The seL4 Foundation

https://sel4.systems/Foundation

The seL4® Microkernel An Introduction

Gernot Heiser

gernot@sel4.systems

Revision 0.3

2020-04-07

Abstract

This white paper gives an overview of seL4. It explains what seL4 is and what its defining features are. In particular, we explain its assurance story and typical deployment scenarios.

Chapter 1

What Is seL4?

seL4 is an operating system microkernel

An operating system (OS) is the low-level system software that controls a computer system's resources and enforces security. Unlike application software, the OS has exclusive access to a more privileged execution mode of the processor (*kernel mode*) that gives it direct access to hardware. Applications only ever execute in *user mode* and can only access hardware as permitted by the OS.

An OS microkernel is a minimal core of an OS, reducing the code executing at higher privilege to a minimum. seL4 is a member of the L4 family of microkernels that go back to the mid-1990s. (And no, *seL4 has nothing to do with seLinux.*)

seL4 is also a hypervisor

seL4 supports virtual machines that can run a fully fledged guest OS such as Linux. Subject to seL4's enforcement of communication channels, guests and their applications can communicate with each other as well as with native apps.

Learn more about what it means that seL4 is a microkernel and its use as a hypervisor in Chapter 2. And learn about real-world deployment scenarios, including approaches for retrofitting security into legacy systems in Chapter 7.

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 [Klein et al., 2009].

seL4 is provably secure

Besides implementation correctness, seL4 comes with further proofs of *security enforcement* [Klein et al., 2014]. They say that in a correctly configured seL4-based system, the kernel will guarantee the classical security properties of *confidentiality, integrity and availability*. More about these proofs in Chapter 3.

seL4 improves security with fine-grained access control through capabilities

Capabilities are access token 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.

seL4 is still the *world's only OS that is both capability-based and formally verified*, and as such has a defensible claim of being the world's most secure OS. More about capabilities in Chapter 4.

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) [Blackham et al., 2011, Sewell et al., 2017]. 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 timeliness of critical activities must be ensured even if it co-exists with less trusted code executing on the same platform. seL4 achieves this with a flexible model that retains good resource utilisation, unlike the more established MCS OSes that use strict (and inflexible) time and space partitioning [Lyons et al., 2018]. More on seL4's real-time and MCS support in Chapter 5.

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. We built seL4 for real-world use across a wide class of use cases, whether they are security- (or safety-)critical or not. More on seL4 performance in Chapter 6.

seL4 is pronounced "ess-e-ell-four"
The pronunciation "sell-four" is depreciated.

How to read this document

This document is meant to be approachable by a wide audience. However, for completeness, we will add some deeper technical detail in places.

Such detail will be marked with a chilli, like the one on the left. If you see this then you know you can safely skip the marked section if you are not interested in this level of detail, as only other chillied sections will assume you have read it.

Chapter 2

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

2.1 Monolithic Kernels vs Microkernels

To understand the difference between a mainstream OS, such as Linux, and a microkernel, such as seL4, let's look at Figure 2.1.

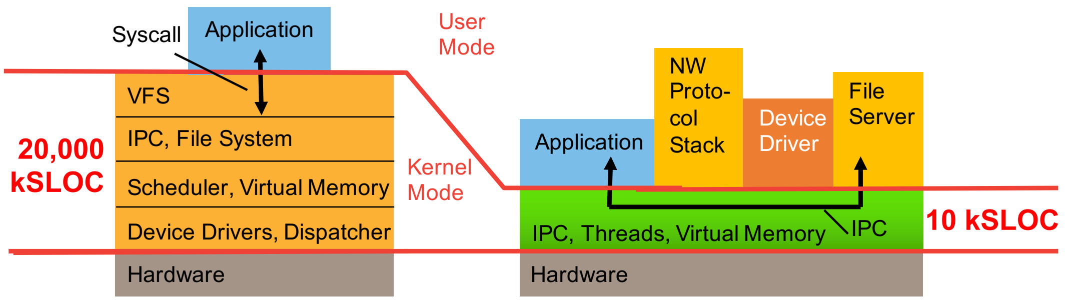

Figure 2.1: Operating-system structure: Monolithic kernel (left) vs microkernel (right).

The left side presents a (fairly abstracted) view of the architecture of a system such as Linux. The orange part is the OS *kernel*, it offers services such as file storage and networking to applications. All the code that implements those services executes in the *privileged mode* of the hardware, also called *kernel mode* or *supervisor mode* – the execution mode that has unfettered access and control of all resources in the system. In contrast, applications run in unprivileged, or *user mode*, and do not have direct access to many hardware resources, which must be accessed through the OS. The OS is internally structured in a number of layers, where each layer provides abstractions implemented by layers below.

