$V \lor \neg V$?

What's verified, what's not, and what does it mean?

June Andronick

https://sel4.systems/Foundation

seL4 FOUNDATION

Overview

"the most verified microkernel" ← true, but requires careful handling to ensure appropriate expectations and valid claims

We should celebrate and promote

the high-assurance

that seL4 and seL4-based systems provide

seL4's verification is its key differentiator

Inappropriate claims will damage seL4 reputation

and could have serious consequences in real systems

It is also important to ensure understanding of:

- **what exactly is proved**
- **the conditions under which the proofs hold**
- **what they imply in practice**

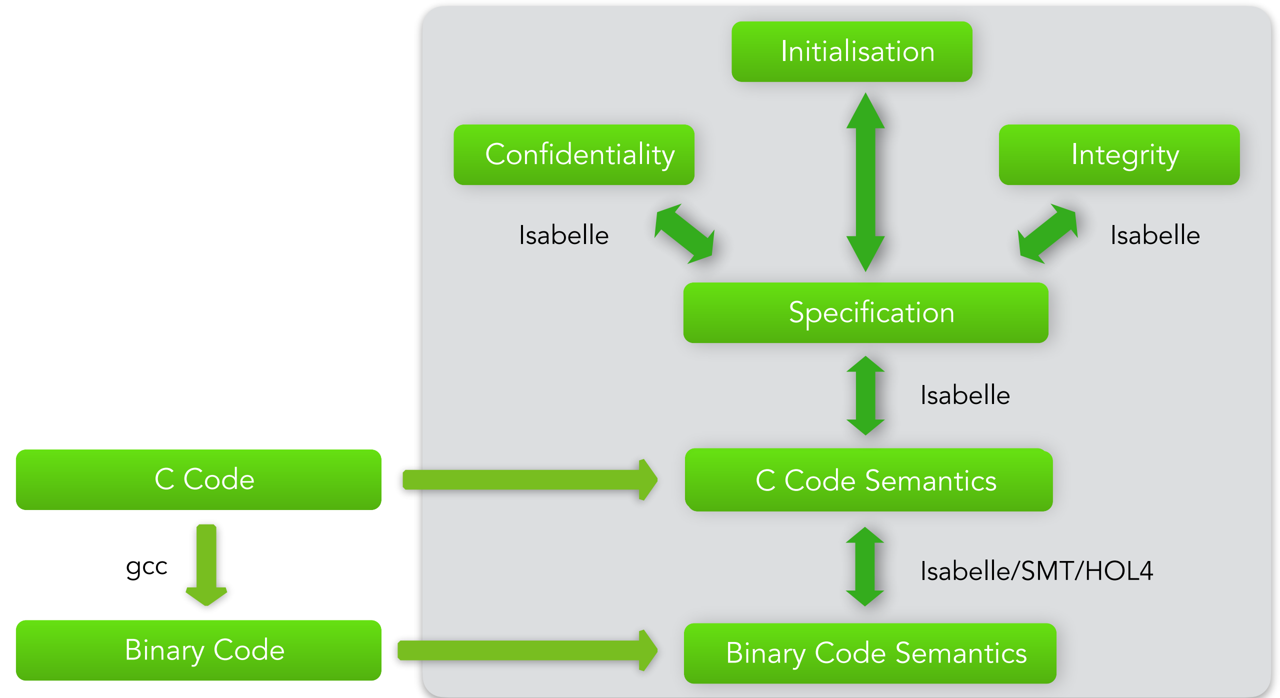
Overview

"the most verified microkernel"

We should celebrate and promote
the high-assurance
that seL4 and seL4-based systems provide

It is also important to ensure understanding of:

- **what exactly is proved**
- **the conditions under which the proofs hold**
- **what they imply in practice**



Formal, machine-checked proof
the seL4 binary is correct with respect to spec
and enforces isolation

Overview

1.
2.
3.
4.

