x86-64 VMM for seL4: Security Analysis and Requirements

Neutrality*(Geneva, Switzerland)

Trustworthy Systems, University of New South Wales†(Sydney, Australia)

Abstract

Communication security inherently depends on the security of the underlying operating system (OS), which itself can only be as good as the isolation provided by the underlying hypervisor – if that is compromised, so is the whole OS. Common hypervisors such as Xen or KVM are highly complex, their trusted computing base (TCB) consist of hundreds of thousands of lines of monolithic code. Serious vulnerabilities are regularly reported for Xen, for instance.

This project aims to significantly improve the security of a hypervisor-based OS by building it on top of seL4, leveraging the formal correctness proofs and much reduced TCB of seL4. In particular, seL4's virtualisation design does not share the virtualisation layer, the virtual-machine monitor (VMM), between VMs, meaning the VMM cannot break isolation even if it is compromised. In addition, critical components can run natively on seL4, further reducing their dependence on unverified code.

seL4 is an open-source, formally-verified microkernel that has matured and been maintained for over a decade. seL4's small size (10,000 Lines of Code) and formal verification make it an appealing Xen replacement, however, its virtualization support is currently limited. A first step to enabling a complete isolated OS on seL4 is to implement a hardened, open-source, x86 64-bit Virtual Machine Monitor (VMM) for the seL4 microkernel capable of hosting the core OS virtual machines.

*https://neutrality.ch/

†https://trustworthy.systems/

Contents

Abstract **1**

1 Scope **3**

2 Functional Specifications **3**
2.1 System Architecture 3
2.2 Paravirtualisation 3
2.3 Hardware-Assisted Virtualisation 4
2.4 Device Emulation 4

3 Security Analysis **4**
3.1 System Inventory 4
3.1.1 Components 4
3.1.2 Interactions 5
3.2 Security Assumptions 6
3.2.1 Hardware 6
3.2.2 seL4 Microkernel 6
3.2.3 Toolchain 7
3.2.4 System Components 7
3.2.5 VMM 8
3.2.6 Device Backends 8
3.2.7 Guest VM 8
3.3 Security Threats 9
3.3.1 Trust Domain Boundaries 9
3.3.2 Threat Enumeration 9

4 User Requirements **10**
4.1 Functional Requirements 10
4.1.1 Core Functions 10
4.1.2 Device Support 10
4.2 Non-functional Requirements 10
4.2.1 Performance 10
4.2.2 Security 11
4.2.3 Maintainability 11
4.2.4 Scalability 11
4.2.5 Flexibility 11
4.3 Implied Requirements 12
4.3.1 Manageability 12

5 Future Requirements **12**
5.1 Functional Requirements 12
5.1.1 Core Functions 12
5.1.2 Device Support 12
5.2 Non-functional Requirements 12
5.2.1 Performance 12
5.3 Implied Requirements 12
5.3.1 Manageability 12

Acknowledgements **12**

Bibliography **13**

A Appendix **13**
Glossary 13

1 Scope

This document is written as a part of the *Makatea* project, which consists in the development of an open-source x86-64 Virtual Machine Monitor (VMM) for the seL4 microkernel.

This document is intended as an input to the development and to the verification activities of the project. It establishes the high level functional specifications of the VMM, a high level security analysis, and a list of user requirements.

2 Functional Specifications

A Virtual Machine Monitor (VMM), sometimes called a Hypervisor, is a piece of software that creates the abstraction of a Virtual Machine (VM), a virtual computing environment inside of which a guest OS can run isolated from other components of the system.

Devices exposed to the guest OS such as serial ports, PCIe devices, real-time clocks or interrupt controllers are either emulated by the VMM, or pass-through hardware devices. Devices that are passed-through are directly managed by the guest, and in this case the VMM merely relays memory maps, I/O access as well as interrupt events between the guest and the hardware.

Type 1 VMMs are baremetal and run directly on top of the hardware while *Type 2* VMMs run on top of an OS, though in practice the distinction is not that clear cut. In seL4, VMMs are regular tasks that run under the seL4 microkernel, and each VMM manages a single VM. The VMMs receive the necessary seL4 capabilities that give access to hardware resources and enable communication with other tasks.