The problem with privileged-mode code is that it is dangerous: If anything goes wrong here, there's nothing to stop the damage. In particular, if this code has a bug that can be exploited by an attacker to run the attacker's code in privileged mode (called a privilege-escalation or arbitrary code-execution attack) then the attacker can do what they want with the system. Such flaws are the root problem of the many system compromises we experience.

Of course, software bugs are mostly a fact of life, and OSes are not different. In fact, with the Linux kernel comprising of the order of 20 million lines of source code (20 MSLOC), we can estimate that it contains literally tens of thousands of bugs [Biggs et al., 2018]. This is obviously a huge attack surface! This idea is captured by saying that Linux has a large *trusted computing base* (TCB), which is defined as the subset of the overall system that must be trusted to operate correctly for the system to be secure.

The idea behind a microkernel design is to drastically reduce the TCB and thus the attack surface. As schematically shown at the right of Figure 2.1, the kernel, i.e. the part of the system executing in privileged mode, is much smaller. In a well-designed microkernel, such as seL4, it is of the order of ten thousand lines of source code (10 kSLOC). This is literally three orders of magnitude smaller than the Linux kernel, and the attack surface shrinks accordingly (maybe more, as the density of bugs probably grows more than linear with code size).

Obviously, it is not possible to provide the same functionality, in terms of OS services, in such a small code base. In fact, the microkernel provides almost no services: it is just a thin wrapper around hardware, just enough to securely multiplex hardware resources. What the microkernel mostly provides is isolation, sandboxes in which programs can execute without interference from other programs. And, critically, it provides a *protected procedure call* mechanism, for historic reasons called *IPC*. This allows one program to securely call a function in a different program, where the microkernel transports function inputs and outputs between the programs and, importantly, enforces interfaces: the “remote” (as in contained in a different sandbox) function can only be called with exactly the parameters its signature specifies.

The microkernel system uses this approach to provide the services the monolithic OS implements in the kernel. In the microkernel world, these services are just programs, no different from apps, that run in their own sandboxes, and provide an IPC interface for apps to call. Should a server be compromised, that compromise is confined to the server, its sandbox protects the rest of the system. This is in stark contrast to the monolithic case, where a compromise of an OS service compromises the complete system.

This effect can be quantified: Our recent study shows that of the known Linux compromises classified as *critical*, i.e. most severe, 29% would be fully eliminated by a microkernel design, and another 55% would be mitigated enough to no longer qualify as critical [Biggs et al., 2018].

2.2 seL4 Is a Microkernel, Not an OS

When we talk about seL4, we talk about the seL4 microkernel. Many people confuse it with seLinux (probably because seL4 might be mistaken as a shorthand for the 4th version of seLinux). seLinux is a security policy framework built into Linux. While in some ways more secure than standard Linux, it suffers from the same problem of a huge TCB, and correspondingly huge attack surface, as Linux does. *seLinux is not suitable for truly security-critical uses.*

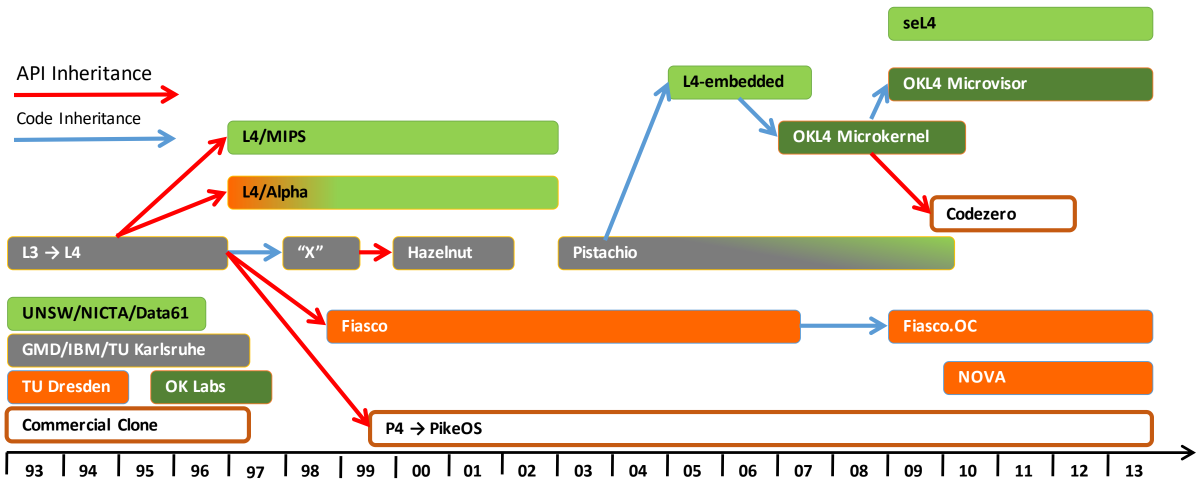

Figure 2.2: L4 microkernel family tree.

