TII Technology Innovation Institute

Why collaboration on a robust Virtual Machine Monitor (VMM) will deliver on the promise of seL4

Secure Systems
Research Center

tii.ae

Innovation for a better world

Contents

Abstract

Security and resiliency are often approached as an afterthought in system design. Developers write code to run on various operating systems and then try to harden all the different vulnerabilities and data leaks that may occur after the fact. This approach is costly and spends considerable resources trying to fix problems after the fact.

Virtualization can play an essential role in providing security to computational systems by isolating execution environments. Virtualization solutions through different Hypervisors have extensively been deployed in Cloud and High-Performance Computing as a way to share the compute resources in these complex environments. However, only a few hypervisors were designed to be deployed at the edge of the network, in devices with fewer computation resources when compared with servers in the Cloud. Among the few lightweight software that can play the hypervisor role, seL4 microkernel stands out by providing a small Trusted Computing Base and formally verified components, enhancing its security.

Researchers have been exploring ways to start with more secure microkernels like L4 and later seL4. Over the last few years, the main development efforts have been put into increasing the maturity of the seL4 kernel itself and not the tools and frameworks that can be hosted on top of it. As such, seL4 lacks the proper support required for many Edge computing virtualization use cases. Thus, a security engineering cost is incurred in adapting seL4 to secure these use cases.

As a result of these tooling and framework gaps in Edge computing virtualization, researchers have turned to other open-source initiatives to improve the security of mainstream hypervisors, like KVM[1] and Xen[2], through Protected KVM and Dom0-less Xen. But these approaches tend to sacrifice either the performance or the feature set of the underlying hypervisor.

Microkernel-based designs like seL4 promise to address security from conception to implementation. seL4 can also be implemented as a base hypervisor with sufficient native and virtual machine applications to support the large number of use cases in Edge computing devices. To achieve the rich support for broad virtual machine environments, a robust virtual machine monitor (VMM) will be required to tune seL4 for these use cases.

We argue that a community-wide effort is needed to build out the standardized VMM infrastructure required to customize seL4 for various use cases. Here we elaborate on some of the design principles and the potential feature set we need to collaborate on to support new high-assurance use cases cost-effectively.

[1] https://www.linux-kvm.org/page/Main_Page
[2] https://xenproject.org/

Why a standardized seL4 VMM?

The most popular kernels were conceived for feature-rich environments like laptops and servers. But supporting all these features consumes considerable overhead and is not suitable for constrained devices in embedded systems. This is important for use cases such as building secure mobile phones, drones, edge devices, and modems. Existing kernels like KVM are monolithic systems with a large Trusted Computing Base (TCB), making it less secure and less efficient to run on constrained devices. Monolithic systems are also not modular which increases the development and maintenance costs of their solutions on embedded devices.

So, researchers developed L4 for more specialized applications with a need for high assurance and efficiency. It works well in tightly controlled environments where there is no need for lots of services. For example, Qualcomm uses a custom version of L4 in their modem and DSP solutions.

In 2006, researchers at the NICTA leveraged recent advances in the L4 microkernel to design a new secure microkernel called seL4 that is provably secure. In 2014 NICTA and General Dynamics open-sourced the kernel in the hopes it would drive innovation in secure systems.

seL4 is a small and simple microkernel-based type-1 hypervisor. It follows the principle of least privilege with capability-based access control mechanisms, performant inter-process communication channels, and userspace services [1]. seL4 works in two different modes, although sharing the same concepts, either as an Operating System (OS) or as a Hypervisor, depending on a set of design-time configurations. The microkernel standalone requires userspace services and VMMs that run along with guest OSes.

Using seL4 in complex production environments[2] brings new challenges, which are often quite different from research-oriented environments. There is a problem of fragmentation since different companies are using their own closed-source tools and userspace libraries.

A standard Virtual Machine Monitor could help address could help satisfy the business goals of seL4 ecosystem members in a standard way. This effort would benefit from the input and authorship of seL4 integrators [3], which are active commercially focused teams in the seL4 ecosystem, including but not limited to Cog Systems[3], Technology Innovation Institute (TII)[4], Hensoldt Cyber[5], DornerWorks[6], and NIO[7]. Each of those businesses has different goals, but there is still a common baseline and philosophy binding them around the VMM when using seL4 as a hypervisor.

At a high level, there are gaps in the seL4 stack, specifically the VMM, userspace, and tooling, which complicate matters for integrators attempting to meet real-world customer use cases. Not all business opportunities require a solution using a VM architecture, but those that do quickly become complex and would benefit enormously from an established standard or reference baseline. The lack of a robust and consistent VMM for seL4 has created a highly fractured environment. Most integrators have their own specialized customer use cases, and they have found that the quickest path is to use a forked and modified VMM.

This practice may have short-term benefits to that integrator. Still, it does not allow the community to benefit from commonality and guarantees that the fork will quickly get old and out of sync with the mainline. For instance, there will be VMM fork features that overlap, and which should be implemented in a standard way for the sake of ongoing community benefit and maintenance.

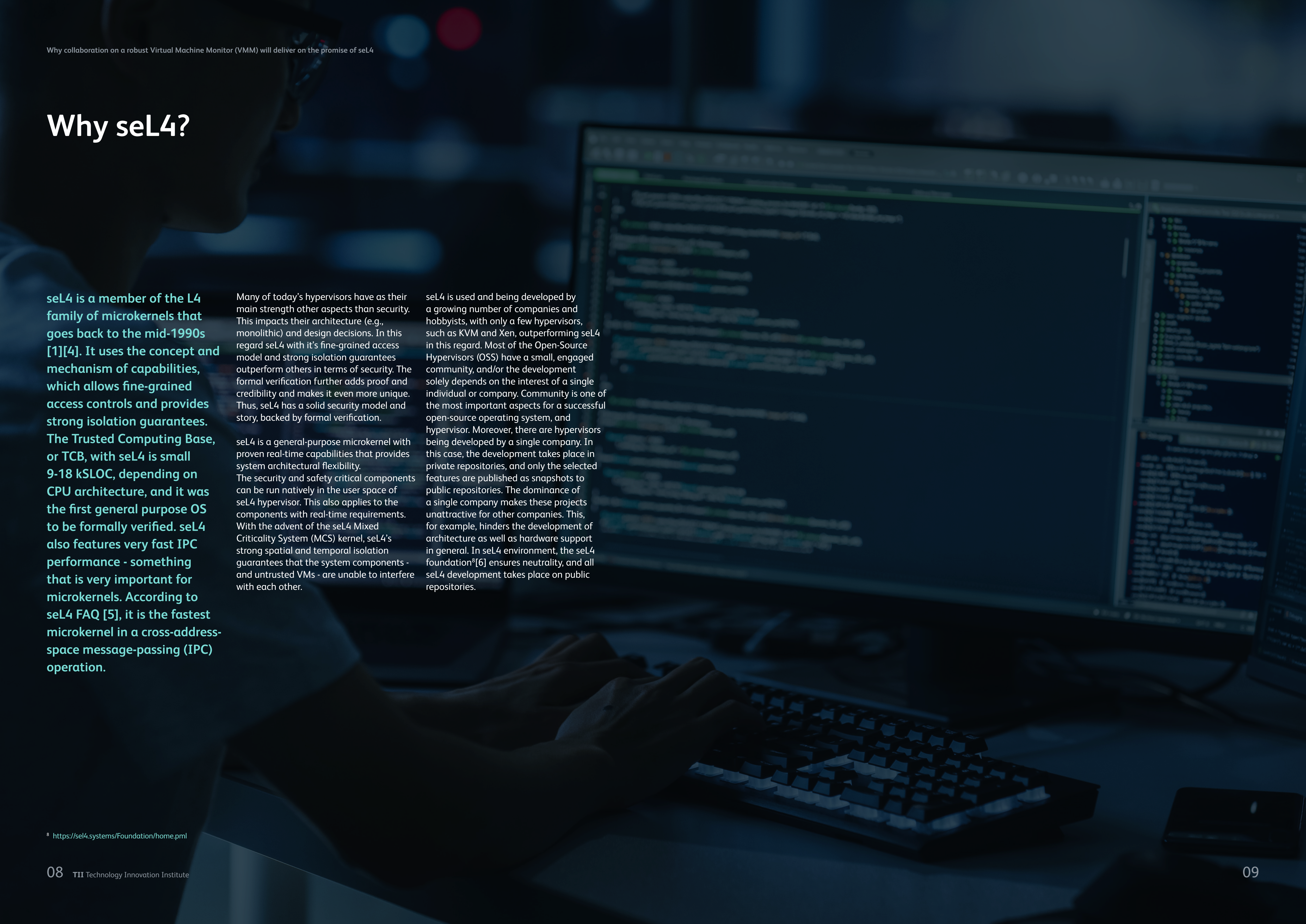
[3] https://cog.systems/
[4] https://www.tii.ae/secure-systems
[5] https://hensoldt-cyber.com/
[6] https://dornerworks.com/
[7] https://www.nio.com/

As a community, a VMM standard can be created which is consistent, maintainable, and works for all the varied business focuses. Critically, this effort must get the conceptual buy-in of the seL4 Foundation and the larger seL4 community for it to be successful going forward. Additionally, a reference implementation, properly maintained, would help to solidify and promote the concept and to provide consistency to complex projects using seL4.

