased in such a way that they are still readable by humans (or at least human experts).
The nicely written-out proof below is due to Makarius Wenzel.

**Theorem** sqrt-prime-irrational:
assumes prime p
shows sqrt (real p) /∈ Q

**Proof**
from ⟨prime p⟩ have p: 1 < p by (simp add: prime-def)
assume sqrt (real p) ∈ Q
then obtain m n where
  n: n ≠ 0 and sqrt-rat: |sqrt (real p)| = real m / real n
  and gcd: gcd (m, n) = 1 ..
have eq: m² = p * n²
proof –
from n and sqrt-rat have real m = |sqrt (real p)| * real n by simp
then have real (m²) = (sqrt (real p))² * real (n²)
  by (auto simp add: power2-eq-square)
also have (sqrt (real p))² = real p by simp
also have ... * real (n²) = real (p * n²) by simp
finally show thesis ..
qed
have p dvd m ∧ p dvd n
proof
from eq have p dvd m² ..
with ⟨prime p⟩ show p dvd m by (rule prime-dvd-power-two)
then obtain k where m = p * k ..
with eq have p * n² = p² * k² by (auto simp add: power2-eq-square mult-ac)
with p have n² = p * k² by (simp add: power2-eq-square)
then have p dvd n² ..
with ⟨prime p⟩ show p dvd n by (rule prime-dvd-power-two)
qed
then have p dvd gcd (m, n) ..
with gcd have p dvd 1 by simp
then have p ≤ 1 by (simp add: dvd-imp-le)
with p show False by simp
qed

**Corollary** sqrt (real (2::nat)) /∈ Q
by (rule sqrt-prime-irrational) (rule two-is-prime)

We are not always patient enough to explain all details to the human reader, we
sometimes switch to a more machine-oriented style. It is quicker to type, but does not
offer much in the way of explanation. This might be fine, though. The computer will
check that the proof is right, and sometimes you only want to know what is proved, not
necessarily how the proof works.

**Theorem** sqrt-prime-irrational: prime p → sqrt (real p) /∈ Q
apply clarsimp
apply (elim rationals-rep)
apply simp
apply (simp add: nonzero-eq-divide-eq)
apply (drule arg-cong [where f = λx. x+x])
apply (simp only: mult-ac)
apply (simp only: power2-eq-square[symmetric])
apply (simp only: real-sqrt-pow2)
apply (simp only: mult-assoc[symmetric])
apply (simp add: power2-eq-square)
apply (simp only: real-of-nat-mult[symmetric])
apply (simp only: real-of-nat-inject)
apply (simp add: power2-eq-square[symmetric] eq-commute)
apply (frule-tac m=m in prime-dvd-power-two, simp)
apply (frule-tac m=n in prime-dvd-power-two)
apply (frule iffD1 [OF dvd-def], clarsimp)
apply (simp add: power2-eq-square mult-ac prime-def)
apply (simp add: eq-commute)
apply (drule (1) gcd-greatest)
apply (thin-tac p dvd n)
apply (drule dvd-imp-le, simp)
apply (clarsimp simp: prime-def)
done

References