Figure 2.2 shows seL4's provenance as a member of the L4 microkernel family. It was developed by the UNSW/NICTA group that, at the time, had 15 years of experience in developing high-performance microkernels. And it had a track-record of real-world deployments: Our *L4-embedded* kernel from the mid-Noughties shipped on billions of Qualcomm cellular modem chips, and our *OKL4 Microkernel* runs on the secure enclave of all recent iOS devices (iPhones etc).

But, because seL4 is a microkernel, it contains none of the usual OS services, these are separate programs running in user mode. This has its good and its bad sides. The bad side is that these components must be provided. Some can be ported from open-source OSes such as Linux or FreeBSD, or they can be written from scratch. But in any case, this is significant work.

To scale up we need the help of the community, and the seL4 Foundation is the key mechanism for enabling the community to cooperate and develop or port such services for seL4-based systems. The most important ones are device drivers, network protocol stacks, and file systems. We have a fair number these, but much more is needed..

An important enabler are component frameworks; they allow developers to focus on the code that implements the services, and automate much of the system integration. 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. It is arguably more convenient to work with than CAmkES, and is certainly the way to get a complex system up quickly. 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's security and safety features. 2. It has no assurance story. More on this in

Section 3.2.

2.3 seL4 Is Also a Hypervisor

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

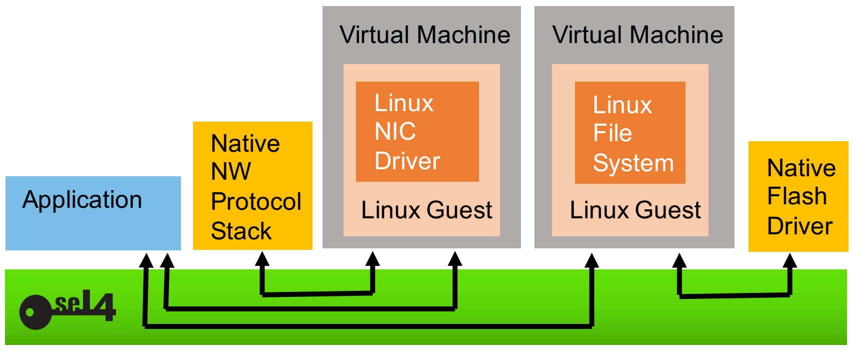

Figure 2.3: Using virtualisation to integrate native OS services with Linux-provided services.

This enables an alternative way of provisioning system services, by having a Linux VM provide them. Such a setup is shown in Figure 2.3, which shows how some services are borrowed from multiple Linux instances running as guest OSes in separate VMs.

In this example, we provide two system services: file system and networking. Networking is provided by a native protocol stack running directly on seL4, lwIP or PicoIP are frequently used stacks. Instead of porting a network driver, we borrow one from Linux, by running a VM with a stripped-down Linux guest that has little more than the NIC driver. 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.

A similar setup is shown for the file-system service, this time 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. In particular, the app cannot talk to the storage driver (except through the file system), and the two Linux systems cannot communicate with each others.

When used as a hypervisor, seL4 runs in the appropriate hypervisor mode (EL2 on Arm, Root Ring-0 on x86, HS on RISC-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 virtualisation exceptions ("VM

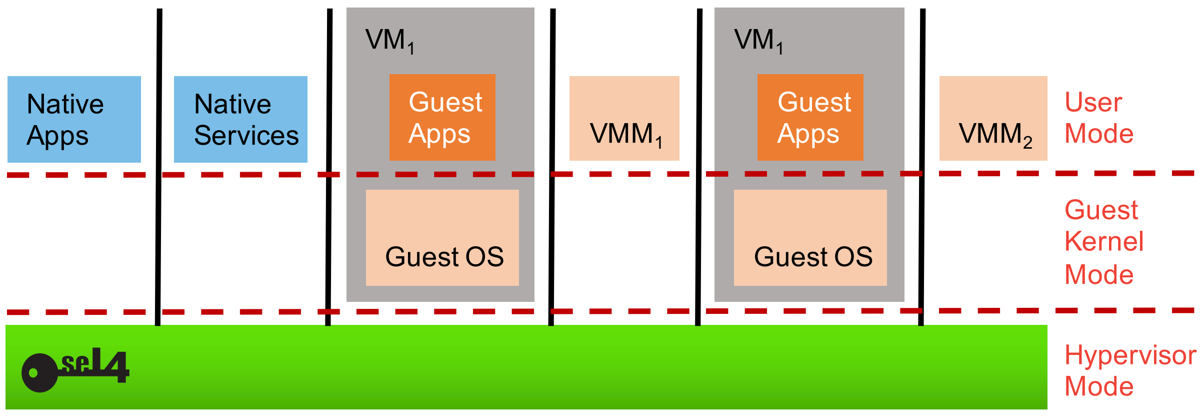

Figure 2.4: seL4 virtualisation support with usermode VMMs.

exits" 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 Figure 2.4. 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.