In light of this, the present article has the following goals:

To enroll the community in building a standard VMM as a shared effort by highlighting the discussion about the challenges and potential directions of a standard VMM.

To present the potential key design principles and feature set support toward seL4 VMM standardization. The items shown in this article can be the basis for an extended version with a more comprehensive list of required properties and features.



Why seL4?

seL4 is a member of the L4 family of microkernels that goes back to the mid-1990s [1][4]. It uses the concept and mechanism of capabilities, which allows fine-grained access controls and provides strong isolation guarantees. The Trusted Computing Base, or TCB, with seL4 is small 9-18 kSLOC, depending on CPU architecture, and it was the first general purpose OS to be formally verified. seL4 also features very fast IPC performance - something that is very important for microkernels. According to seL4 FAQ [5], it is the fastest microkernel in a cross-address-space message-passing (IPC) operation.

Many of today's hypervisors have as their main strength other aspects than security. This impacts their architecture (e.g., monolithic) and design decisions. In this regard seL4 with it's fine-grained access model and strong isolation guarantees outperform others in terms of security. The formal verification further adds proof and credibility and makes it even more unique. Thus, seL4 has a solid security model and story, backed by formal verification.

seL4 is a general-purpose microkernel with proven real-time capabilities that provides system architectural flexibility.
The security and safety critical components can be run natively in the user space of seL4 hypervisor. This also applies to the components with real-time requirements. With the advent of the seL4 Mixed Criticality System (MCS) kernel, seL4's strong spatial and temporal isolation guarantees that the system components - and untrusted VMs - are unable to interfere with each other.

seL4 is used and being developed by a growing number of companies and hobbyists, with only a few hypervisors, such as KVM and Xen, outperforming seL4 in this regard. Most of the Open-Source Hypervisors (OSS) have a small, engaged community, and/or the development solely depends on the interest of a single individual or company. Community is one of the most important aspects for a successful open-source operating system, and hypervisor. Moreover, there are hypervisors being developed by a single company. In this case, the development takes place in private repositories, and only the selected features are published as snapshots to public repositories. The dominance of a single company makes these projects unattractive for other companies. This, for example, hinders the development of architecture as well as hardware support in general. In seL4 environment, the seL4 foundation[8][6] ensures neutrality, and all seL4 development takes place on public repositories.

[8] https://sel4.systems/Foundation/home.pml

Virtualization and seL4 VMM

Virtualization

Virtualization is a technique that allows several operating systems to run side-by-side on given hardware [7] [8]. Virtualization brings different kinds of benefits to the environment that it is deployed. One of the benefits would be the heterogeneity that it can bring, being possible to deploy various operating systems and applications in the same hardware [9]. Moreover, it improves the system's security by achieving security by separation [10] [11]. It is achieved as each operating system has its own space, not having an explicit connection with others, keeping software instances isolated. Nevertheless, virtualization requires a software layer responsible for system management, known as a hypervisor.

The hypervisor is a software layer responsible for managing the hardware and explicitly making it available to the upper layers [12]. It has privileged access to the hardware resources and can allocate it accordingly to the operating systems. Examples of hardware resources or devices are: storage memory, network device, I/O devices, etc. The hypervisor is responsible for memory management, scheduling tasks, basic Inter-Process Communication (IPC). For security reasons, the hardware should not be shared directly by different operating systems. However, the hypervisor can provide virtual copies of the same hardware to other operating systems [13]. Many computer architectures have specific privilege levels to run the hypervisor, such as EL2 on ARM and HS-mode on RISC-V. Examples of hypervisors are Xen, KVM, ACRN[9], Bao[10], and seL4.

The hypervisors can be categorized into type-1 and type-2. The type-1 hypervisors runs on bare metal (i.e., directly on the host machine's physical hardware) and type-2 hypervisors, also called hosted hypervisors, runs on top of an operating system [14]. The type-1 hypervisors are considered more secure by not relying on a host operating system. KVM is an example of type-2 hypervisor by running on Linux kernel while seL4 is an example of type-1 hypervisor.

The Virtual Machine Monitor (VMM) is a piece of software that interacts with the hypervisor in the virtualization environment. It has its own responsibilities apart from the hypervisor. The VMM is a user space program that provides emulation for virtual devices and control mechanisms to manage VM Guests (virtual machines) [15]. The VMM enables the virtualization layer to create, manage, and govern operating systems [16]. By running at the user space, the VMM runs at privilege level EL0 on ARM and U-mode on RISC-V. Examples of VMMs are Firecracker[11], crosvm[12], QEMU[13]. Depending on the hypervisor and on the characteristics of the deployed environment, it is possible to have one or multiple VMMs. A common approach is to have one VMM per each operating system Virtual Machine.

Each operating system sits inside a Virtual Machine (VM). A VM behaves like an actual operating system from the point of view of the user, being possible to run applications and interact with it [17]. From the point of view of the hypervisor, a VM has access to a specific set of hardware resources managed by the hypervisor. It is the VMM that makes the bridge from the hardware resources of the hypervisor to make them available to the VM by managing the backend operations [18]. From the scalability perspective, it is possible to have multiple VMs in a virtualization environment, where each VM is isolated from the other by principle. The quantity of VMs depends on the amount of physical resources available for such an environment.

[9] https://projectacrn.org/
[10] http://www.bao-project.org/
[11] https://firecracker-microvm.github.io/
[12] https://chromium.googlesource.com/chromiumos/platform/crosvm/
[13] https://www.qemu.org/

The Figure 1 presents a high-level overview of the components present in a virtualization environment: hardware resources or devices, hypervisor, Virtual Machine Monitor, and Virtual Machine. As we can have different hypervisors, the configuration of the upper layers (VMM and VM) will rely on the chosen hypervisor.

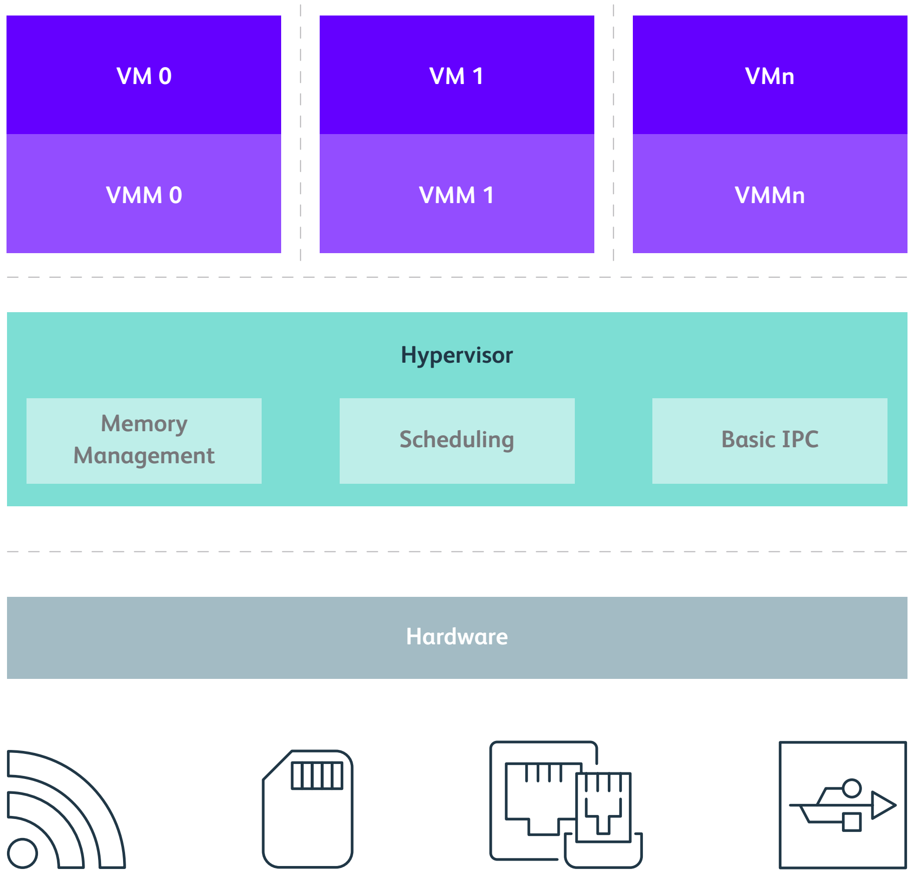


Figure 1 Overview of a virtualization environment

Related hypervisors and VMMs

Apart from seL4, there are other open source hypervisors available in the market. KVM and Xen are examples of traditional hypervisors that have been in the market for more than 15 years and were deployed in different solutions [19] [20]. While both hypervisors are widely used, feature rich and well supported, the huge TCB makes them vulnerable.