Overview

1. I have download seL4 ⟹ I have a verified kernel

2. My system is built on a verified seL4 configuration ⟹ My system enforces isolation and is verified

3. Verified ⟹ 0 bugs

4. “Verified” ⟹ formal, machine-checked proof of strong properties down to the binary

Overview

1. I have downloaded seL4 $\not\Longrightarrow$ I have a verified kernel

   Not all seL4 configurations are verified

2. My system is built on a verified seL4 configuration $\overset{?}{\Longrightarrow}$ My system enforces isolation and is verified

3. Verified $\overset{?}{\Longrightarrow}$ 0 bugs

4. “Verified” $\overset{?}{\Longrightarrow}$ formal, machine-checked proof of strong properties down to the binary

Overview

1.	I have downloaded seL4	$\not\Longrightarrow$	I have a verified kernel	Not all seL4 configurations are verified
2.	My system is built on a verified seL4 configuration	$\not\Longrightarrow$	My system enforces isolation and is verified	System design and initialisation is key
3.	Verified	$\overset{?}{\Longrightarrow}$	0 bugs	
4.	“Verified”	$\overset{?}{\Longrightarrow}$	formal, machine-checked proof of strong properties down to the binary	

Overview

1.	I have downloaded seL4	⇏	I have a verified kernel	Not all seL4 configurations are verified
2.	My system is built on a verified seL4 configuration	⇏	My system enforces isolation and is verified	System design and initialisation is key
3.	Verified	⇏	0 bugs	Proofs have assumptions and scope
4.	"Verified"	? ⟹	formal, machine-checked proof of strong properties down to the binary	