Chapter 3

seL4's Verification Story

In 2009, seL4 became the world's first OS kernel with a machine-checked functional correctness proof at the code level. This proof was 200,000 lines of proof script at the time, one of the largest ever (we think it was the second largest then). It showed that functionally correct OSes were possible, something that until then had been considered infeasible.

Since then we have extended the scope of the verification to higher level properties. Importantly, we maintained the proof with the on-going evolution of the kernel: Commits to the mainline kernel source are only allowed if they do not break proofs, or the poofs have been updated as well. This *proof engineering* is also a novelty. Our seL4 proofs are by far the largest proof base that is actively maintained, they have by now grown to well over a million lines, most of this manually written and then machine checked.

3.1 Correctness and Security Enforcement

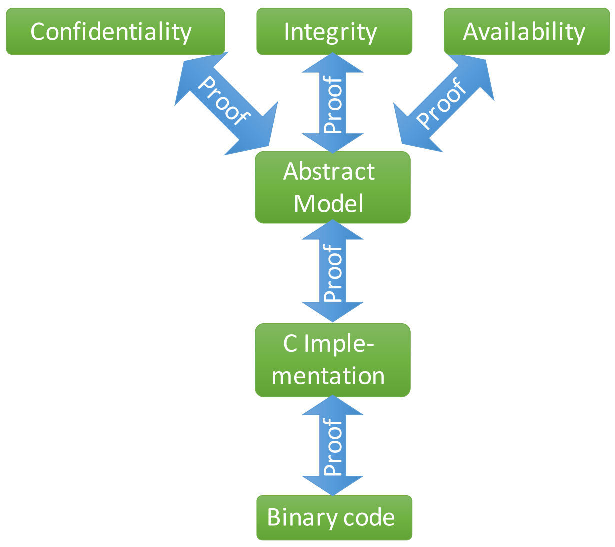

Figure 3.1: seL4's proof chain.

Figure 3.1 shows the chain of proofs we have for seL4.

Functional correctness

The core of it 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). This is represented by the box labelled *abstract model* in the figure. The functional correctness proof then says that the C implementation is a *refinement* of the abstract model, meaning its possible behaviours are a subset of those allowed by the abstract model.

This informal description glosses over a lot of detail. Here is some of it in case you wonder.

C is not a formal language; in oder to allow reasoning about a C program in the theorem prover (we use Isabelle/HOL), it has to be transformed into mathematical logic (HOL). This is done by a C parser written in Isabelle. The parser defines the semantics of the C program, and gives it meaning in HOL according to this semantics. It is this formalisation which we prove to be a refinement of the mathematical (abstract) model.

Note that C does not have an official mathematical semantics, and parts of the C language are notoriously subtle and not necessarily that well defined. We solve this by restricting our use of C to a well-defined subset of the language, for which we have an unambiguous semantics. However, this does not guarantee that our assumed semantics for that subset is the same as the compiler's. More on that below.

The proof means that everything we want to know about the kernel's behaviour (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 highjacking etc.

Translation validation

Having a bug-free C implementation of the kernel is great, but still leaves us at the mercy of the C 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.

To protect against such compiler defects, we have a separate proof, from C to the executable binary that is produced by the compiler and linker. It proves that the binary is a correct translation of the (proved correct) C code, and thus that the binary refines the abstract spec.

Unlike the verification of the C code, this proof is not done manually but by an automatic tool chain. It consists of several phases. A formal model of the processor's instruction set architecture (ISA) formalises the binary in the theorem prover; we use the an L3 formalisation of the RISC-V ISA, as well as the extensively tested L3 Arm ISA formalisation of Fox and Myreen [2010].

Then a disassembler, written in the HOL-4 theorem prover, translates this low-level representation into a higher-level representation in a graph language that basically represents control flow. This transformation is provably correct.

The formalised C program is also translated into the same graph language, also provably correct in the Isabelle/HOL theorem prover. We then have two programs in the same representation for which we need to show that they are the same. This is a bit tricky, as compilers apply a number of heuristic-driven transformations to optimise the code. We apply a number of such transformations through rewrite rules on the graph-language representation of the C program (still in the theorem prover, and thus provably correct).

