L4: - Provides a small set of fundamental mechanisms and abstractions that run privileged services in kernel-mode, leaving typical operating systems tasks (such as process management, device drivers, interrupt handlers, file system, etc.) to be implemented and run as unprivileged user-mode servers. L4’s main features include memory protection, memory mapping between address spaces, low inter-process communication (IPC) overhead (very close to the host platform’s hardware-dictated context-switch costs), and a small footprint.

In particular, since L4 also avoids implementing policy, it does not provide any specific model of operating system services such as process management, memory and address space management, access control, etc. This task is left up to a supervisory OS running in user-mode on top of the microkernel. In our case this OS is Iguana

**What is seL4**?

**seL4 is an operating system microkernel**

An OS microkernel is a minimal core of an OS, reducing the code executing at higher privilege to a minimum.

**seL4 is also a hypervisor**

seL4 supports virtual machines that can run a fully fledged guest OS such as Linux.

**seL4 is proved correct**

seL4 comes with a formal, mathematical, machine-checked proof of implementation correctness, meaning the kernel is in a very strong sense “bug free” with respect to its specification. In fact, seL4 is the world’s first OS kernel with such a proof at the code level.

**seL4 is provably secure**

The kernel guarantees the classical security properties of confidentiality, integrity and availability.

**seL4 improves security with fine-grained access control through capabilities**

Capabilities are access tokens which support very fine-grained control over which entity can access a particular resource in a system. They support strong security according to the principle of least privilege (also called principle of least authority, POLA). This is a core design principle of highly secure system, and is impossible to achieve with the way access control happens in mainstream systems such as Linux or Windows.

POLA, every module (such as a process, a user, or a program, depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose.

seL4 is still the world’s only OS that is both capability-based and formally verified, and as such has defensible claim of being the world’s most secure OS.

**seL4 ensures safety of time-critical systems**

seL4 is the world’s only OS kernel (at least in the open literature) that has undergone a complete and sound analysis of its worst-case execution time (WCET). This means, if the kernel is configured appropriately, all kernel operations are bounded in time, and the bound is known.

This is a prerequisite for building hard real-time systems, where failure to react to an event within a strictly bounded time period is catastrophic.

**seL4 is the world’s most advanced mixed-criticality OS**

seL4 provides strong support for mixed criticality real-time systems (MCS), where the timelines of critical activities must be ensured even if they co-exist with less trusted code executing on the same platform. seL4 achieves this with a flexible model that retains good resource utilization, unlike the more established MCS OSes that use strict (and inflexible) time and space partitioning.

Note: - A mixed criticality system is a system containing computer hardware and software that can execute several applications of different criticality, such as safety-critical and non-safety critical, or of different Safety Integrity Level.

**seL4 is the world’s fastest microkernel**

Traditionally, systems are either (sort-of) secure, or they are fast. seL4 is unique in that it is both.

**seL4 Is a Microkernel and a Hypervisor, It Is Not an OS**

seL4 is a microkernel, and designed for generality while minimizing the TCB.

There are presently two main component frameworks for seL4, both open source: CAmkES and Genode.

CAmkES is a framework that is aimed at embedded and cyber-physical systems, which typically have a static architecture, meaning they consist of a defined set of components that does not change once the system has fully booted up.

Genode is in many ways a more powerful and general framework, that supports multiple microkernels and already comes with a wealth of services and device drivers, especially for x86 platforms