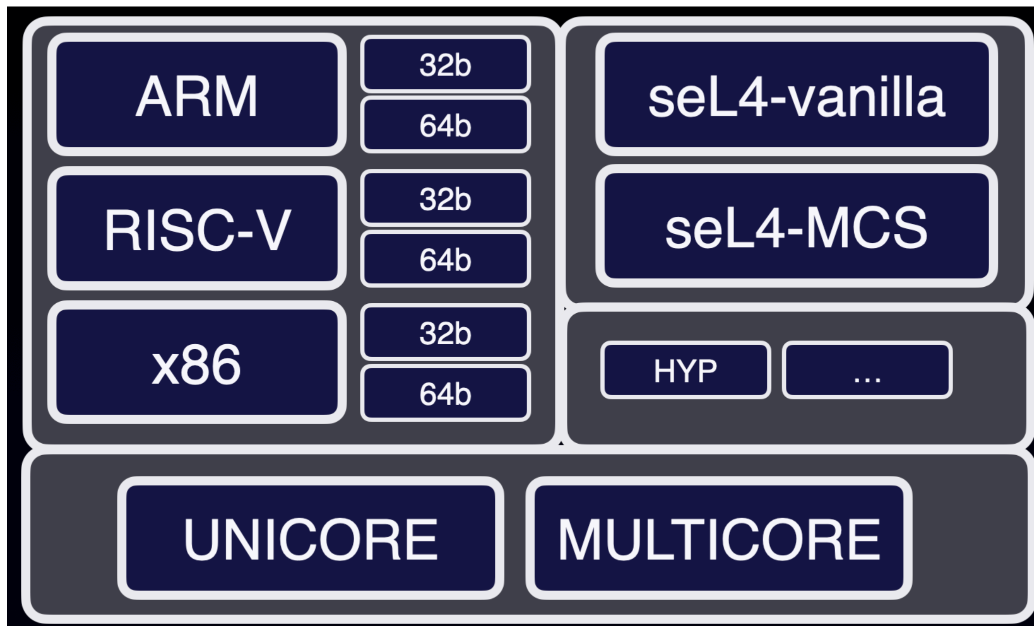
Overview

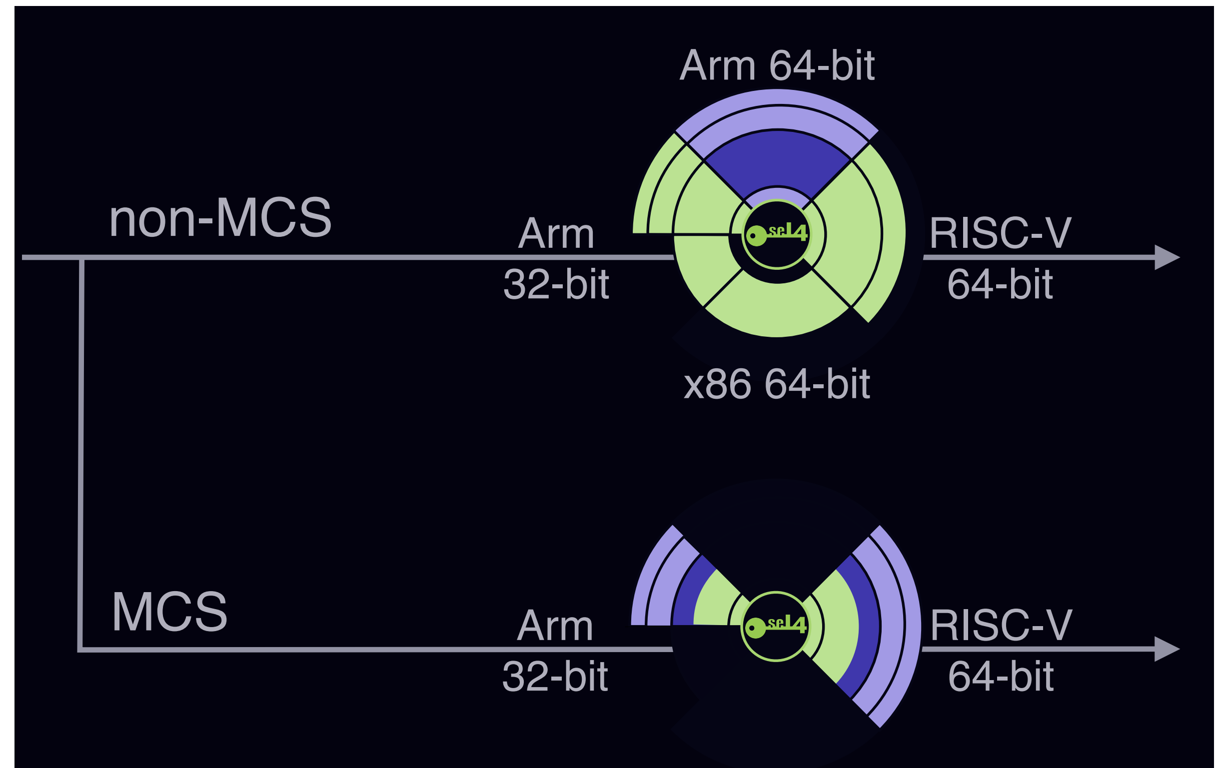
1.	I have downloaded seL4	$\not\Rightarrow$	I have a verified kernel	Not all seL4 configurations are verified
2.	My system is built on a verified seL4 configuration	$\not\Rightarrow$	My system enforces isolation and is verified	System design and initialisation is key
3.	Verified	$\not\Rightarrow$	0 bugs	Proofs have assumptions and scope
4.	"Verified"	$\not\Rightarrow$	formal, machine-checked proof of strong properties down to the binary	Beware of claims from other systems

Know which configurations are verified

1. I have downloaded seL4 ⇏ I have a verified kernel

Not all seL4 configurations are verified



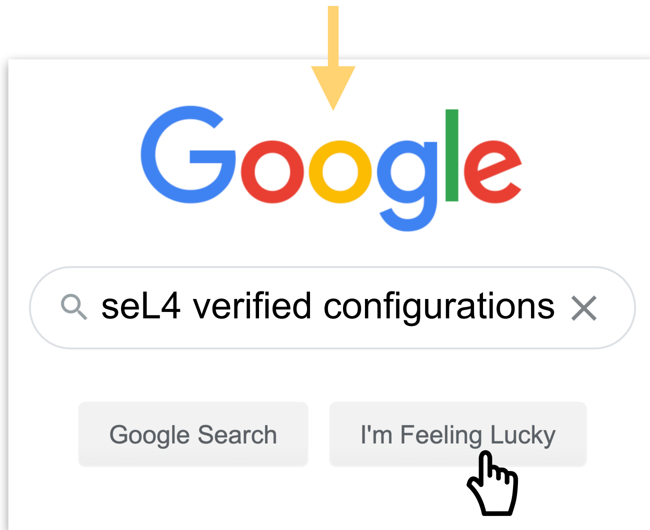


Know which configurations are verified

1. I have downloaded seL4 ⇏ I have a verified kernel

Not all seL4 configurations are verified

How do I know?