KVM (Kernel-based Virtual Machine) is a type-2 hypervisor that added virtualization capabilities to Linux. KVM is integrated in the Linux kernel, thus benefiting from reusing many Linux functionalities such as memory management and CPU scheduling. The downside of it is the huge TCB that comes along KVM. The KVM was originally built for x86 architecture and then ported to ARM [21]. The KVM on ARM implementation has been split in the so-called Highvisor and Lowvisor. The Highvisor lies in ARMs kernel space (EL1) and handles most of the hypervisor functionalities. The Lowvisor resides in hypervisor mode (EL2) and is responsible for enforcing isolation, handling hypervisor traps and performing the world switches (context execution switches between VMs and host) [22].

Xen is defined as a type-1 hypervisor. The x86 version of Xen, is a bare-metal hypervisor that supports both fully virtualized and para-virtualized guests. On ARM, the code for Xen is reduced to one type of guest which uses para-virtualized drivers and the ARM virtualization extensions [22]. The Xen hypervisor resides in hypervisor mode. On top of it, everything is executed as a guest placed in different domains. The most privileged domain is called Dom0, it has access to hardware and runs Linux to manage other guests, named DomU[14]. A DomU is the counterpart to Dom0; it is an unprivileged domain with (by default) no access to the hardware. DomU's use Dom0's para-virtualized services through Xen PV calls. Recently, the Dom0-less variant was introduced. With Dom0-less[15], Xen boots selected VMs in parallel on different physical CPU cores directly from the hypervisor at boot time. Xen Dom0-less is a natural fit for static partitioning, where a user splits the platform into multiple isolated domains and runs different operating systems on each domain.

Traditionally, KVM and Xen hypervisors were designed to be deployed at the Cloud Computing level to provide virtualization to high-density machines. However, recent solutions were developed to use those kinds of hypervisors also at the Edge level [23] [24], being able to have such virtualization solutions in devices with less processing power than the servers at the Cloud [25] [26] [21]. There are also hypervisors that were designed in a lightweight manner, with the intention to be applied in resource-constrained environments at the Edge level. Examples of lightweight hypervisors are Bao [27] and ACRN [28], among others.

Bao is a lightweight bare-metal hypervisor designed for mixed-criticality systems. It strongly focuses on isolation for fault-containment and real-time behavior. Its implementation comprises a thin layer of privileged software leveraging ISA virtualization support to implement a static partitioning hypervisor architecture [27]. ACRN targets itself to IoT and Egde systems, placing a lot of emphasis to performance, real-time capabilities and functional safety. ACRN currently only supports x86 architectures, and as it is mainly backed by Intel, support to other architectures may not appear any time soon [28].

Gunyah[16] is a relatively new hypervisor by Qualcomm. It is a microkernel design with capability access controls. Gunyah being a new project has a very limited HW support, and practically non-existent community outside Qualcomm. KVMs[17] is an aarch64 specific hypervisor, building upon popular KVM, bringing a lot of flexibility for example in terms of choice of VMMs. Thanks to a small size, it is possible to formally verify hypervisor EL2 functionality [29]. While there are a lot of benefits, it is limited to on CPU architecture, and maintaining KVMs patch series across several versions of Linux kernel may become an issue.

KVM relies in user space tools such as the Quick Emulator (QEMU) [30] to serve as VMM and instantiating virtual machines. In the KVM paradigm guests are seen by the host as normal POSIX processes, with QEMU residing in the host userspace and utilizing KVM to take advantage of the hardware virtualization extensions [22]. Other VMM can be use on top of KVM, as Firecracker, Cloud Hypervisor and crosvm. Firecracker uses the KVM to create and manage microVMs. Firecracker has a minimalist design. It excludes unnecessary devices and guest functionality to reduce the memory footprint and attack surface area of each microVM [31]. Cloud Hypervisor focuses on exclusively running modern, cloud workloads, on top of a limited set of hardware architectures and platforms. Cloud workloads refers to those that are usually run by customers inside a cloud provider. Cloud Hypervisor is implemented in Rust and is based on the rust-vmm[18] crates. The crosvm VMM is intended to run Linux guests, originally as a security boundary for running native applications on the Chrome OS platform. Compared to QEMU, crosvm does not emulate architectures or real hardware, instead concentrating on para-virtualized devices, such as the VirtIO [32] standard.

[14] https://wiki.xenproject.org/wiki/DomU
[15] https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/
[16] https://github.com/quic/gunyah-hypervisor
[17] https://github.com/jkrh/kvms
[18] https://github.com/rust-vmm

seL4 VMM

An Operating System (OS) microkernel is a minimal core of an OS, reducing the code executing at higher privilege to a minimum. The seL4 is a microkernel and hypervisor capable of providing virtualization support [1]. It has a small trusted computing base (TCB), making a minor surface attack compared to traditional hypervisors such as KVM and Xen.

The seL4 supports virtualization by providing specifically two libraries: (i) \textit{libsel4vm}, and (ii) libsel4vmmplatsupport [33]. The first (i) is a guest hardware virtualization library for x86 (ia32) and ARM (ARMv7/w virtualization extensions \& ARMv8) architectures. The second (ii) is a library containing various VMM utilities and drivers that can be used to construct a guest VM on a supported platform. These libraries can be utilized to construct VMM servers through providing useful interfaces to create VM instances, manage guest physical address spaces and provide virtual device support (e.g., VirtIO Net, VirtIO PCI, VirtIO Console). Projects exist that make use of the seL4 virtualization infrastructure, supporting the provision of virtualization environments. Examples of those kinds of projects are CAmkES and Core Platform.

The CAmkES project is a framework for running virtualized Linux guests on seL4 for ARM and x86 platforms. The camkes-vm implements a virtual machine monitor (VMM) server, facilitating the initialization, booting and run-time management of a guest OS [34]. The CAmkES project provides an easy way to run different virtualization examples with one or more VMs and different applications. It also provides a way how to passthrough devices in such environments. One drawback of such a framework is that it is only possible to run static VMs, in which the VM configuration should be defined at design time.

CAmkES proved to be too complex, static and maintenance intensive. Because of this reason, many projects and companies have rolled their own user space. As the VMM is in the user space, the challenges and limitations are imminent in the virtualization too. To remedy the situation, the seL4 community is introducing seL4 Core Platform [35] [36], or seL4cp, and seL4 Device Driver Framework[19], or sDDF. The two new components are attempts to fix the shortcomings of CAmkES. This also means that the VMM parts will be changed significantly too.

The Core Platform provides the following abstractions: protection domain (PD), communication channel (CC), memory region (MR), and notification and protected procedure call (PPC). A VM is a special case of a PD with extra, virtualization-related attributes. The original version of the seL4CP was fully static, in that all code had to be fixed at system build time, and PDs could not be restarted. The addition of dynamic features is in progress [37]. The seL4 Device Driver Framework (sDDF) provides libraries, interfaces and protocols for writing/porting device drivers to run as performant user level programs on seL4. The sDDF also aims to be extended to a device virtualization framework (sDVF) for sharing devices between virtual machines and native components on seL4.

Even though the seL4 VMM exists and is available to use, it lacks in providing essential features for virtualization support in complex scenarios. Moreover, its fragmentation by different closed-source deployments makes the mainline depreciate fast. Thus, it is necessary to discuss the desired features for such a standard VMM.

[19] https://sel4.atlassian.net/browse/RFC-12

Philosophy of a Standard VMM

It should be immediately obvious that even a community as small as the commercial users of seL4 will have difficulty agreeing to an all-encompassing standard. Thus, what is proposed is to establish a driving philosophy for the design of a baseline VMM rather than prescribe a specific system architecture. There is the need to discuss the possible missing features of the existing seL4 VMM [33] concerning a standard VMM, more so than a prescription for the right way to do it. Indeed, this will entail recommending high-level architecture patterns but cannot lock an adopter into specific implementations. Each adopting integrator will inevitably start from the new standard and refine the implementation for their use case. One size does not fit all, so customization will always occur. The effort here is to close the gap between the current VMM baseline and the point of necessary deviation. Refinement should only be necessary to cover specific requirements and edge cases highly unlikely to appear in multiple projects across the integrator community.