In the end we then have two programs that are quite similar but not the same, and we need to prove that they have the same semantics. In theory this is equivalent to the halting problem and as such unsolvable. In practice, what the compiler does is deterministic enough to make the problem tractable. We do this by throwing the programs, bit for bit, at a set of multiple SMT solvers. If one of these can prove that all the corresponding pieces have the same semantics, then we know that the two programs are equivalent.

Note also that the C program that is proved to refine the abstract spec, and the C program that we prove to be equivalent to the binary, are the same Isabelle/HOL formalisations. This means that our assumptions on C semantics drop out of the assumptions made by the proofs. Altogether, the proofs not only show that the compiler did not introduce bugs, but also that its semantics for the C subset we use are the same as ours.

Security properties

Figure 3.1 also shows proofs between the abstract spec and the high-level security properties *confidentiality*, *integrity* and *availability* (these are commonly dubbed the *CIA properties*). These basically state that the abstract spec is actually useful for security: They prove that in a correctly configured system, the kernel will enforce these properties.

What this means is that:

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 authorised use of resources.

These proofs presently do not capture properties associated with time. Our confidentiality proofs rule out *covert storage channels* but presently not *covert timing channels*, although this is something we are working on [Heiser et al., 2019]. Similarly, the integrity and availability proofs presently do not cover timeliness. Our new MCS model [Lyons et al., 2018] is designed to cover those aspects.

Proof assumptions

All reasoning about correctness must be 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 behaviour 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 assumptions, 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.

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. Of course, the same problem exists if there is *no* formal specification: if the spec is informal or non-existent, then it is obviously impossible to precisely reason about correct behaviour.

One can reduce this risk by proving properties about the spec, as we have done with our security proofs, which show that seL4 is able to enforce certain security properties. That then shifts the problem to the specification of those properties. They are much simpler than the kernel spec, reducing the risk of misunderstanding.

But in the end, there is always a gap between the cyber world 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. Some of these are more complete than others, but the missing bits are generally worked on or waiting for funding. Please refer to the seL4 project status page for details.

3.2 The CAmkES component framework

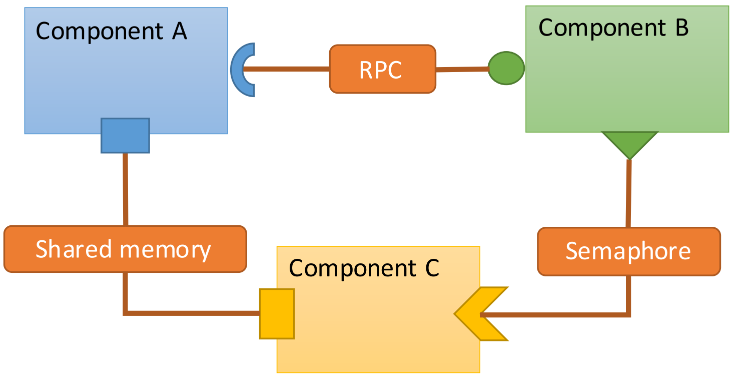

Figure 3.2: CAmkES components and connectors.

CAmkES is a component framework that allows you to reason about a system architecturally, i.e. as a collection of sandboxed components with defined communication channels. Figure 3.2 shows the main abstractions.

Components are represented as square boxes. They represent encapsulated programs, code and data, running on seL4.

Interfaces are shown as decoration on the components. They define how a component can be invoked, or can invoke others. An interface is either imported (invoking an interface of another component) or exported (able to be invoked by another components imported interface), except for the shared-memory interface, which is symmetric.

Connectors connect like interfaces, each an importing with an exporting interface. Connectors in CAmkES are always one-on-one, broadcast or multicast functionality could be implemented on top of this model by replicating components.

The CAmkES system is specified in a formal *architecture description language* (the CAmkES ADL), which contains a precise description of the components, their interfaces and the connectors that link them up. The CAmkES's promise to the system designer is that what is specified in the ADL (and visualised as in Figure 3.2) is a faithful representation of the possible interactions. In particular, it promises that no interactions are possible beyond those shown in the diagram.

Of course, this promise depends on enforcement by seL4, and the ADL representation must be mapped onto low-level seL4 objects and access rights to them. This is what the CAmkES machinery achieves, and is shown in Figure 3.3.

In the figure, the architecture (i.e. what is described in the ADL) is shown at the top. This is a fairly simple system, consisting of four native components and one component that houses a virtual machine hosting a Linux guest with a couple of networking drivers. The Linux VM is only connected to other components via the crypto component, which ensures that it can only access encrypted links and cannot leak data.

Even this simple system maps to hundreds if not thousands of seL4 objects, an indication of the complexity reduction provided by the CAmkES component abstraction.