Verified Configurations

This page describes which architecture/platform/configuration combinations of seL4 have verified properties, which configurations possess which properties, and how to obtain an seL4 version for a specified configuration.

At this time, verification of seL4 remains a more time-intensive process than software development. Consequently, while seL4 has been ported to multiple architectures, and its build system allows further configuration of internal and hardware features, verified configurations are necessarily both less numerous and more specific.

These configurations are also referred to as *verification platforms*, currently constituting: AARCH64, ARM, ARM_HYP, X64, RISCV64, ARM_MCS, RISCV64_MCS

Please consult Frequently Asked Questions, as well as the proof and assumptions page for a better understanding of the intersection of verification and seL4.

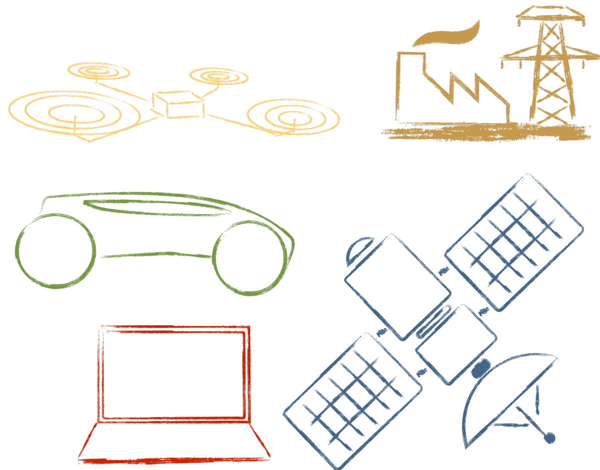
ARM

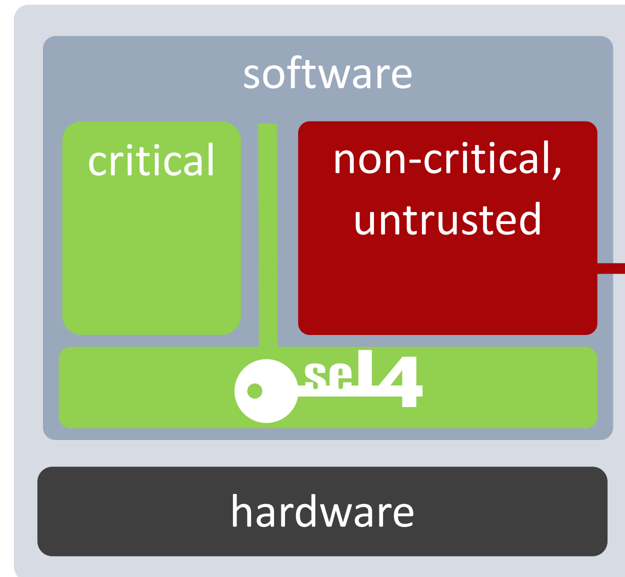
File	`ARM_verified.cmake`
Architecture	ARMv7
Platform	iMX.6 (e.g. Sabre Lite)
Floating-point support	No
Hypervisor mode	No
Verified properties	functional correctness incl fast path, integrity (access control), confidentiality (information flow), binary correctness (covers all verified C code), user-level system initialisation