For this discussion, driving philosophical concepts can be roughly binned into Design Principles and Feature Support Tenets.
The Design Principles and Feature Support Tenets were defined based on features present in already available VMMs and the technical challenges they posed. A deeper discussion about the Design Principles and Feature Support Tenets will be needed before implementation at seL4 mainline.
This list intends to serve as a starting point for discussing such topics.

Design Principles

Five major design principles are recommended as potential directions towards the standard VMM. They are motivated to be open, modular, portable, scalable, and secure.

Official and Open Source

The existing seL4 VMM [33] employs an open-source license, and any new implementations under the proposed standard should remain in accordance with this approach. This applies to all the code up to the point of necessary differentiation. Individual integrators should always retain the ability to keep closed-sourced their highly specialized or trade secret modifications. This strikes a balance between business needs such as maintaining a competitive edge and fully participating in a collaborative community around a common baseline. Open sourcing the standard VMM is essential for the seL4 community to engage collaboratively and improve the VMM by either contributing to the source code repository or using and learning from it.

It is recommended to place the standard VMM baseline under the purview of the seL4 Foundation to benefit from the structure and governance of that organization. The desire is that it will gain in stature as well, as the current VMM is a second-class citizen in the community. Alongside the source code, the Foundation should periodically publish reports about major updates and possible new directions as new technologies mature. In this way, it will help to maintain a long-term roadmap to incorporate new features such as ARMv9 Realms [38], for instance.

Modular: Maintainable and Upgradable

It is expected that the standardized VMM would be deployed in heterogeneous environments and scenarios under quite varied use cases. This will require flexibility in aspects such as system architecture, hardware requirements, performance, etc. It is essential to follow a modular design approach to guarantee the applicability of the VMM in any of those variants.

In implementing the VMM modularly, it is essential to achieve its readability by following the C4 (Context, Containers, Components, and Code) model, for instance. The C4 model is an "abstraction-first" approach to diagramming software architecture [39]. It decomposes the system so community members can pick and choose components for their project. The Context shows how the software system in scope fits into the environment around it. Containers inside a Context define the high-level technical building blocks. A Component is a zoom-in to an individual Container and shows its responsibilities and implementation details. Finally, Code is a specific description of how a Container is implemented. The modular approach makes it possible for integrators to define the Context, Containers, Components, and Code that must be pieced together for a VMM to support specific features, making their VMM highly customized to their end goal.

Portable: Hardware Independence

The seL4 community has done a phenomenal job in supporting seL4 across a variety of hardware platforms [40]. The VMM should be generic enough to support them as well. This may be a lofty goal.

One good starting point may be to design and write the VMM to support popular hardware such as ARM, x86, and RISC-V Instruction Set Architecture. This will ensure the standard VMM is not explicitly linked to a specific set of hardware characteristics. Of course, different ISAs may impose architectural differences. However, there is the need for a minimal and modular VMM that could be easily moved from 4 core ARM SoC (big.LITTLE) to a 48-core Thread Ripper AMD x86, as an example.

The standard VMM could be seen as a baseline for different hardware implementations. Obviously, the baseline will not take advantage of all the platforms' hardware features. However, it can be used for Proof-of-Concept implementation and learning purposes for being easy to deploy on different platforms.

Additionally, it is essential to consider and accommodate the rather large differences architecture-wise, even with the same ISA implementation. For example, Codasip[20] and SiFive[21] implementations of RISC-V have non-ignorable differences, while ARM implementations from Qualcomm, Samsung, and NXP exhibit wildly different behavior [41]. Though SoC vendors may be compliant with the ISA specification, there usually is some collection of deviations or enhancements present, often implemented as a black-box binary. Areas of concern include control of the system's Memory Management Units, Generic Interrupt Controller, TPM/TEE, secure boot process, and access to the appropriate privilege level for the seL4 kernel (e.g., EL2 for Qualcomm-ARM).

[20] https://sel4.atlassian.net/browse/RFC-12

[21] https://www.sifive.com/

Scalable: Application-agnostic

A standard VMM should be scalable in the sense that it needs to be able to support several applications running on top for different specific purposes. Different applications may have a distinct set of requirements such as performance, safety, security, or real-time. The VMM should be able to meet those requirements and provide a way for the applications to reach them. Moreover, the VMM should guarantee that the applications will run as expected, being able to initiate and finish the tasks successfully. A VMM scheduler should be responsible for balancing the loads and ensuring that no application (i.e., thread) is left unattended.

The scalability of the systems is also tied to their performance. In light of this, it is essential that the VMM supports from one to an arbitrary number of processing units or cores. The existing seL4 VMM does not support multiprocessing and consequently highly restricts the number of applications that can be run atop. Enabling multiprocessing would help achieve better performance, thus improving the scalability of the system performance as a whole. We discuss in detail the possibilities to enable multicore VMM further in this paper in the Multicore & Time Isolation section.

Secure by Design

A security by design approach requires enforcing as much isolation as possible at every level of implementation. A standard seL4 VMM implementation should support one VMM instance per VM. Even though this approach is well followed by most of the integrators and supported by seL4, it is important to highlight its benefits. This approach improves both scalability and security of the solution.

If a guest OS is compromised, it opens an attack vector toward the VMM. However, the risk is limited if there is a dedicated VMM per VM. The other VMs, their VMMs, and guest OSes are completely isolated by the stage 2 translation. This assumes a formally verified kernel and that the translation tables or the memory areas the tables point to are distinct for each VM.

Though this approach is already common today, some integrators do not always implement it for time-to-market pressure, reusable code, or other unusual circumstances. Support for this design should be standardized so that the enabling code can be considered boilerplate and easily consumed. Figure 2 shows a representation of a secured by design architecture, with one VMM per VM. Even though the VMM has more direct interaction with the hypervisor, it is placed in the User Mode. The VMs are present at both User and Kernel modes, as they can have applications and drivers, respectively.

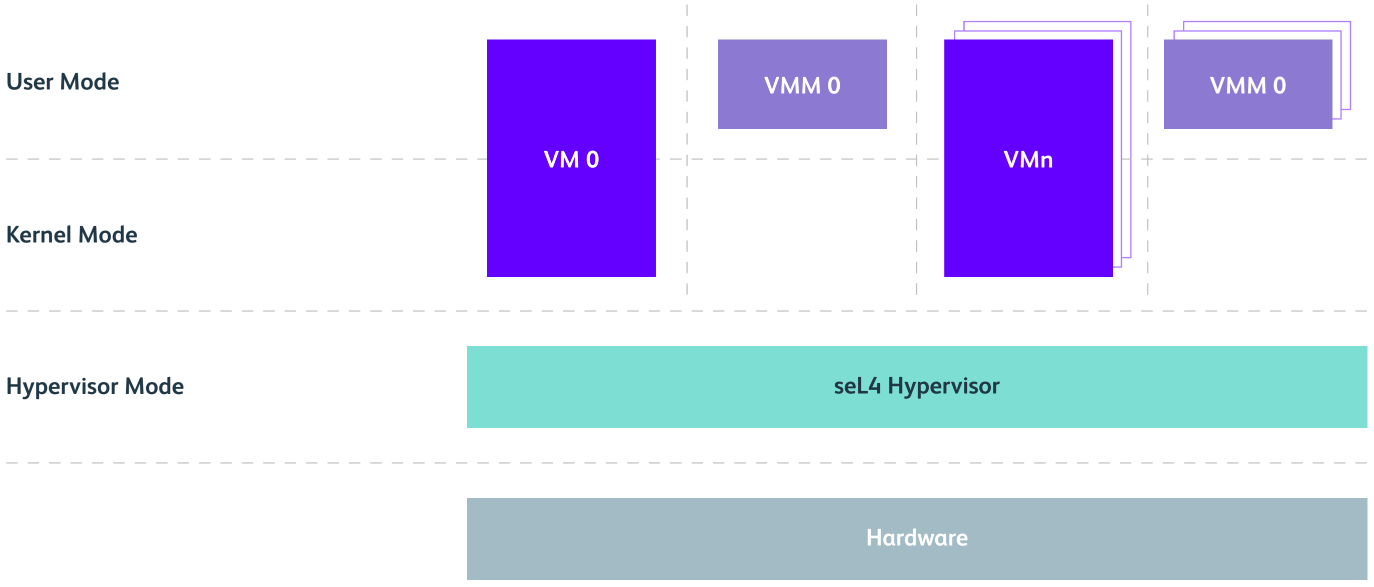


Figure 2 Example of an architecture with one VMM per VM

Feature Set Support Tenets

Four major features are recommended as potential directions towards the standard VMM to support hardware mechanisms and provide security and performance benefits.