For the seL4-level description we have another formal language, called CapDL (capability description language). The system designer never needs to deal with CapDL, it is a purely internal representation. The CAmkES framework contains a compiler which automatically translates CAmkES ADL into CapDL, indicated by the box arrow pointing left-down. The box in the left of the figure gives a (simplified) representation of the seL4 objects described in CapDL. (It is actually a simplified representation of a much simpler system, basically just the two components at the top of Figure 3.2 and the connector between them.)

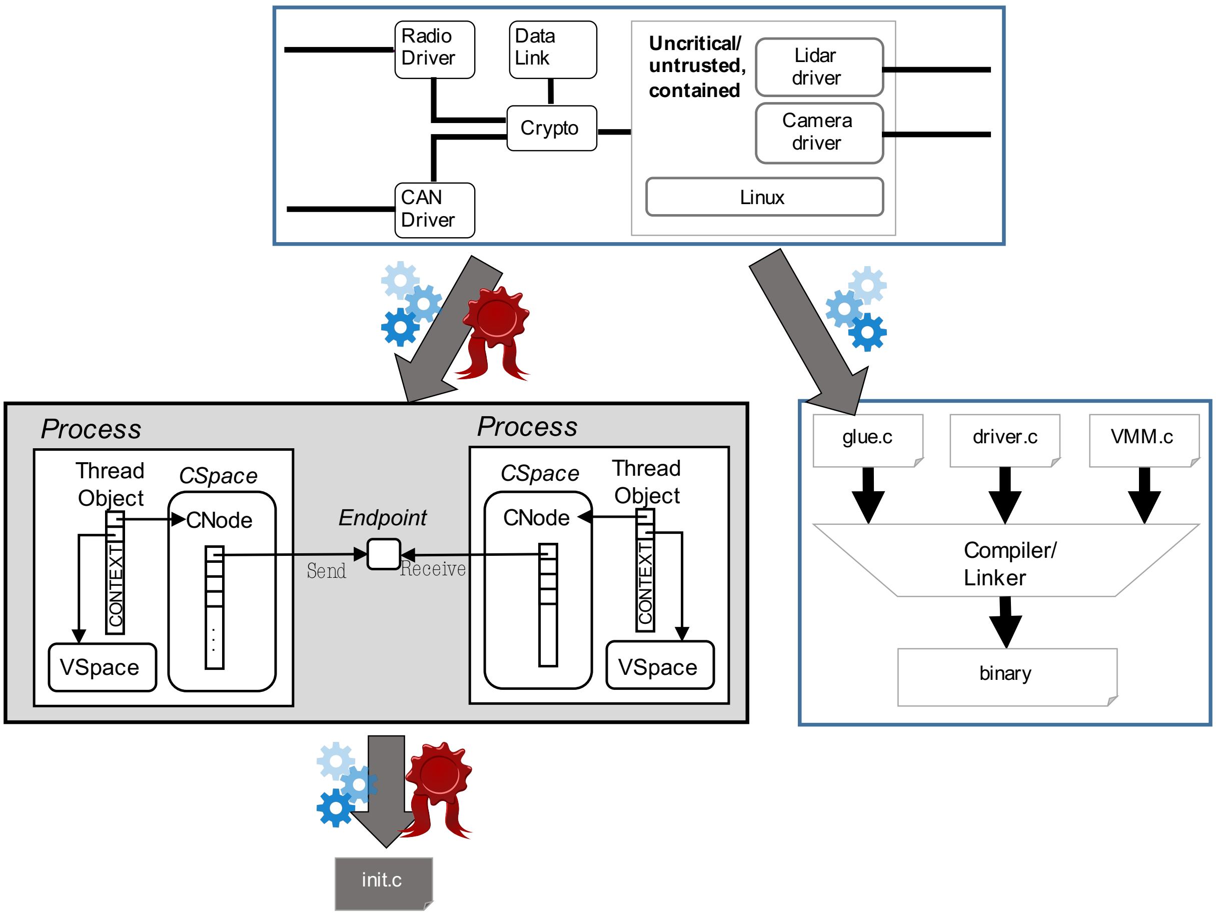

Figure 3.3: Verified architecture mapping and system generation (note that not all verification steps are of full strength yet). Gray boxes are generated provably correct.

The CapDL spec is a precise representation of access rights in the system, and it is what seL4 enforces. Which means that once the system gets into the state described by the CapDL spec, it is guaranteed to behave as described by the CAmkES ADL spec, and therefore the architecture-level description is sufficient for further reasoning about security properties.

So we need assurance that the system boots up into the state described by the CapDL spec. We achieve this with a second automated step: We generate from CapDL the startup code that, as soon as seL4 itself has booted, takes control and generates all the seL4 objects referenced by the spec, including the ones describing active components, and distributes the *capabilities* (see Chapter 4) that grant access to those objects according to the spec. At the end of the execution of this init code, the system is provably in the state described by the CapDL spec, and thus in the state represented by the ADL spec.