Design and initialise your system correctly

2. My system is built on a verified seL4 configuration My system enforces isolation and is verified

System design and initialisation is key





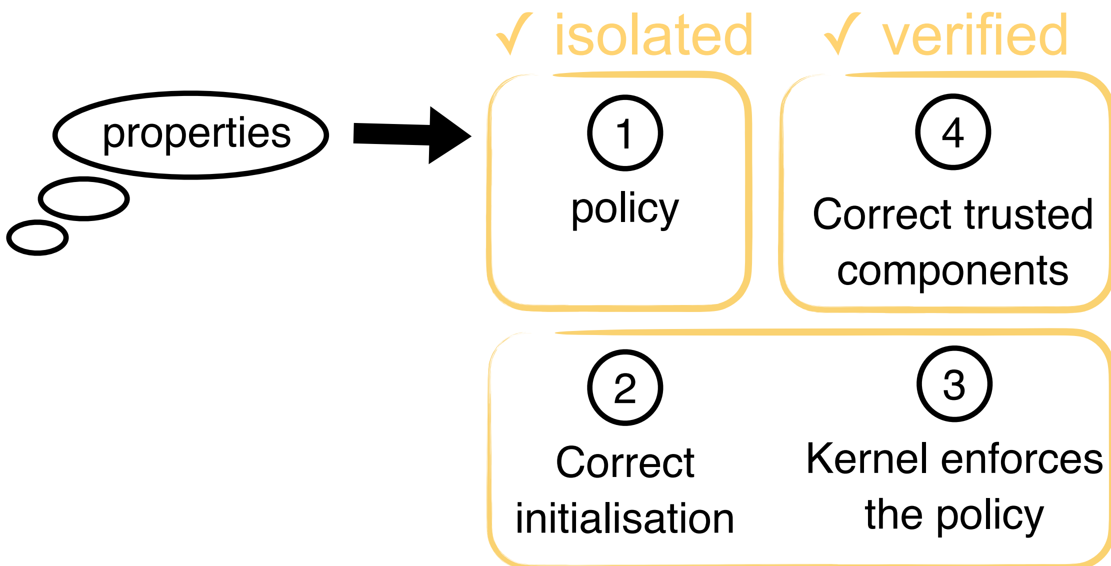
Design and initialise your system correctly

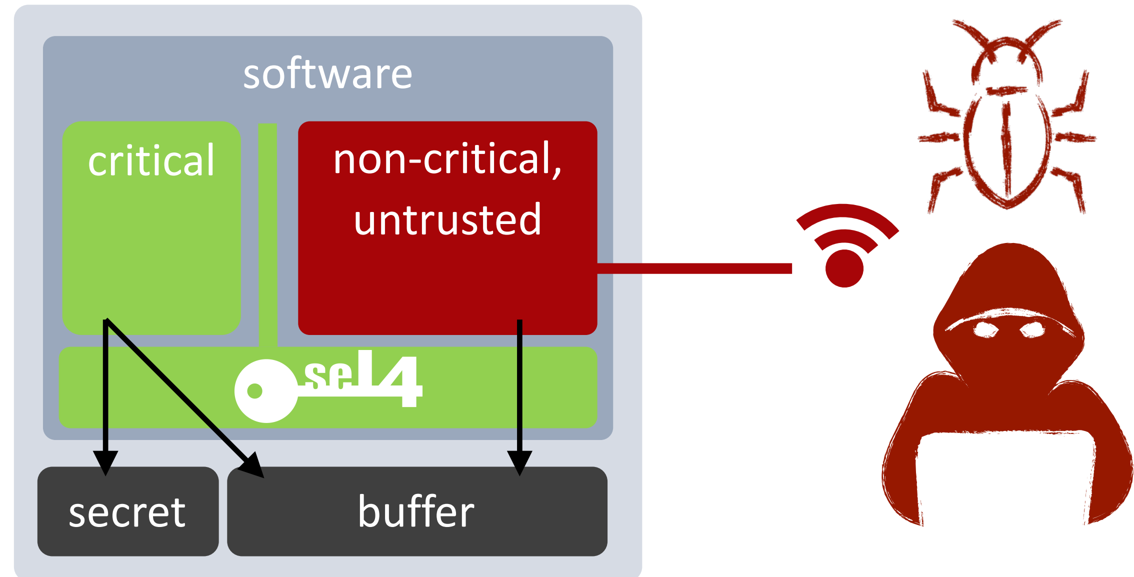
2. My system is built on a verified seL4 configuration My system enforces isolation and is verified

System design and initialisation is key

How? → *Componentise, Isolate, Prove*

Minimised **verified** TCB





✓ if using a fully verified seL4 config

Know the scope and assumptions of the proofs

seL4 FOUNDATION

3. Verified 0 bugs

Proofs have assumptions and scope

How do I know?

seL4 faq

Google Search | I'm Feeling Lucky

What are the proof assumptions?