2.1 System Architecture

Similarly to the existing 32-bit x86 VMMs for seL4, the 64-bit x86 VMM runs as a regular user task under seL4 and access to the hardware and to other tasks is mediated by the seL4 microkernel. Both the seL4 microkernel and the VMM run in the *root* domain of the CPU. The guest VM runs in a *non-root* domain of the CPU, where the kernel runs in ring 0 and user processes in ring 3.

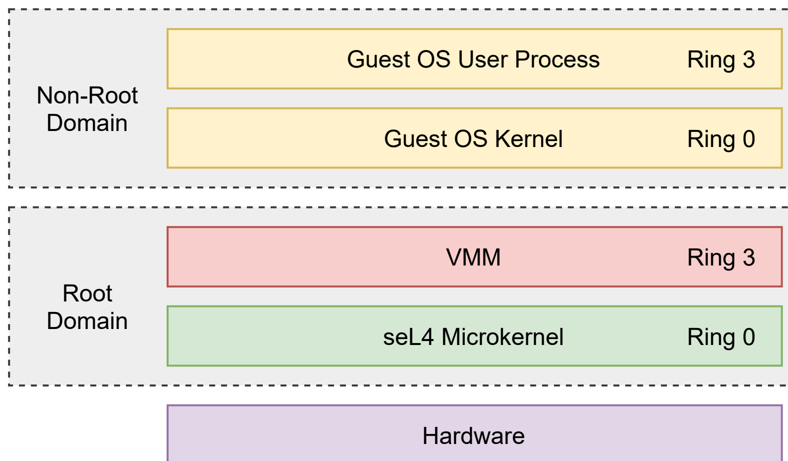

Figure 1: CPU domains and rings

2.2 Paravirtualisation

The 64-bit x86 VMM uses paravirtualisation, like its 32-bit counterpart. Guest paravirtualisation improves performance over the full virtualisation of an unmodified guest, and more importantly it enables a significant reduction of its code complexity.

The VMM implements the industry standard `paravirt-ops` (pv-ops) guest API. This API is implemented by Xen and KVM, amongst others, and is supported out of the box by a large number of OS such as Linux, OpenBSD and FreeBSD.

2.3 Hardware-Assisted Virtualisation

Modern x86-64 processors include hardware virtualisation extensions. These extensions are now ubiquitous, especially on processors designed for the server market. Intel and AMD together dominate the x86 market with processors that implement the same ISA but different yet functionally equivalent extensions. The VMM requires specific hardware virtualisation extensions from the processor.

On Intel processors, the VMM requires the following extensions:

- VT-x, also known as *Virtual Machine Extensions* (VMX).
- *Extended Page Tables* (EPT).
- *APIC virtualisation* (APICv).
- VT-d, also known as *Virtualisation Technology for Directed I/O* or simply IOMMU.

Eventually, support for AMD processors will be added.

2.4 Device Emulation

The VMM emulates a modern x86-64 system with a small set of standard devices such as an APIC, clocks and power management subsystems, and a PCIe bus. These core devices are fully implemented by the VMM. In addition, the VMM supports pass-through and emulated devices.

Pass-through devices are hardware devices directly exposed to the guest OS. Once set up they require little interaction by the VMM besides interrupt forwarding and PCI bus management. The required hardware IOMMU protects system memory from rogue DMA transfers and thus allows the drivers in the guest VM to be fully untrusted.

Initially, few emulated devices are supported. A virtual serial port provides access to the boot console of the guest OS and can be used for simple communication with the guest. A network-like mechanism enables high-performance communication between VMs in a point-to-point configuration.

Eventually, emulated devices implementing the VirtIO specifications will be added. Common device types include Console devices, Network devices, Block devices, and Socket devices.

The emulated devices implemented by the VMM are exposed as CAmkES components that can be composed into a full-fledged system like any other component. VMM devices are typically connected to other CAmkES components that act as backends. For instance a VirtIO Block Device could be connected to a component that uses Untyped memory to emulate a ramdisk, or it could be connected to a component that uses a disk partition. While the VirtIO devices are considered to be an integral part of the VMM, the various backends are not.

3 Security Analysis

3.1 System Inventory

The system components and their interactions identified in this section are represented in fig. 2 below.