The third thing that gets generated from the ADL spec is the “glue” code between components. Sending data through a connector requires invocation of seL4 system calls, the exact details of which are hidden by the CAmkES abstraction. The glue code is setting up these system calls. For example, an “RPC” connector abstracts the invocation of a function provided by another component as a regular function call performed by the client component.

Note: At the time of writing, the proofs about CAmkES and CapDL are not yet complete, but completion should not be far off.

Note also that none of the verification work mentioned deals with information leakage through timing channels (yet). This is a major unsolved research problem, but we’re at the forefront of solving it.

Chapter 4

About Capabilities

We encountered capabilities in Chapter 1, noting that they are access token. We will now look at the concept in more detail.

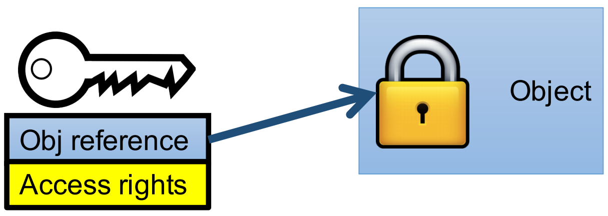

Figure 4.1: A capability is key that conveys specific rights to a particular object.

As shown in Figure 4.1, a capability is an object reference, in that sense it is similar to a pointer (and implementation of capabilities are often referred as “fat pointers”). They are immutable pointers, in the sense that a capability will always reference the *same* object, so each capability uniquely specifies a particular object.

In addition to pointers, a capability also encodes access rights, in fact, the capability is an encapsulation of an object reference and the rights it conveys to that object. In a capability-based system, such as seL4, invoking a capability is the one and only way of performing an operation on a system object.

For example, an operation may be to call a function in a component. The object reference embedded in the capability then points to an interface to that object, and conveys the right to invoke that function. The capability may or may not at the same time convey the right to pass another capability along as a function argument (delegating to the component the right to use the object referenced by the capability argument).

This is a high-level description of what happens at the CAmkES abstraction level. In fact, at the CAmkES level, the capabilities themselves are abstracted away.

Underneath, the connector is represented by an *endpoint* object, and the client component needs a capability with `call` right.

It is this fine-grained object-oriented nature that makes capabilities the access-control mechanism of choice for security-oriented systems. The rights given to a component can be restricted to the absolute minimum it needs ot do its job, as required by the principle of least privilege.

Note that this notion of *object capabilities* is quite different from (and far more powerful than) what Linux calls “capabilities”, which are really access-control lists (ACLs) with system-call granularity. Linux capabilities, like all ACL schemes, suffer from the confused deputy problem, which is at the root of many security breaches. seL4 capabilities do not have this problem.

seL4 capabilities are also not susceptible to the attack of Boebert [1984]; this attack applies to capabilities directly implemented in hardware while seL4’s capabilities are implemented and protected by the kernel.

The seL4 objects referenced by capabilities are:

Endpoints which are used to perform protected function calls;

Address Spaces which provide the sandboxes around components and are thin wrappers around hardware page tables;

Cnodes which store capabilities and represent a component’s access rights;

Notifications which are synchronisation objects (similar to semaphores);

Frames which represent physical memory that can be mapped into address spaces;

Scheduling Contexts which represent the right to access a certain fraction of execution time on a core; and

Untypeds which represent unused (free) physical memory and can be converted (“retyped”) into any of the other types.

Chapter 5

Support for Mixed-Criticality Real-Time Systems

TBD

Chapter 6

Security is No Excuse for Poor Performance

Performance has always been the hallmark of L4 microkernels, and seL4 is no exception. We built seL4 for real-world use, and our aim was not to lose more than 10% in IPC performance relative to the fastest kernels we had before. In fact, seL4 ended up beating the performance of those kernels.

And it beats the performance of any other microkernel. This is a claim that is difficult to prove, as the competition generally holds their performance data close to their chest (for very good reason!)

However we make this performance claim, publicly, at every opportunity. If anyone disagrees they need to present evidence. We also know through a number of informal channels that IPC performance of other systems tends to range between 2 times slower than seL4 to *much* slower, typically around a factor 10.

The few independent performance comparisons certainly back our claim.

Mi et al. [2019] compare the performance of three open-source systems, seL4, Fiasco.OC and Zircon. It finds that seL4 IPC costs are about 10–20% above the hardware limit of kernel entry, address-space switch, kernel exit. Fiasco.OC is more than a factor two slower than seL4 (close to three times the hardware limit), and Zircon is almost nine times slower than seL4.