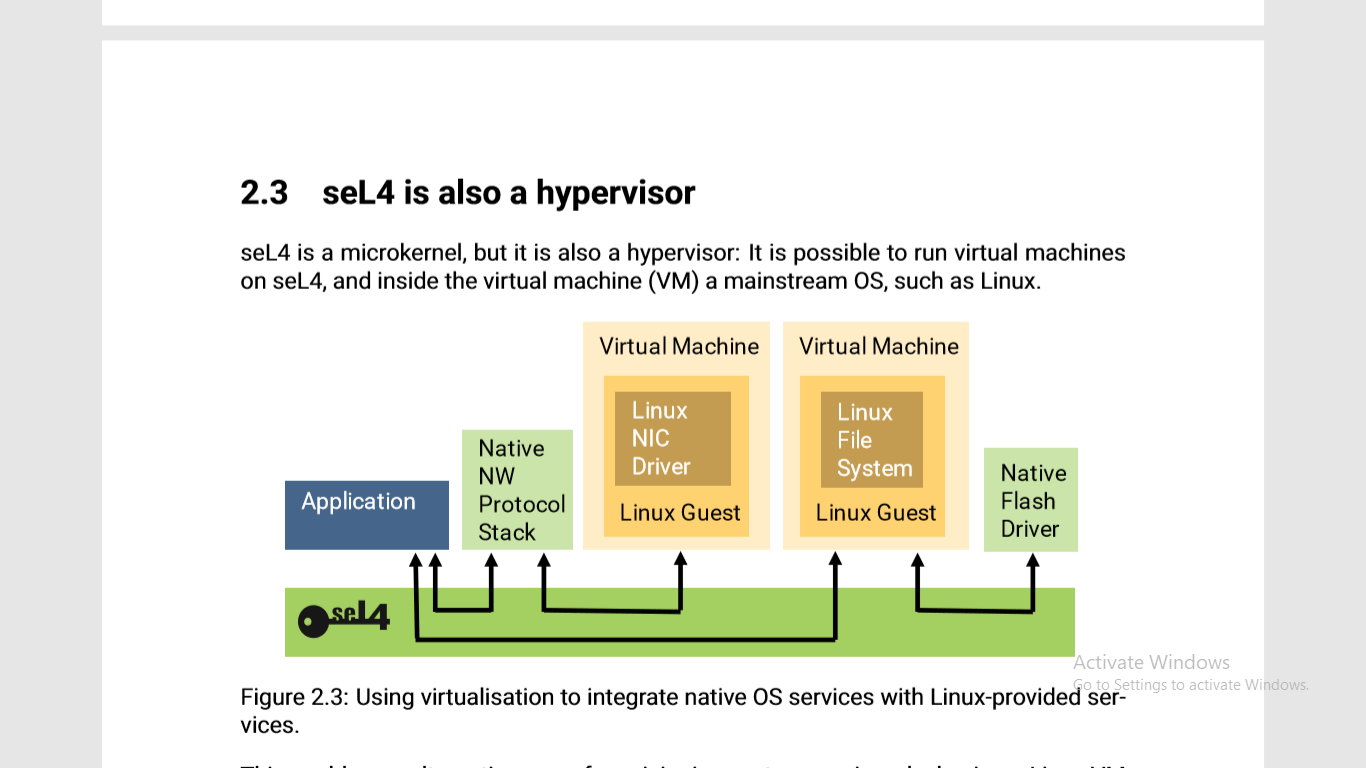
However, Genode has drawbacks:

1. As it supports multiple microkernels, not all as powerful as seL4, Genode is based on the least common denominator. In particular, it cannot use all of seL4’ssecurityand safety features.

2. It has no assurance story

**seL4 is also a hypervisor**

seL4 is a microkernel, but it is also a hypervisor: It is possible to run virtual machines onseL4, and inside the virtual machine(VM) a main stream OS, such as Linux.