The brief version is: we assume that the few lines of in-kernel assembly code are correct, hardware behaves correctly, in-kernel hardware management (TLB and caches) is correct, and boot code is correct. The hardware model assumes DMA to be off or to be trusted. The security proofs additionally give a list of conditions how the system is configured.

For a more in-depth description, see the proof and assumptions page.

Know the scope and assumptions of the proofs

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 languag

The binary code of the ARM and RISCV64
implement the behaviour described in its
Furthermore, the specification and the se
called *integrity* and *confidentiality*.

Integrity means that data cannot be changed with
be read without permission.

Our proof even goes one step further and shows t
certain degree. So-called information *side channe*
information inference channels that are present i
kernel *storage* channels, but excludes *timing* cha

Note that this proof statement is much stronger t
2009, we had only proved functional correctness
we have additionally shown binary correctness, th
have shown that the specification indeed implies

As with all proofs, there are still assumptions that
expectations on kernel behaviour that are not cap
degree of strong evidence for security and correc
kernel and is currently unrivalled.

What we assume

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

- **Assembly:** the seL4 kernel, like all operating system kernels, contains some assembly code
lines of ARM assembly in our case, and a similar amount for the RISCV64 version. For seL4,
concerns mainly entry to and exit from the kernel, as well as direct hardware accesses. For t
assume this code is correct.
- **Hardware:** we assume the hardware works correctly. In practice, this means the hardware i
not to be tampered with, and working according to specification. It also means, it must be ru
operating conditions.
- **Hardware management:** the proof makes only the most minimal assumptions on the und
hardware. It abstracts from cache consistency, cache colouring and TLB (translation lookasi
management. The proof assumes these functions are implemented correctly in the assembl
mentioned above and that the hardware works as advertised. The proof also assumes that e
these three hardware management functions do not have any effect on the behaviour of the
is true if they are used correctly.
- **Boot code:** the proof currently is about the operation of the kernel after it has been loaded
into memory and brought into a consistent, minimal initial state. This leaves out about 1,200
code base that a kernel programmer would usually consider to be part of the kernel.
- **Virtual memory:** under the standard of 'normal' formal verification projects, virtual memor
need to be considered an assumption of this proof. However, the degree of assurance is low
other parts of our proof where we reason from first principle. In more detail, virtual memory i
hardware mechanism that the kernel uses to protect itself from user programs and user prog
each other. This part is fully verified. However, virtual memory introduces a complication, be
affect how the kernel itself accesses memory. Our execution model assumes a certain stanc
behaviour of memory while the kernel executes, and we justify this assumption by proving th
conditions on kernel behaviour. The thing is: you have to trust us that we got *all* necessary c
and that we got them right. Our machine-checked proof doesn't force us to be complete at th
short, in this part of the proof, unlike the other parts, there is potential for human error.
- **DMA:** we assume that the CPU and MMU are the only devices that access memory directly.
can correctly ignore memory-mapped registers of devices, but has to assume that DMA devi
either not present or do not misbehave, for instance by overwriting the kernel. In practise this
that while normal user-level drivers cannot break kernel security, drivers for DMA enabled de
and must be formally verified for the proof to carry over. We have current work underway to e
assumption using the SystemMMU on ARM.
- **Information side-channels:** this assumption applies to the confidentiality proof only and
present for functional correctness or integrity. The assumption is that the binary-level model
hardware captures all relevant information channels. We know this not to be the case. This is
problem for the validity of the confidentiality proof, but means that its conclusion (that secre
leak) holds only for the channels visible in the model. This is a standard situation in informat
proofs: they can never be absolute. As mentioned above, in practice the proof covers all in-ke
channels but does not cover timing channels

Note that we *do not* need to trust the compiler and linker any more. Their output is formally verified
an automatic tool if the kernel is compiled with moderate optimisation levels. The same proof for r
aggressive optimisations is under development.