3.1.1 Components

The following components related to an seL4 VMM have been identified:

- The **Hardware** is defined by all physical and logical components that provide the computing environment. It includes processors and their features, system memory, interrupt controllers, system firmware, and devices. In the context of this document a modern x86-64 system with PCIe buses is assumed, from either Intel or AMD.
- The **seL4 microkernel** runs directly on top of the hardware and controls all accesses to resources. It provides fine-grained access control through capabilities, and enables communication between components of the system. It is the most critical part of the software system, and is the only component that runs in privileged mode (root domain ring 0).
- The **VMM** itself runs as an seL4 task. It manages the guest computing environment and emulates exposed devices. Access to the hardware is controlled by capabilities on seL4 objects such as TCBs (Thread Control Block, not to be confused with Trusted Computing Base) to emulate execution contexts and VSpaces to

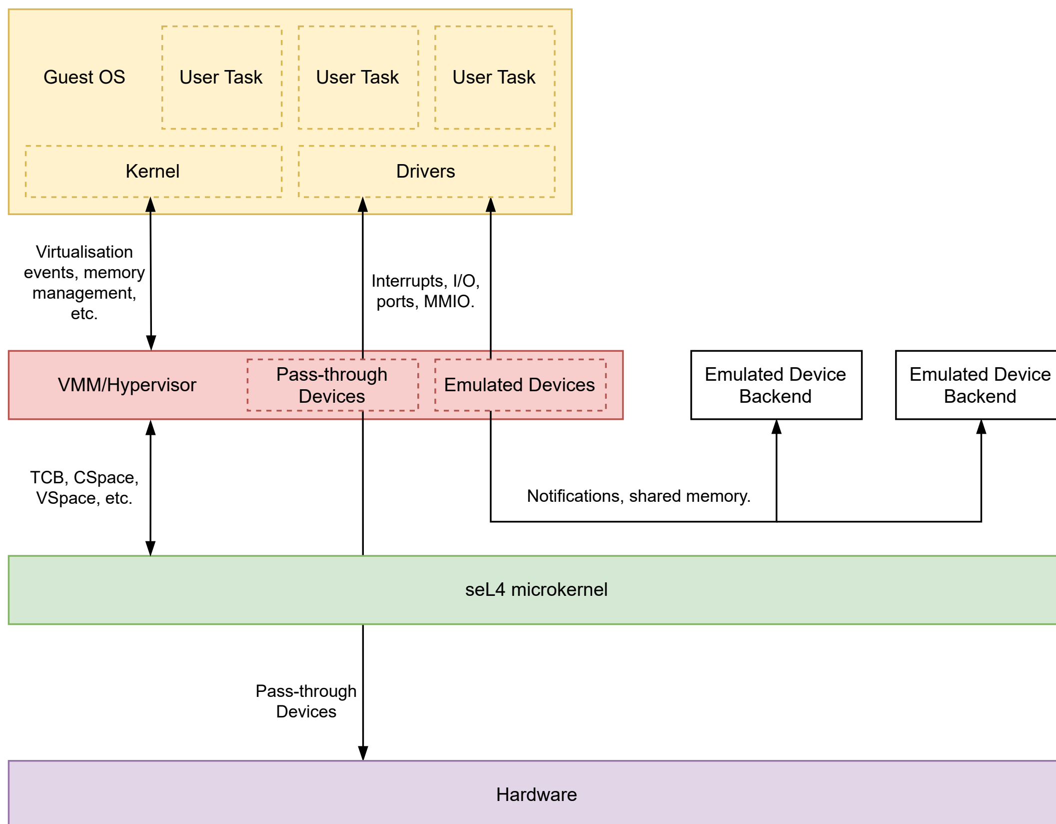

Figure 2: VMM Component Diagram

emulate the guest address space. The VMM exposes a number of devices to the guest which may be emulated or physical devices passed-through.

- The **Guest** OS runs in the VM and is isolated from all other components of the system. It is typically a generic OS such as Linux, supporting paravirtualisation hypercalls. It includes specific drivers for the devices exposed by the VMM, be they pass-through or emulated.
- A number of **seL4 tasks** distinct from the VMM provide services useful to the VMM. A typical example of such service is a serial console that can interface with a guest system console. Another example is a communication component that can interconnect multiple VMs, emulating a virtual Ethernet network between them. The backends of emulated devices are typically implemented as distinct seL4 tasks.