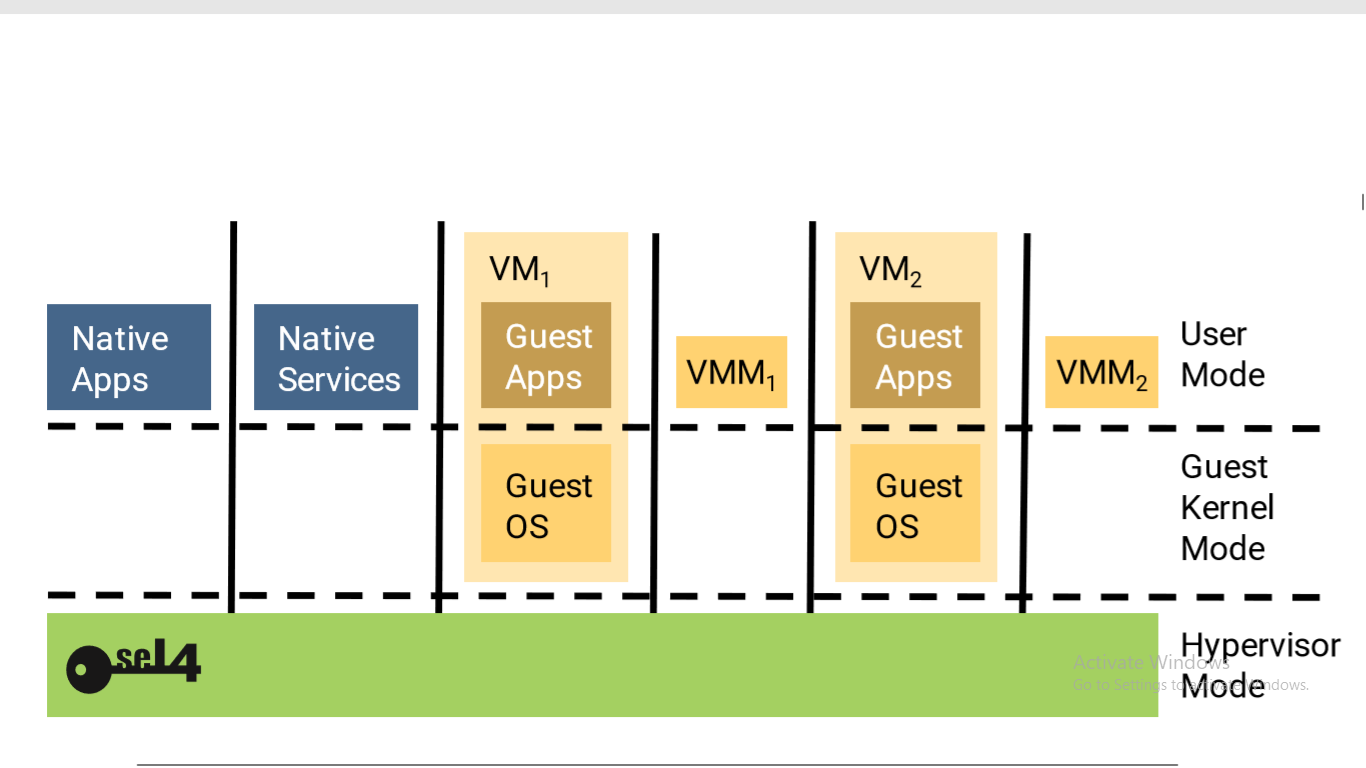
Such a setup is shown in Figure, which shows how some services are borrowed from multiple Linux instances running as guest OSes in separateVMs.

The protocol stack communicates with Linux via an seL4- provided channel, and the application similarly obtains network services by communicating with the protocol stack. Note that in the setup shown in the figure, the application has no channel to the NIC-driver VM, and thus cannot communicate with it directly, only via the NW stack; this is enabled by seL4’s capability-based protection.

The ﬁle system is a Linux one running in a VM, while the storage driver is native. Again, communication between the components is limited to the minimum channels required.

When used as a hypervisor, seL4 runs in the appropriate hypervisor mode (EL2on Arm, RootRing-0 on x86, HSonRISC-V), which is a higher privilege level than the guest operating system. Just as when running as the OS kernel, it only does the minimum work that has to be performed in the privileged (hypervisor) mode and leaves everything else to user mode.