System Configuration

Currently, there are two main approaches to facilitate the system configuration when running virtual environments on top of seL4. The first to be introduced was CAmkES, that stands for Component Architecture for microkernel-based Embedded Systems [42]. The second one is seL4 Core Platform (seL4CP) [35]. The Core Platform, which was recently introduced, intends to be the standard for such virtual environments on top of seL4. Thus, the CAmkES is being deprecated.

CAmkES is a software development and runtime framework for quickly and reliably building microkernel-based multiserver (operating) systems [42]. Currently, using the CAmkES framework with the VMM will result in a fully static configuration. The VMs must be defined and configured during the build. This also includes defining the RAM area. It is designed to achieve security guarantees so as not to allow post-build modifications to the number of running VMs and their interconnections. This is a highly desirable aspect when the use case calls for it. However, it can be inflexible and even short-sighted when the nature of the user experience requires dynamic configuration, i.e. no dynamic start/stop/restart capability.

It is often necessary to have a more dynamic seL4-based environment for the purpose of allowing better usability, modularity, or even scalability. The Core Platform is an operating system (OS) personality for the seL4 microkernel. The Core Platform makes seL4-based systems easy to develop and deploy within the target areas. It can be used to bring up VMs on top of seL4. Core Platform promises to deliver dynamic features to the seL4 environment [35]. However, it is still in progress with ongoing virtualization features in development[22]. The Trustworthy Systems - UNSW[23] group also intends to formally verify two core aspects of the seL4 Core Platform[24]: (i) correctness of the implementation, i.e. its abstractions function as specified, and (ii) correctness of the system initialization, i.e. the collection of underlying seL4 objects are fairly represented by the system specification.

A new VMM standard should enhance the existing static build approach with a build-time specification stating that dynamic configurations are also permitted. They could be limited by providing build-time parameters for acceptable configurations. To achieve a dynamic environment, it should be possible to use the seL4 mechanisms for transferring/revoking capabilities to the entities during runtime, providing a potential implementation mechanism for this feature. It may also be an option to build a core common component to serve as an "admin VM" for dynamic configurations, even subjecting it to some degree of formal methods verification. This is anticipated to be an area of much research and prototyping to achieve the desired balance of security and flexibility.

Multicore & Time Isolation

One of the key aspects of virtualization is the need for efficiency, where multiprocessing configurations play an important role. Although multicore support is a complex engineering task, it should be supported in its simplest shape to avoid contention and potential deadlocks. Different physical CPUs (pCPUs) can be enabled by the kernel (in a Symmetric Multiprocessing - SMP configuration) in order to allocate them to a different system running threads according to the use-case application requirements. Next, we present potential multi-core configurations that a standard VMM should be able to support using a clear multiprocessing protocol:

[22] https://github.com/Ivan-Velickovic/sel4cp/tree/virtualisation_support
[23] https://trustworthy.systems/about/
[24] https://trustworthy.systems/projects/TS/sel4cp/verification

Direct Mapping Configuration:

Multiple single-core VMs running concurrently and physically distributed over dedicated CPUs. Figure 3 shows the representation of the Direct Mapping Configuration approach.

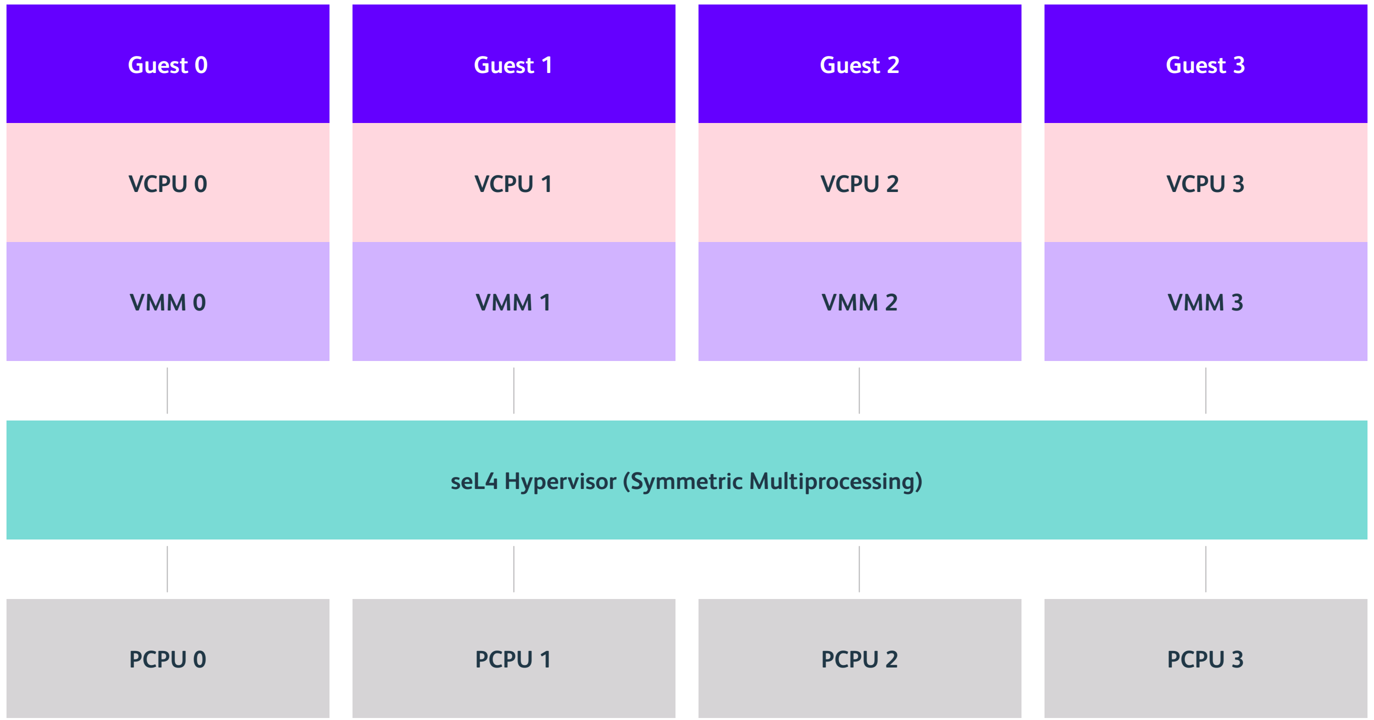


Figure 3 Direct Mapping Configuration overview

Hybrid Multiprocessing Configuration:

It can have multiple single-core VMs running in dedicated CPUs as the Direct Mapping Configuration, however, it can also have multicore VMs running in different CPUs. Figure 4 shows the representation of the Hybrid Multiprocessing Configuration approach.