What do these assumptions mean?

The reduced proof assumptions mean that we do not need to trust the compiler or linker, but there may still be faults remaining in specific low-level parts of the kernel (TLB, cache handling, handwritten assembly, boot code). These parts are thoroughly manually reviewed.

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 at least two prominent research groups that have demonstrated successful formal verification of assembly code and low-level hardware management functions
and we have ourselves proved an earlier versio
executable specification. There are still significa
framework, but it is clear at this point that it can

With all the purity and strength of mathematical
theoretical limit of formal verification: there will
physical world left and these assumptions have
long as it talks about formal concepts. It is whe
Albert Einstein is quoted as saying "As far as th
as far as they are certain, they do not refer to re
randomly changes memory, the correctness pre
tests or verification methods help against cosm
mathematical proof, there are no absolute guar

There are two other assumptions that we do no

- We assume the axioms of higher-order lo
- We assume our prover checks this partic

The first is a fundamental question of formal lo
bigger problem than one verified OS kernel. The

From security properties down to C code, we us
so-called LCF family of provers and is engineere
correct. In particular, it supports external proof
absolute guarantee that the proof is correct, but
proofs, computers are *very good* at checking th
about the proof, be worried about the assumpti
problems.

From C code to binary code, we employ a set of
SONOLAR, Z3, Isabelle/HOL, and HOL4. The co
assurance for this last verification step and wor

What the proof implies

We have already covered the properties that are proved directly: functional correctness, integrity, and confidentiality. These are high-level properties that every OS should provide, that very few manage to provide, and that no OS has better evidence for than seL4.

The formal proof of functional correctness implies the absence of whole classes of common programming errors. Provided our assumptions above are true, some of these excluded 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.
- **Undefined behaviour:** there are many static analysis and verification tools that check for the absence of undefined behaviour in C. Our proof explicitly checks that no such undefined behaviour occurs.

The list goes on. 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), and why the code always executes in a defined and safe manner.

The term "verified" is overloaded out there

4.

"Verified" formal, machine-checked proof of strong properties down to the binary

Beware of claims from other systems

distance to reality

Static analysis

"no division by 0"

model

Assumptions

real world

Access control and Information flow

— —

Full functional correctness

"all possible behaviours are as expected in the specification"

seL4 has proofs of FC and security down to binary!

strength of properties

"the most verified microkernel"

We should celebrate and promote

the high-assurance

that seL4 and seL4-based systems provide

It is also important to ensure understanding of:

- **what exactly is proved**
- **the conditions under which the proofs hold**
- **what they imply in practice**

Summary

	Claim		Reality	Lesson
1.	I have downloaded seL4	⇏	I have a verified kernel	**Know what proofs hold for the configuration you have**
2.	My system is built on a verified seL4 configuration	⇏	My system enforces isolation and is verified	**Design and initialise your system correctly (and verify critical components)**
3.	Verified	⇏	0 bugs	**Know the scope and assumptions of the proofs for your configuration and what they mean for your**
4.	“Verified”	⇏	formal, machine-checked proof of strong properties down to the binary	**Beware of other uses of “verified”. seL4 has proofs of FC and security down to binary!**

Questions?

Google

seL4 FAQ

Google Search   I'm Feeling Lucky

- What is formal verification?
  - What does seL4's formal verification mean?
  - Does seL4 have zero bugs?
  - Is seL4 proved secure?
  - If I run seL4, is my system secure?
  - What are the proof assumptions?
  - How do I leverage seL4's formal proofs?
  - Have OS kernels not been verified before?
  - When and how often does seL4 get updated and re-proved?
  - How do I tell which code in GitHub is covered by the proof and which isn't?