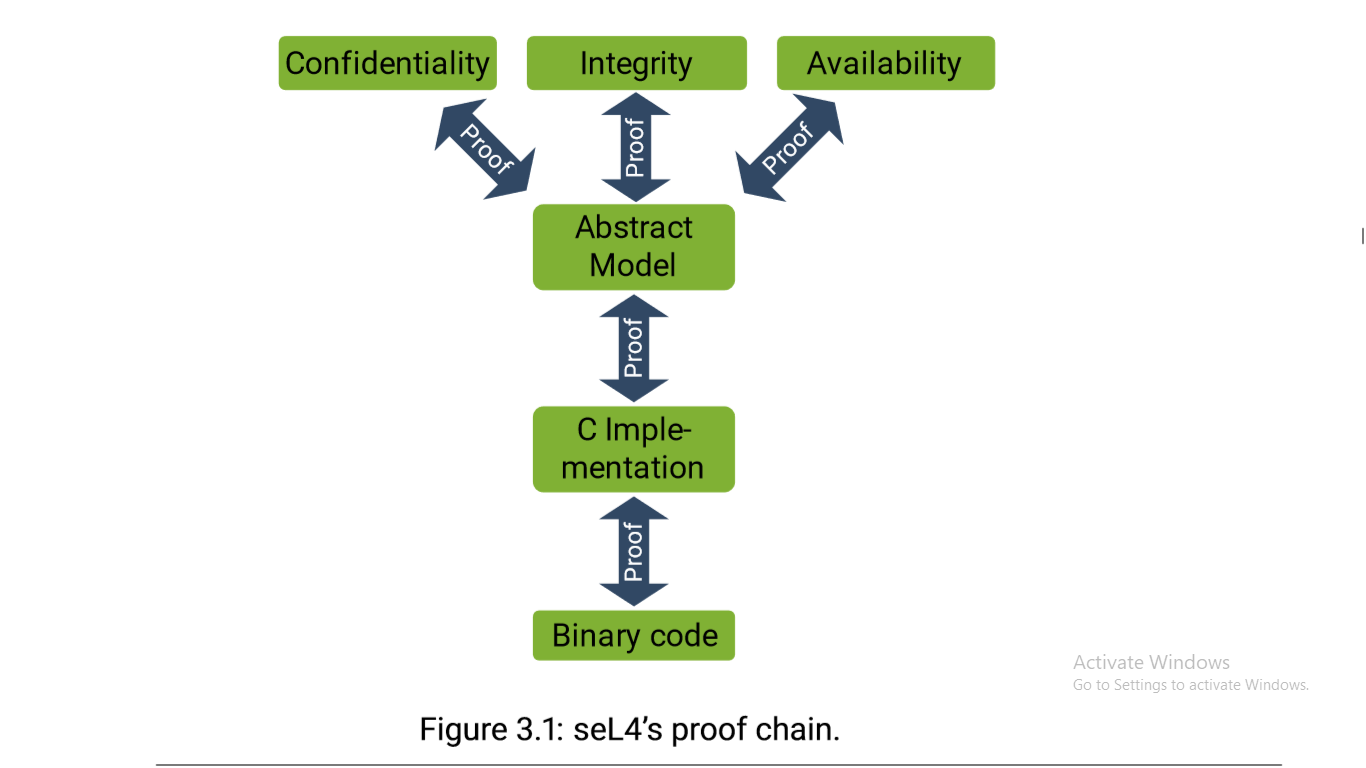
Specifically this means that seL4 performs world switches, meaning it switches virtual machine state when a VM’s execution time is up, or VMs must be switched for some other reason. It also catches virtualization exceptions (“VMexits” in Intel lingo) and forwards them to a user-level handler, called the virtual machine monitor (VMM). The VMM is then responsible for performing any emulation operations needed.



Each VM has its private copy of the VMM, isolated from the guest OS as well as from other VMs, as shown in. This means that the VMM cannot break isolation, and is therefore not more trusted than the guest OS itself. In particular, this means that there is no need to verify the VMM, as that would not add real assurance as long as the guest OS, typically Linux, is not verified.

seL4’s Verification Story

seL4 became the world’s ﬁrst OS kernel with a machine-checked functional correctness proof at the source-code level.



The set of proofs has by now grown to well over a million lines, most of this manually written and then machine checked.