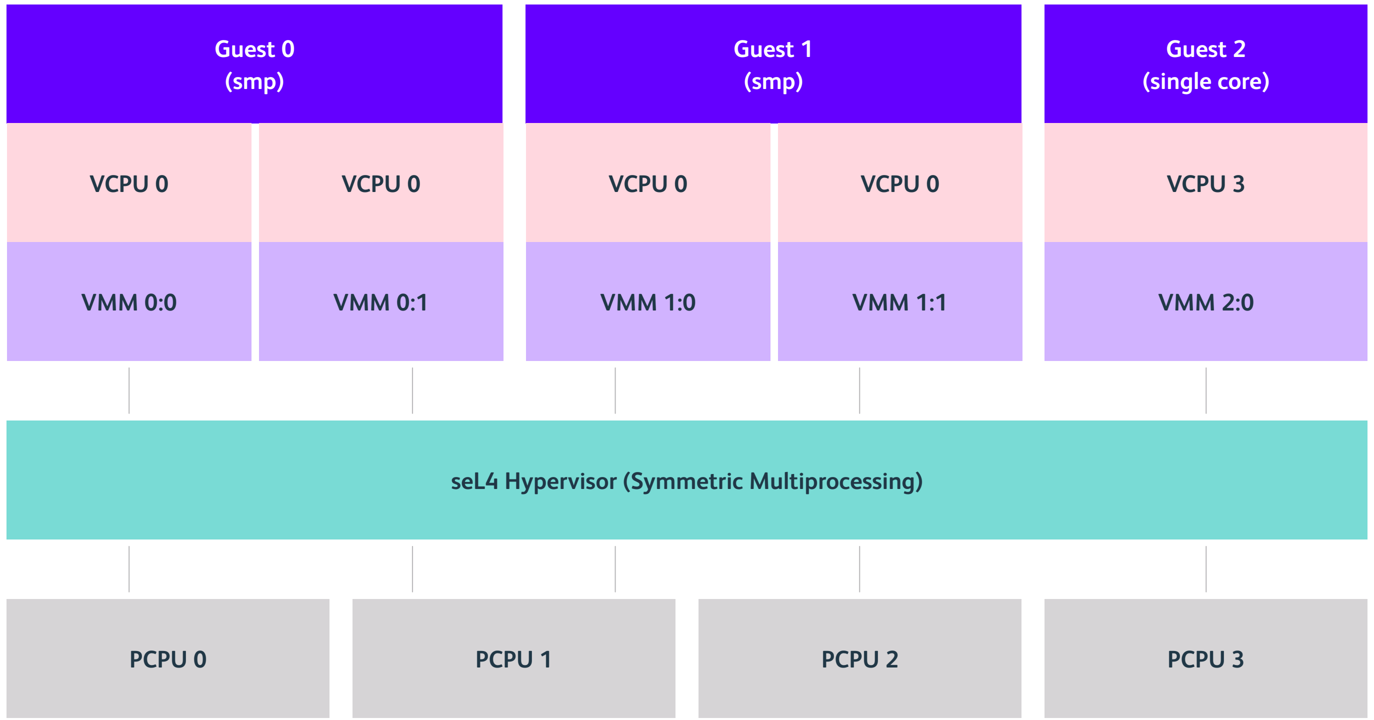


Figure 4 Hybrid Multiprocessing Configuration overview

The two depicted configurations are examples for future reference of a standard VMM, but it is not strictly limited. Most, if not all, current and near-future use cases are covered by a model where there are multicore VMMs that are pinned to exclusive cores and unicore VMMs that can be multiplexed on a core. Ideally, it would be up to the system designer to decide which configuration to use. It could be either static or dynamic, enabling switching from a given configuration to another in run-time. The selected configuration will affect several threads in execution.

In the seL4 context, threads can be running either Native apps, OSes, and/or VMMs. The former is typically used to run device drivers or support libraries. OSes are using threads running over virtual abstractions, or VMs, while VMMs are creating and multiplexing these abstractions to be able to encapsulate OSes. They all require an abstraction representing the pCPU time but differ from the supported execution level and their scope over other system components. For example, a VMM can access the VM internals but not the opposite.

Other features that are likely required by multicore design are:

vCPUs Scheduling: The ability to schedule threads on each pCPU based on their priority, credit-based time slicing, or budgeting depending on the algorithm selected. As an example, It could be a design configuration whether it supports vCPU migration (a vCPU switching from pCPU id:0 to id:1) with also the possibility to tie up a set of the vCPUs to pCPUs. Another potential configuration is the static partitioning one, where all the vCPUs are assigned to pCPUs at design-time and are immutable at run-time. In addition, having dynamic and static VMs configuration in a hybrid mode could be something to support. A multiprocessing protocol with acquire/release ownership of vCPUs should be supported. The seL4 kernel has a scheduler that chooses the next thread to run on a specific processing core, and is a priority-based round-robin scheduler. The scheduler picks threads that are runnable: that is, resumed, and not blocked on any IPC operation. The scheduler picks the highest-priority, runnable thread (0~255). When multiple TCBs are runnable and have the same priority, they are scheduled in a first-in, first-out round-robin fashion. The seL4 kernel scheduler could be extended for the VMMs.

pIRQ/vIRQs ownership: physical interrupts (pIRQs) shall be virtualized (vIRQs) and require a multiprocessing protocol with simple acquire/release ownership of interrupts per pCPU/vCPU targets. Besides, support hardware-assisted interrupt-controllers with multicore support is required.

Inter-vCPU/inter-pCPU communication: Another key aspect of multiprocessing architectures is the ability to communicate between pCPUs. Also, with equal importance, communication between vCPUs results in not only inter-pCPU but also inter-vCPU communication. Communication is very important in multiprocessing protocols, but it should be designed in a way that is simple to verify and validate.

Memory Isolation

All virtual services need to access memory to do their work. The memory needs to be a shared resource used to run code and services. We need to strike the appropriate balance between ensuring that memory is a shared resource and is securely accessed by competing engines and services.

Memory isolation is critical to enforce the security properties such as VMs confidentiality and integrity. Hardware-enforced and partial microkernel access-controlled memory translation and protection between VMs/VMMs and Native Apps are key security requirements for security-critical use-cases. Support for hardware-assisted virtualization (extended Page Tables or second-stage) MMU should be an integral part of the standard VMM.

Next, are some features for future reference that can leverage such hardware for memory isolation: (i) configurable VM Virtual Address Space (VAS); (ii) device memory isolation; and (iii) cache isolation.

Configurable VM Virtual Address Space: Multiple virtual Address Spaces are an important feature supported by high-end processors and have the same paramount importance for hardware-assisted virtualization. There should be different Virtual Address Spaces for different software entities: Hypervisor, VMM, and their respective VMs. User-controlled and configurable address spaces are important features for VMs. For example, (i) setting up a contiguous virtual address space ranges from fragmented physical memory as well as small memory segments shared with other VMs, (ii) hiding physical platform memory segments or devices from the VMs, (iii) no need to recompile a non-relocatable VM image.

Device memory isolation by hardware-support or purely software: Devices that are connected to the System-on-Chip (SoC) bus interconnection and are masters can trigger read and write DMA transactions from and to the main memory. This memory, typically DRAM, is physically shared and logically partitioned among different VMs by the hypervisor. Some requirements could be met in a standard VMM: (i) a device can only access the memory of the VM it belongs to; (ii) the device could understand the virtual AS of its VM; and (iii) the virtualization layer could intercept all accesses to the device and decode only those that intend to configure its DMA engine in order to do the corresponding translation if needed, and control access to specific physical memory regions. In order to meet these three requirements a standard VMM requires support for either an IOMMU (with one or two stage translation regimes) or software mechanisms for mediation.

Cache isolation through page-coloring: Micro-architectural hardware features like pipelines, branch predictors, and caches are typically available and essential for well performant CPUs. These hardware enhancements are mostly seen as software-transparent but currently leaving traces behind and opening up backdoors that can be exploited by attackers to break memory isolation and consequently compromising the memory confidentiality of a given VM. One mitigation for this problem is to apply page coloring in software and could be an optional feature supported by a standard VMM. Page coloring is meant to map frame pages to different VMs without colliding into the same allocated cache line. A given cache allocated by a VM cannot evict a previously allocated cache line by another VM. This technique, by partitioning the cache in different colors, can protect to some extent (shared caches) against timing cache-based side channel attacks, however, it strongly depends on some architectural/platform parameter limitations such as cache size, number of ways and page size granularity used to configure the virtual address space. L1 cache is typically small and private to the pCPU while L2 cache is typically bigger and seen as the last level of cache that is shared among several pCPUs. It would be possible to assign a color to a set of VMs based on their criticality level. For example, assuming the hardware limits the system to encode up to 4 colors, where one color can be shared by a set of non-critical VMs, other for real-time VM for deterministic behavior, and the other two for a security- and performance-critical VM that requires increased cache utilization and at the same isolation against side-channel attacks.

Hypervisor-agnostic I/O Virtualization and its derivations

Many security use-cases require virtualization environments with reduced privilege such that only specific VMs, called driver VMs, can directly access hardware resources while the others, called User VMs, run in a driverless mode since device drivers are seen today as a major source of bugs. A compromise caused by exploitation of a driver bug can be contained in its own VM. Typically, in such environments, any VM that will potentially run unknown code and/or untrusted applications may require isolation from key device drivers sequestered into their dedicated VMs. Inter-VM communication, including access to the devices, must be done by proxy over well-known and managed interfaces. This approach requires a combination of VM kernel modifications and VMM modules to be able to communicate and share basic hardware devices over virtual interfaces.

The OASIS collaboration community manages the set of VirtIO standards [32] that are implemented to various degrees by Linux and Android. Given the excellent support, it is recommended to adopt VirtIO implementations for multiple interfaces in the standard VMM. Support for standardized VirtIO server implementations in the VMM would be a meaningful complement to guest OS clients. For instance, the VirtIO-Net server in the VMM could store a table of MAC addresses, creating a virtual switch. In the case of the VirtIO-Block server, the VMM could terminate VirtIO-Block requests so that address mappings are not known by the user-facing guest OS, then start up another request to the VM containing the device driver to perform the actual write. For instance, in complex architectures with more than one guest OS accessible from the user perspective, VMM VirtIO servers could also handle multiplexing access to various devices between VMs, creating a "multi-persona" capability.