3.1.2 Interactions

The following principal interactions between the aforementioned components have been identified:

- The VMM interfaces with seL4 kernel objects to manage its own internal resources as well as those used to create the guest environment. Thread Control Block (TCB) objects represent execution contexts (threads) and are used to emulate CPUs. Page directories, Page tables, and Frames are used to manage the address spaces of both the VMM and the guest VM.
- The VMM interfaces with seL4 kernel objects to interact with physical devices to be exposed ("pass-through") to the guest. The devices are accessed via seL4 kernel objects that represent mappable register memory, notifications of interrupts via asynchronous IPC, and controlled access to device ports.
- The VMM communicates with other seL4 tasks using seL4 endpoint objects to implement the backend of emulated device drivers. The communication typically involves shared memory and synchronisation IPCs.
- The VMM implements a virtual CPU (vCPU) for the guest domain. While regular instructions execute within the guest domain, some sensitive instructions cause the guest domain to exit (VM exit) and control flow to

enter dedicated hooks in the VMM. The VMM hooks handle these events and eventually re-enter the domain VM.

- The virtual memory space of the guest OS is managed using CPU virtualisation extensions. The translation tables (EPT, etc.) that map guest virtual memory pages to VMM pages are handled by the VMM.
- Devices exposed to the guest OS use memory mapped register regions, I/O ports as well as interrupts. Passed-through hardware devices have their registers directly mapped into the guest address space and their access does not require actions from the VMM. Interrupts from hardware devices reach the VMM as seL4 events which the VMM forwards to the guest as virtual interrupts. Devices that are emulated by the VMM require the VMM to manipulate guest pages to emulate registers as well as virtual interrupts. Device implementation may be implemented by separate seL4 tasks communicating with the VMM using shared memory and seL4 events (IPC endpoints).

3.2 Security Assumptions

Two Trusted Computing Bases (TCBs), are identified:

- The **System TCB** includes all components that if compromised could potentially compromise the entire system. These components must be fully trusted to guarantee the security properties of any other component.
- The **VM TCB** includes all components that if compromised could potentially compromise the VMs that they relate to, but no other VM. These components must be fully trusted to guarantee the security properties of the related VMs.

3.2.1 Hardware

The complete hardware, including its various embedded firmware, must be fully trusted and is integral to the System TCB. This is a necessity since many hardware components can easily gain access to system memory. In particular, the following threat vectors are not considered in this analysis:

- Hardware tampering (e.g. supply chain attacks).
- Hardware vulnerabilities (e.g. Spectre, Row hammer).
- Covert hardware channels (e.g. timing attacks).
- Physical interference (e.g. jamming, probing, bit flips).

3.2.2 seL4 Microkernel

The seL4 microkernel itself, by virtue of running in privileged mode in the root domain, has direct access to all memory pages and as such it must be trusted and is part of the System TCB.

seL4 is however a high-assurance microkernel with a lean feature set that keeps the code complexity and size small, and thus less likely to have errors. In addition, its implementation is proven to be correct against its specification using formal methods. The formal proof guarantees the absence of errors such as:

- Undefined behaviour
- Memory leaks
- Buffer overflows
- Pointer errors
- Arithmetic errors

The proof also guarantees that the seL4 microkernel does correctly enforce the strong security properties of integrity and confidentiality. The extend of the formal proof however differs depending on the target platform as shown in table 1 below.

Table 1: seL4 verification claims per architecture as of 2021.

Architecture	Functional Correctness	Virtualisation	Security Properties	Binary Verification
x86 32-bit	N	N	N	N
x86 64-bit	Y	N	N	N
ARMv7	Y	N	Y	Y
ARMv7 (hyp.)	Y	Y	N	N (soon)
ARM 64-bit	N	N	N	N
RISC-V 64-bit	Y	N	Y	Y

On the x86-64 platform, the proof only guarantees the functional correctness of the C code against the specifications, as opposed to the binary code, and specifically does not cover:

- Configuration options that differs from the verified configuration
- Virtualisation extensions (VT-x on Intel, AMD-V on AMD)
- IOMMU (VT-d on Intel, AMD-Vi on AMD)
- Kernel start-up code
- Security properties

For these aspects, the security of the seL4 microkernel relies on conventional security practices.