**Functional Correctness**

The core of seL4’s verification is the functional correctness proof, which says that the C implementation is free of implementation defects. More precisely, there is a formal specification of the kernel’s functionality, expressed in a mathematical language called higher-order logic (HOL). The functional correctness proof then says the possible behaviors of the C code are a subset of those allowed by the abstract model.

The proof means that everything we want to know about the kernel’s behavior (other than timing) is expressed by the abstract spec, and the kernel cannot behave in ways that are not allowed by the spec. Among others, this rules out the usual attacks against operating systems, such as stack smashing, null-pointer dereference, any code injection or control flow high jacking etc.

**Translation validation**

Having a bug-free C implementation of the kernel is great, but still leaves us at the mercy of the compiler. Those compilers (we use GCC) are themselves large, complex programs that have bugs. So we could have a bug-free kernel that gets compiled into a buggy binary.

In the security-critical space, compiler bugs are not the only problem. A compiler could be out right malicious, containing a Trojan that automatically builds in a backdoor when compiling the OS.

To protect against defective or malicious compilers, we additionally verify the executable binary that is produced by the compiler and linker. Specifically, we prove that the binary is a correct translation of the (proved correct) C code, and thus that the binary refines the abstract spec.

*I didn’t immerse in to details!!!!*

**Security Properties**

Confidentiality: seL4 will not allow an entity to read (or otherwise infer) data without having been explicitly given read access to the data;

Integrity: seL4 will not allow an entity to modify data without having been explicitly given write access to the data;

Availability: seL4 will not allow an entity to prevent another entity’s authorized use of resources.

They prove that in a correctly configured system, the kernel will enforce these properties.

**Proof assumptions**

All reasoning about correctness is **based on assumptions**, whether the reasoning is formal, as with seL4, or informal, when someone thinks about why their program might be “correct”. Every program executes in some context, and its correct behavior inevitably depends on some assumptions about this context.

One of the advantages of machine-checked formal reasoning is that it forces people to make those assumptions explicit. It is not possible to make unstated asumptions; the proofs will just not succeed if they depend on assumptions that are not clearly stated.

In that sense, formal reasoning protects against forgetting assumptions, or not being clear about them; that in itself is a significant benefit of verification.

*The verification of seL4 makes three assumptions:*

**Hardware behaves as expected**: - This should be obvious. The kernel is at the mercy of the underlying hardware, and if the hardware is buggy (or worse, has Trojans), then all bets are off, whether you are running veri­fied seL4 or any unverified OS.

**The spec matches expectations**. This is a difficult one, because one can never be sure that a formal speci­fication means what we think it should mean. But in the end, there is always a gap between the world of mathematics and the physical world, and no end of reasoning (formal or informal) can remove this completely. The advantage of formal reasoning is that you know exactly what this gap is.

**The theorem prover is correct**. This sounds like a serious problem, given that theorem provers are themselves large and complex programs. However, in reality this is the least concerning of the three assumptions. The reason is that the **Isabelle/HOL** theorem prover has a small core (of a few 10 kSLOC) that checks all proofs against the axioms of the logic. And this core has checked many proofs small and large from a wide ­field of formal reasoning, so the chance of it containing a correctness critical bug is extremely small.

**Proof status and coverage**

seL4 has been or is being verified for multiple architectures: Arm, x86 and RISC-V.

**Real-World Deployment and Incremental Cyber Retrofit**

Cyber Retrofit

7.1 General considerations

1. When planning to protect the security or safety of your system with seL4, the first step should be to identify the critical assets you need to protect. The aim should be to minimize this part of your trusted computing base, and make it as modular as feasible, with each module becoming an seL4-protected CAmkES component.

2. The other important preparation is to check availability and verification status of seL4 on your platform. Obviously you will want a verified kernel, that’s what seL4 is all about.