Gu et al. [2016] compare the performance of CertiKOS to seL4, measuring 3,820 cycles for a round-trip IPC operation in CertiKOS compared to 1,830 in seL4, a factor of two. However, it turns out `sel4bench`, the seL4 benchmarking suite, had at the time a bug in dealing with timers on x86, resulting in exaggerated latencies. The correct seL4 performance figure is around 720 cycles, or more than five times faster than CertiKOS. This is in the context of CertiKOS offering very limited functionality, and no capability-based security.

Chapter 7

Real-World Deployment and Incremental Cyber Retrofit

7.1 General considerations

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 minimise this part of your trusted computing base, and make it as modular as feasible, with each module becoming an seL4-protected CAmkES component.

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. However, even on platforms where the kernel is not verified, the fact that it shares much of its code with a verified platform will give you much higher assurance than with almost any other OS. But keep in mind that without verification the assurance is not what it can be. Also, you must not make any verification claim if you are not using a kernel that is not verified for your platform, or that is in any way modified.

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 specialising on providing support for seL4 adoption. Also, if you develop any generally-useful components yourself, you should seriously consider sharing them with the community under an appropriate open-source license. Those who give back will find it easier to get help from others.

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 natively.

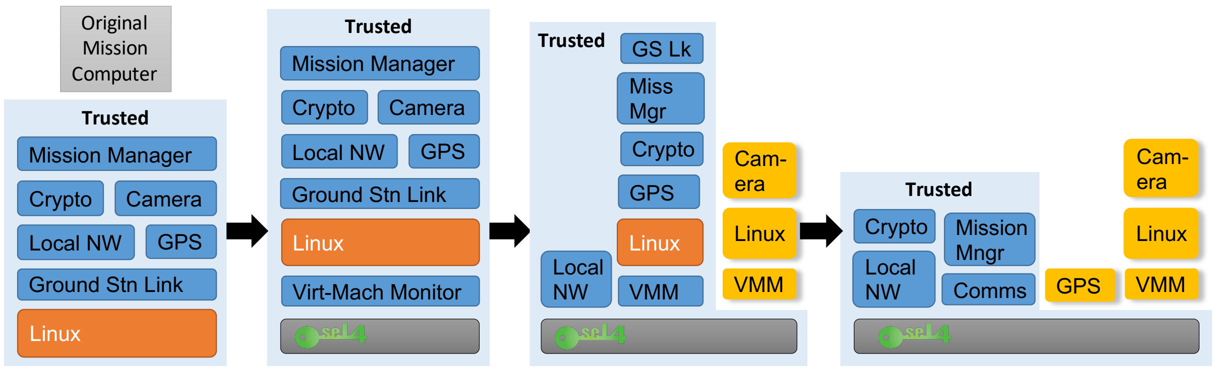

Figure 7.1: Incremental cyber-retrofit on the Boeing ULB mission computer during the DARPA HACMS program.

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

A typical way to proceed is what we call *incremental cyber-retrofit*, a term coined by then DARPA program director John Launchbury. It typically 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.

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.

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 pentesters 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.

Chapter 8

Conclusions

This white paper has hopefully given you a reasonable idea of what seL4 is, what you can do with it, and, importantly, why you would want to use it. I hope this will help you become an active member of the seL4 community, including joining and participating in the seL4 Foundation.

I expect this document will keep evolving, and I am keen on feedback. But most of all, I’m keen to hear of your experience with deploying seL4.

Bibliography

Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic OS design is flawed. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018. ACM SIGOPS. URL `https://ts.data61.csiro.au/publications/csiro_full_text//Biggs_LH_18.pdf`. 4

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011. IEEE Computer Society. URL `https://ts.data61.csiro.au/publications/nicta_full_text/4863.pdf`. 2

William Earl Boebert. On the inability of an unmodified capability machine to enforce the $\star$–property. In *7th DoD/NBS Computer Security Conference*, pages 291–293, September 1984. 16

Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258, Edinburgh, UK, July 2010. Springer. 10

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 653–669, Savannah, GA, US, November 2016. USENIX Association. 18

Gernot Heiser, Gerwin Klein, and Toby Murray. Can we prove time protection? In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 23–29, Bertinoro, Italy, May 2019. ACM. URL `https://ts.data61.csiro.au/publications/csiro_full_text//Heiser_KM_19.pdf`. 11

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, et al. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, US, October 2009. URL `https://ts.data61.csiro.au/publications/papers/Klein_EHA_etal_09.pdf`. 1

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. URL `https://ts.data61.csiro.au/publications/nicta_full_text/7371.pdf`. 1

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM. URL `https://ts.data61.csiro.au/publications/csiro_full_text//Lyons_MAH_18.pdf`. 2, 11

Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In *eurosys*, March 2019. 18

Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 53:812–853, September 2017. URL `https://ts.data61.csiro.au/publications/csiro_full_text//Sewell_KH_17.pdf`. 2