Among the possibilities of implementing VirtIO interfaces, the following items present examples of how it can be used and integrated with a standard VMM:

[25] https://trustworthy.systems/projects/TS/drivers/

VirtIO can be used for interfacing VMs with host device drivers. It can support VirtIO driver backends and frontends on top of seL4. VirtIO interfaces can be connected to open-source technologies such as QEMU, crosvm, and Firecracker, among others. In this scenario, the open-source technologies will execute in the user space of a VM different from the one using the device itself. This approach helps in achieving reusability, portability, and scalability. Figure 5 shows the representation of such an approach considering a VirtIO Net scenario in which a Guest VM consumes the services provided by a back-end Host VM.

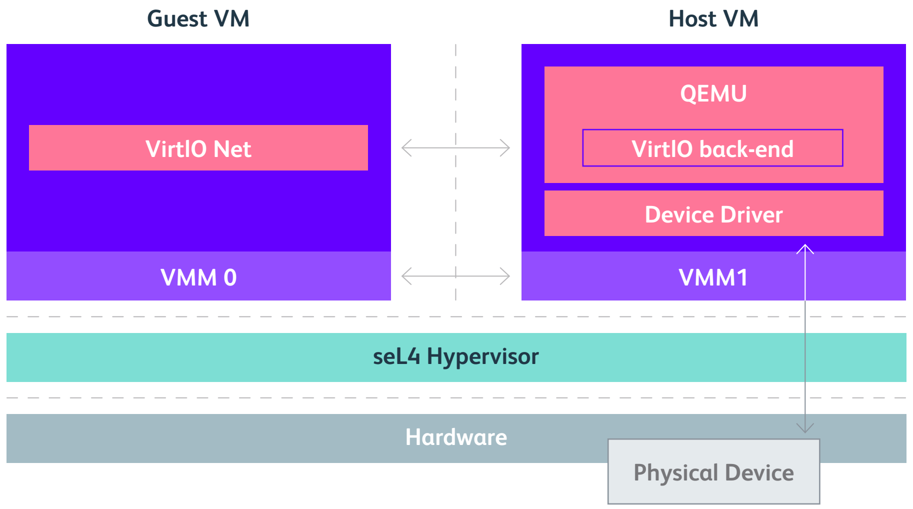


Figure 5 VirtIO drivers example on top of seL4 hypervisor

VirtIO interfaces can be connected to formal verified native device drivers. The use of such kinds of device drivers increases the security of the whole system. Moreover, the verified device drivers can be multiplexed to different accesses, switching device access between multiple clients. The multiplexer is transparent to native clients, as it uses the same protocol as the (native) clients use to access an exclusively owned device. Figure 6 shows the representation of a device virtualization through a multiplexer. In this example each device has a single driver, encapsulated either in a native component or a virtual machine, and is multiplexed securely between clients.[25]

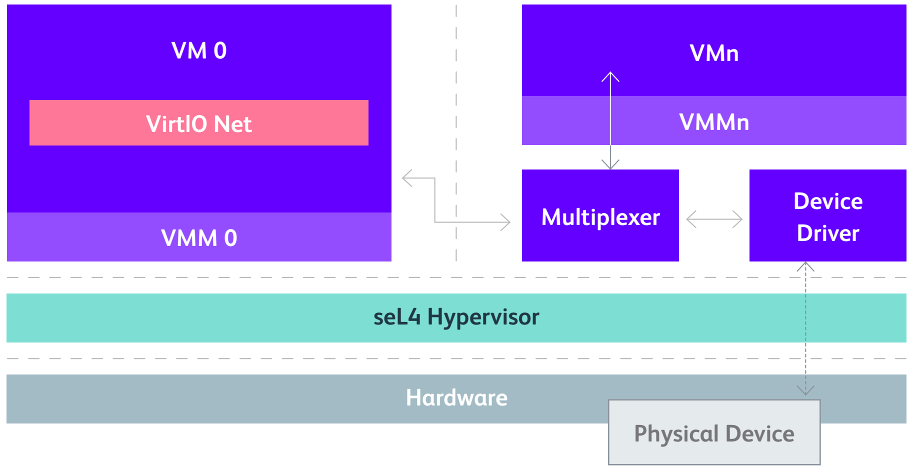


Figure 6 VirtIO interfaces considering a formally verified Device Driver

[25] https://trustworthy.systems/projects/TS/drivers/

VirtIO also includes standards for Touch, Audio, GPU, and a generic VirtIO-Socket interface which can be used to pass data of any form. Standardized implementations for these are not mature or widely available outside of the automotive use case. OpenSynergy actively worked with Google and Qualcomm to include these interfaces in Android Auto [43]. It may be possible for the seL4 community to expand those implementations to other areas through customer-funded projects.

Discussion Topics

VMM API

Apart from the previously mentioned topics, a seL4 standard VMM could also be a programmable API rather than something configured with static Domain Specific Language (DSL) during compilation (e.g., CAmkES). The API makes it possible to wrap the functionality to any compile-time DSLs, custom native services and enables run-time dynamism. The API could have a compile-time configuration for enabling/disabling dynamic features. It should build upon layers so one can use the low-level APIs with all seL4-specific complexity involved, but the API should keep the seL4-specific things minimal at a high level.

An API would make it possible for some elements of the VMM not to be wrapped in a runtime context, like it is now, because it then already makes an assumption about the architecture. That assumption might not be what most integrators (i.e., companies) are after. Let's take KVM as an example. If KVM would provide more than basic constructs and include runtime context (essentially VMM), then we would not be able to have different VMMs (QEMU, crosvm, cloud-hypervisor). It does not mean that there is not an API already in the seL4 environment. But it is pretty fragmented and not uniform as one might expect.

The integrators could have an option to use the seL4 VMM (i.e., with characteristics similar to the ones presented in this article) and also the VMM API to have a more diverse virtualization environment. There is a certain minimal subset that a VMM must handle, like handling the hardware virtualization of Generic Interrupt Controller Architecture (GIC) and handling faults. However, it should also be possible to define where VirtIO-console should be handled or that VirtIO-blk device must be handled by QEMU in some VM. If someone has a native VirtIO-backend for some of those examples, it should be possible to use it.

With the seL4 VMM API, it is possible to follow the one VM per VMM "rule" as it is a safer approach from a trust point of view. We could have different flavors of VMMs, such as QEMU, crosvm, and cloud-hypervisor, as each one of them will have its strengths and weakness [44] [45].

Formal Methods

No discussion of an seL4 adjacent system is complete without consideration for the impact of formal methods. Since this discussion is driven by the need for a VMM which can handle complex, real-world use cases, an integrator would likely be using a hardware platform for which seL4 does not yet support formal methods, such as aarch64 or a multicore configuration. In this case, the effect of formal verification is a moot point. However, in the future, or for a simpler configuration, we can still assess the impact.

Currently, the VMM is assigned per each VM, and thus it is in the VM's Trusted Computing Base. If we consider the scenario in which it is possible to use a VMM API to run VMMs from different flavors, the formal verification would rely just on the minimal part responsible to execute those VMMs and not in the VMM itself. The VMM is considered part of a guest for the purposes of formal methods, so maintaining the proofs would be challenging. However, there may be a specific case to be made for the standard VMM to be shared across all VMs in a particular system. In that instance, the VMM could be subject to formal methods verification. However, it would be a complex and costly undertaking and goes against the "One VMM Per VM" principle detailed previously in this document.

Parts of the standard VMM could be subject to verification, an example could be the device drivers. The Device Virtualization on seL4 project[26] has the long-term goal of formal verify device drivers, which is enabled by the strong isolation provided for usermode drivers on seL4, which allows verifying drivers in isolation. The seL4 Core Platform has a working in progress project[27] to formally verify two core aspects of it: (i) correctness of the implementation (i.e. its abstractions function as specified), and (ii) correctness of the system initialization (i.e. the collection of underlying seL4 objects are fairly represented by the system specification).

[26] https://trustworthy.systems/projects/TS/drivers/devvirt

[27] https://trustworthy.systems/projects/TS/sel4cp/verification

Next Steps

The Technology Innovation Institute (TII) can draw from its experience building hypervisor-based systems of significant complexity to conclude that the existing VMM baseline is not ideal. It lacks support for many practical design features. We can remedy these defects by collaborating to build a new VMM standard. We should consider important principles to guide these efforts to ensure the result is open, modular, portable, scalable, and secure by design.