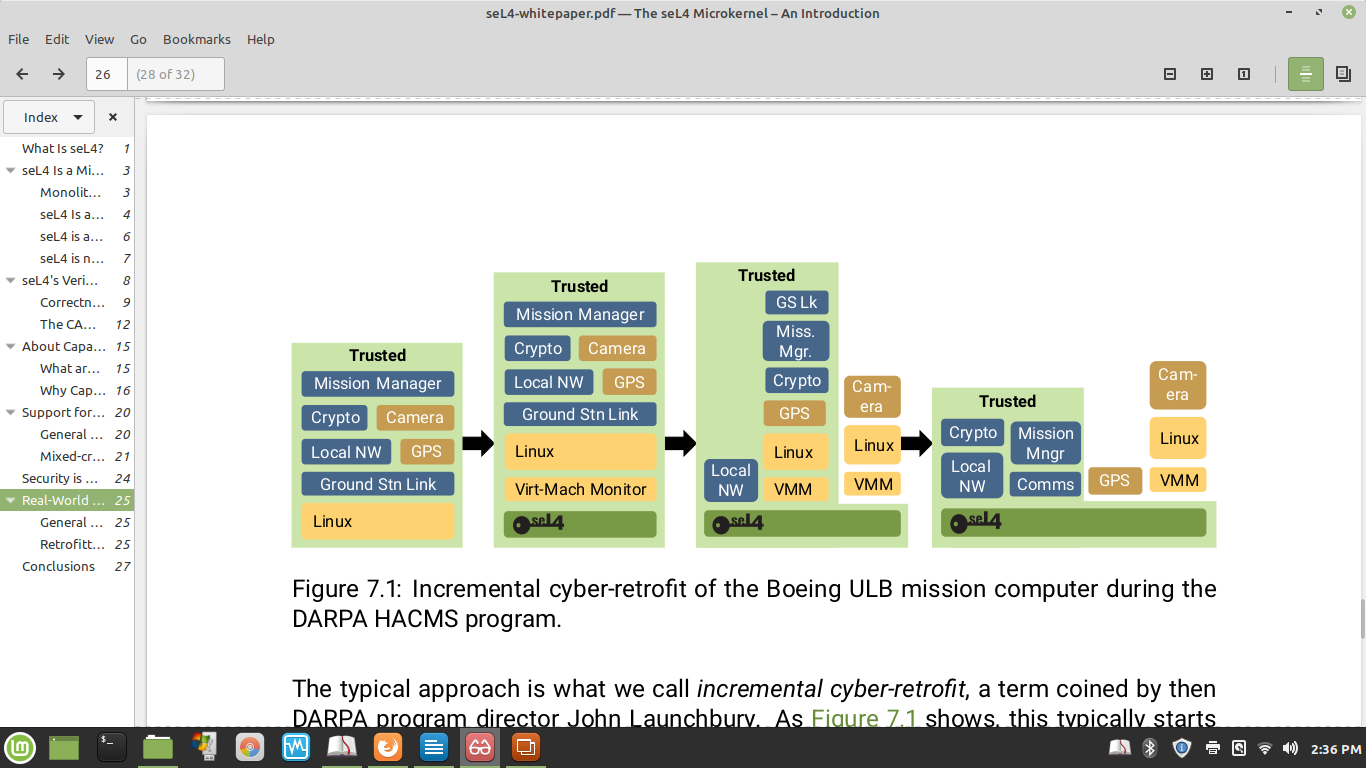
You must not make any verification claim if you are using a kernel that is not verified for your platform, or that is in any way modified.

3. You furthermore will need to assess whether the available user-level infrastructure is sufficient for your purpose. If not, then this is where the community may help you. There are companies specializing in providing support for seL4 adoption.

7.2 Retrofitting existing systems

Most real-world deployments of seL4 will not run everything native. Typically, there are significant legacy components that would be expensive to port, because they are too big or rely on too many system services that are not presently supported by seL4. Also, frequently there would be little security or safety gain from running such legacy stacks naively.

Using seL4’s virtualization capabilities is frequently the pragmatic way to proceed, Section 2.3 shows examples.



The typical approach is what we call incremental cyber-retrofit, a term coined by then DARPA program director John Launchbury.