3.2.3 Toolchain

The current formal verification of the seL4 microkernel on x86-64 architecture stops at the C code. Hence the toolchain must be fully trusted and is part of the System TCB. The toolchain includes the compiler (`gcc`), the assembler and linker (`binutils`), the C runtime and standard library (`musl`), and generally all tools that together produce the binary code, or have the capacity to inject code that runs in ring 0 in the root CPU domain. In particular, the following threat vectors are not considered in this analysis:

- Code errors in the toolchain causing functional changes between the binary code and the C source, such changes negatively affecting the security properties of the seL4 microkernel.
- Malware injection by the toolchain introducing backdoors or changes that negatively affect the security properties of the seL4 microkernel.

The toolchain is however open-source and consists mostly of general tools that have been ported to seL4 such as `gcc`, `binutils` and `musl`. The code changes introduced by the port are relatively small and may be audited.

3.2.4 System Components

seL4 being a lean microkernel, a number of system critical features are implemented as user tasks running in ring 3 rather than as components of the kernel running in ring 0 as it is commonly done on with monolithic kernels. While this approach allows for a code base running in ring 0 to be much smaller and more secure than a monolithic kernel, a number of user tasks inherit strong security responsibilities that if breached could lead to a full system compromise. The security properties of seL4 rely on the correct implementation of a capability system implemented by the microkernel and covered by the formal proof.

Upon boot, a root task is created and handed over all capabilities. This root task is then responsible for creating all other tasks and donating the capabilities that they require to fulfil their functions. Typically this root task is the `CapDL Loader` which loads the various ELF binaries and donates capabilities according to a static configuration file.

A complete seL4 system is typically composed of a large number of components whose interactions are governed by an even larger number of capabilities. Static systems are therefore typically built using the convenient `CAmkES` framework along with a library of standard components, greatly simplifying the system definition and capability allocation.

In addition, the seL4 ecosystem comes with a number of system libraries that implement fundamental functionalities such as address space management, memory allocation, and hardware abstractions. These libraries are foundational blocks that most seL4 tasks link against.

A number of these system components must be fully trusted and are part of the System TCB since their compromission could potentially lead to a full system compromise.

3.2.5 VMM

The VMM runs as a user task (ring 3) in the CPU root domain, and it is provided the capabilities that it requires to function. A misbehaving or compromised VMM cannot affect other components that it does not have the capabilities to. Two caveats are identified and mitigated:

- DMA transfers from physical PCI devices managed by the guest (pass-through) can evade the capability system enforced by the microkernel. A guest could trigger the device into writing to system memory and potentially compromise the entire system. This analysis assumes that an IOMMU is present in the system and that is is correctly configured by the microkernel to restrict DMA transfers to memory areas that the guest has the capabilities to write to. As such, the presence of an IOMMU is critical to the security of the system.
- The VMM can cause a denial of service by consuming a large amount of CPU time. Typically the CPU time allocated to all runnable tasks would follow priority based FIFO semantics. This threat is fully mitigated by using the Mixed Criticality Systems (MCS) extensions of the seL4 microkernel. These extensions effectively implement time partitioning via scheduling contexts that require their own set of capabilities. Time partitioning comes at a performance cost in that the system must be statically designed with Worst Case Execution Time (WCET) in mind and typically does not allow for slack time scheduling. As such the use of MCS is a trade-off that might not always be enabled. In the context of this analysis we assume that MCS is enabled and that denial of service attacks caused by a misbehaving or compromised VMM are mitigated.

The VMM has full direct access to all resources exposed to the guest VM and therefore it is an integral part of the VM TCB. Note that since one VMM only manages a single VM, all other VMs in the system would be unaffected by a compromised VMM instance. However since the other instances of the VMM share the same code base, they share potential vulnerabilities.

3.2.6 Device Backends

Device backends are components that implement the backend of emulated devices exposed to a guest VM. They are conceptually located outside of the VMM and are implemented as distinct seL4 tasks. As such they are constrained by the set of capabilities that they inherit throughout their life cycle and are unable to negatively affect any resource that they do not have the capability to affect.