In addition, we should also adapt important features already deployed in other types of systems today. We need to make it easy to reuse system configurations across hardware. It is also essential to ensure support for multicore systems and time isolation. Memory isolation can allow us to share resources securely across processes.

Ultimately, this standard must be put to the test by making a concerted effort to build a real-world proof of concept around it. This will almost certainly require significant funding – either of an R&D nature or from an end customer. Considering the seL4 ecosystem, the first step towards the definition of a standardized VMM would be the creation of an RFC for community discussion and approval. It will be up to one or more members of the seL4 community to look for opportunities to take up this mantle and be a champion for this initiative. Also, such kind of standard VMM will only be successful when discussed within the seL4 community. Thus, the spread of such idea through the seL4 community communication channels is essential. Moreover, the creation of work groups within the seL4 Community, around topics of interest, may be the best approach to leverage such standard VMM.

There has been incredible interest and innovation in edge computing. But the edge is more complex and more pervasive by the day. At the same time, edge computing devices are becoming a larger target because of their growing prevalence and potential damage.

We believe that securing edge computing needs to evolve to the next level. Leveraging a provably secure kernel like seL4 as a baseline will help us improve system security as a whole.

Bibliography

1 G. Heiser, “The seL4 Microkernel. An Introduction Whitepaper,” revision 1.2.

2 Services, Training and Products endorsed by the Foundation, 2022.

3 seL4 Foundation Membership, 2022.

4 Elphinstone, K.; Heiser, G. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In Proceedings of the Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles; Association for Computing Machinery: New York, NY, USA, 2013; SOSP ’13, p. 133–150.

5 seL4 Project. Frequently Asked Questions on seL4, 2022.

6 seL4 Project. seL4 Foundation, 2022.

7 Chiueh, S.N.T.c.; Brook, S. A survey on virtualization technologies. Rpe Report 2005, 142.

8 Popek, G.J.; Goldberg, R.P. Formal Requirements for Virtualizable Third Generation Architectures. Commun. ACM 1974, 17, 412–421.

9 Tiburski, R.T.; Moratelli, C.R.; Johann, S.F.; de Matos, E.; Hessel, F. A lightweight virtualization model to enable edge computing in deeply embedded systems. Software: Practice and Experience 2021, 51, 1964–1981.

10 Moratelli, C.R.; Tiburski, R.T.; de Matos, E.; Portal, G.; Johann, S.F.; Hessel, F. Chapter 9 - Privacy and security of Internet of Things devices. In Real-Time Data Analytics for Large Scale Sensor Data; Das, H.; Dey, N.; Emilia Balas, V., Eds.; Academic Press, 2020; Vol. 6, Advances in Ubiquitous Sensing Applications for Healthcare, pp. 183–214.

11 Martins, J.; Alves, J.; Cabral, J.; Tavares, A.; Pinto, S. μRTZVisor: A Secure and Safe Real-Time Hypervisor. Electronics 2017, 6.

12 Smith, J.; Nair, R. Virtual machines: versatile platforms for systems and processes; Elsevier, 2005.

13 Russell, R. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. SIGOPS Oper. Syst. Rev. 2008, 42, 95–103.

14 Vojnak, D.T.; Đorđević, B.S.; Timčenko, V.V.; Štrbac, S.M. Performance Comparison of the type-2 hypervisor VirtualBox and VMWare Workstation. In Proceedings of the 2019 27th Telecommunications Forum (TELFOR), 2019, pp. 1–4.

15 Azmandian, F.; Moffie, M.; Alshawabkeh, M.; Dy, J.; Aslam, J.; Kaeli, D. Virtual Machine Monitor-Based Lightweight Intrusion Detection. SIGOPS Oper. Syst. Rev. 2011, 45, 38–53.

16 Rosenblum, M.; Garfinkel, T. Virtual machine monitors: current technology and future trends. Computer 2005, 38, 39–47.

17 Tickoo, O.; Iyer, R.; Illikkal, R.; Newell, D. Modeling Virtual Machine Performance: Challenges and Approaches. SIGMETRICS Perform. Eval. Rev. 2010, 37, 55–60.

18 Xu, F.; Liu, F.; Jin, H.; Vasilakos, A.V. Managing Performance Overhead of Virtual Machines in Cloud Computing: A Survey, State of the Art, and Future Directions. Proceedings of the IEEE 2014, 102, 11–31.

19 Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. Xen and the Art of Virtualization. In Proceedings of the Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles; Association for Computing Machinery: New York, NY, USA, 2003; SOSP ’03, p. 164–177.

20 Chierici, A.; Veraldi, R. A quantitative comparison between xen and kvm. Journal of Physics: Conference Series 2010, 219, 042005.

21 Dall, C.; Nieh, J. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. SIGPLAN Not. 2014, 49, 333–348.

22 Raho, M.; Spyridakis, A.; Paolino, M.; Raho, D. KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing. In Proceedings of the 2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), 2015, pp. 1–8.

23 Mansouri, Y.; Babar, M.A. A review of edge computing: Features and resource virtualization. Journal of Parallel and Distributed Computing 2021, 150, 155–183.

24 Ramalho, F.; Neto, A. Virtualization at the network edge: A performance comparison. In Proceedings of the 2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016, pp. 1–6.

25 Hwang, J.Y.; Suh, S.B.; Heo, S.K.; Park, C.J.; Ryu, J.M.; Park, S.Y.; Kim, C.R. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In Proceedings of the 2008 5th IEEE Consumer Communications and Networking Conference, 2008, pp. 257–261.

26 Stabellini, S.; Campbell, I. Xen on arm cortex a15. Xen Summit North America 2012, 2012.

27 Martins, J.; Tavares, A.; Solieri, M.; Bertogna, M.; Pinto, S. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In Proceedings of the Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020); Bertogna, M.; Terraneo, F., Eds.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2020; Vol. 77, OpenAccess Series in Informatics (OASIcs), pp. 3:1–3:14.

28 Li, H.; Xu, X.; Ren, J.; Dong, Y. ACRN: A Big Little Hypervisor for IoT Development. In Proceedings of the Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments; Association for ComputingMachinery: New York, NY, USA, 2019; VEE 2019, p. 31–44.

29 Li, S.W.; Li, X.; Gu, R.; Nieh, J.; Hui, J.Z. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, 2021, pp. 3953–3970.

30 Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC 05); USENIX Association: Anaheim, CA, 2005.

31 Agache, A.; Brooker, M.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.M. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20); USENIX Association: Santa Clara, CA, 2020; pp. 419–434.

32 Tsirkin, M.S.; Huck, C. Virtual I/O Device (VIRTIO) Version 1.1. OASIS Committee 2018.

33 seL4 Project. Virtualisation on seL4, 2022.

34 seL4 Project. CAmkES VMM, 2022.

35 seL4 Project. The seL4 Core Platform, 2022.

36 Leslie, B.; Heiser, G. The seL4 Core Platform, 2022.

37 Leslie, B.; Heiser, G. Evolving seL4CP Into a Dynamic OS, 2022.

38 Li, X.; Li, X.; Dall, C.; Gu, R.; Nieh, J.; Sait, Y.; Stockwell, G. Design and Verification of the Arm Confidential Compute Architecture. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22); USENIX Association: Carlsbad, CA, 2022; pp. 465–484.

39 Vázquez-Ingelmo, A.; García-Holgado, A.; García-Peñalvo, F.J. C4 model in a Software Engineering subject to ease the comprehension of UML and the software. In Proceedings of the 2020 IEEE Global Engineering Education Conference (EDUCON), 2020,pp. 919–924.

40 seL4 Project. Supported Platforms, 2022.

41 Pinto, S.; Santos, N. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Comput. Surv. 2019, 51.

42 Kuz, I.; Liu, Y.; Gorton, I.; Heiser, G. CAmkES: A component model for secure microkernel-based embedded systems. Journal of Systems and Software 2007, 80, 687–699. Component-Based Software Engineering of Trustworthy Embedded Systems,

43 Open Synergy. Android Ecosystem, 2022.

44 Randal, A. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. ACM Comput. Surv. 2020, 53.

45 Zhang, X.; Zheng, X.; Wang, Z.; Li, Q.; Fu, J.; Zhang, Y.; Shen, Y. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In Proceedings of the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems; Association for Computing Machinery: New York, NY, USA, 2019; ASPLOS '19, p. 93–105.

Technology Innovation Institute LLC
P.O. Box 9639
Abu Dhabi, UAE

tii.ae