1. As Figure 7.1 shows, starts out by simply putting the whole existing software stack into a virtual machine running on seL4. Obviously this step buys nothing in terms of security and safety, it only adds (very small) overhead. Its significance is that it provides a baseline from where to start modularising.

*A great example is the work our HACMS project partners did on cyber-retrofitting the Boeing ULB autonomous helicopter. The original system ran on Linux, and in a first step, the team put seL4 underneath.*

2. The next step broke out two components: The particularly untrusted camera software was moved to a second VM, also running Linux, with the two Linux VMs communicating via CAmkES channels. At the same time, the network stack was pulled out of the VM and converted to a native CAmkES component, also communicating with the main VM.

3. The final step pulled all other critical modules, as well as the (untrusted) GPS software, into separate CAmkES components, removing the original main VM. The final system consisted of a number of CAmkES components running seL4-native code, and a single VM running just Linux and the camera software.

The upshot was that while the initial system was readily hacked by the professional

penetration testers hired by DARPA, the end state was highly resilient. The attackers could compromise the Linux system and do whatever they wanted with it, but were unable to break out and compromise any of the rest of the system. The team was confident enough to demonstrate an attack in-flight.

seL4’s assurance story.

Its security and safety relevant features.

Its benchmark-setting performance.

incrementalcyberretroﬁtoflegacysystemss

CSS: - computing and classification system.

The ACM Computing classification system (CCS) is a subject classification system for computing devised by the association for computing machinery (ACM). The system is comparable to the mathematics subject classification (MSC)

CCSConcepts

• Softwareanditsengineering→OperatingSystems

• Securityandprivacy→Systemssecurity

• Securityandprivacy→Formalmethodsandtheoryofsecurity

• Computer systems organization → Real-The principle means giving a [user account](https://en.wikipedia.org/wiki/User_account) or process only those privileges which are essential to perform its intended function. For example, a user account for the sole purpose of creating backups does not need to install software: hence, it has rights only to run backup and backup-related applications. Any other privileges, such as installing new software, are blocked.time systems → Real-time operating systems

• Computersystemsorganization→Real-timesystems→Dependableandfaulttolerantsystemsandnetworks

**seL4 tutorial note**

Historically microkernel was slow when compared to monolithic kernel.

On seL4 microkernel, performance is achieved by enhancing IPC (message passing mechanism)

**Note**:

A **mixed criticality**: - system is a system containing computer hardware and [software](https://en.wikipedia.org/wiki/Software) that can execute several applications of different criticality, such as safety-critical and non-safety critical, or of different Safety Integrity Level (SIL).

The term **trusted computing base** defined as the combination of kernel and trusted processes. The latter refers to processes which are allowed to violate the system's access-control rules.

Find out what Genode is!

The **principle of least privilege** (**PoLP**), also known as the **principle of minimal privilege** or the **principle of least authority**, requires that in a particular [abstraction layer](https://en.wikipedia.org/wiki/Abstraction_layer) of a computing environment, every module (such as a [process](https://en.wikipedia.org/wiki/Process_(computing)), a [user](https://en.wikipedia.org/wiki/User_(computing)), or a [program](https://en.wikipedia.org/wiki/Computer_program), depending on the subject) must be able to access only the information and [resources](https://en.wikipedia.org/wiki/Resource_(computer_science)) that are necessary for its legitimate purpose.

The principle means giving a [user account](https://en.wikipedia.org/wiki/User_account) or process only those privileges which are essential to perform its intended function. For example, a user account for the sole purpose of creating backups does not need to install software: hence, it has rights only to run backup and backup-related applications. Any other privileges, such as installing new software, are blocked.

**Abbreviation**

KSLOC: - Kilo Source Lines of Code

CAmkES: - Component architecture for microkernel-based embedded systems

**Dictionary**

Inevitably: - In such a manner as could not be otherwise.

Retrofit: - Fit in or on an existing structure, such as an older house

Provably: - In an obvious and provable manner

Spec: - A detailed description of design criteria for a piece of work