Amongst the capabilities that they may require to implement their function, they receive the capabilities to interact with the VMM via shared memory and synchronisation messages. Assuming a valid System TCB and a correctly implemented VMM, these capabilities do not allow the device backends to negatively affect the VMM nor the VM beyond affecting the functionality of the devices that they implement.

Device backends have the natural ability to read data exposed by the guest and to tamper with the data provided to the guest. A compromised device backend does breach these security properties, but can not escalate to a VM compromission unless:

- The system design has provided the backend driver with capabilities that would allow it to affect the VM and or the system (i.e. design error).
- The VMM is vulnerable to attacks that can be crafted by the backend driver (i.e. code error).

Therefore under the assumptions that the two vectors mentioned above are mitigated – that is, the backends do not receive capabilities to critical VM resources, and the VMM correctly implements the device backend interface – the device backends can be considered untrusted and are not part of the System TCB nor of the VM TCB.

3.2.7 Guest VM

In the context of the VMM, the guest VM guest must be considered untrusted, it can not be part of any TCB. This is a design requirement of the VMM.

Note that in an integrated system similar to QubesOS, one (or more) guest VMs may be given the capabilities to interact with other VMs, for instance to start and stop them, or to configure some aspects of their VMM. The level of trust given to these administrative VMs depends on the capabilities that they receive, which may assign them to the System TCB. In the context of this document the guest VMs are considered untrusted.

3.3 Security Threats

3.3.1 Trust Domain Boundaries

Security threats originate mainly from untrusted components. As elaborated above, the list of untrusted components is identified as:

- The backend drivers of emulated devices.
- The guest VMs.
- Unrelated seL4 tasks, see below.

The formal proof of the seL4 microkernel guarantees the correct functionalities of its capability mechanism. As such, and with the assumption of a non compromised System TCB, unrelated seL4 tasks can not affect the security properties of components that they do not have capabilities to interact with. These unrelated seL4 tasks are therefore not considered further.

The following trust domain boundaries involving the VMM are identified:

- VMM $\Longleftrightarrow$ Guest VM
- VMM $\Longleftrightarrow$ device backends
- Guest VM $\Longleftrightarrow$ pass-trough devices

3.3.2 Threat Enumeration

3.3.2.1 Security assumptions of the guest VM The guest VM is typically a conventional OS that includes a kernel that runs in supervisor mode (ring 0) and a number of user tasks that run in user mode (ring 3). The guest OS expects to be running in a VM that implement security features such as supervisor mode protection and memory management. The VMM shall not weaken the security of the guest OS by failing to satisfy its security assumptions.

Threat #1: The VMM creates an environment that allows an unprivileged guest task (ring 3) to gain access to data in another unprivileged guest task (ring 3) or to data in guest kernel memory (ring 0).

- **Impact**: Potential full guest VM compromise. The rest of the system would however remain unaffected, the compromission is contained in the VM.
- **Likelihood**: Low; the VMM uses VT-x/EPT extensions that greatly simplify the virtualisation of the MMU and thus reduce the risk of code errors.
- **Severity**: Low; considering guest VMs are untrusted as a whole.

3.3.2.2 Device backends The implementation of emulated VirtIO devices requires a certain communication between the VMM and a distinct seL4 task that acts as the emulated device backend. This communication typically involves shared memory pages between the VMM and the seL4 task implementing the device backend, as well as synchronisation messages. The API between the VMM and the emulated device backends lives at the trust boundary between mutually untrusted peers and as such implementation errors can have security implications.

Threat #2: The VMM shares an incorrect memory page with the backend of an emulated device, allowing the backend to read and/or write to memory areas in the VMM – including the guest VM memory space – that is is not authorised to access.

- **Impact**: Potential full VMM and guest VM compromise. The rest of the system would however remain unaffected, the compromission is contained in the VMM and/or guest VM.
- **Likelihood**: Medium; while the API between the VMM and emulated device backends backends need not be complex, implementation errors are often found at the interface between components. Furthermore zero-copy implementations where the same page is passed across multiple address spaces from the producer to the consumer (backend $\rightarrow$ VMM $\rightarrow$ guest driver $\rightarrow$ guest process) do raise the complexity, and thus the likelihood of code errors.
- **Severity**: Medium.

3.3.2.3 Guest VM isolation

The VMM creates the virtual environment, the non-root domain, in which the guest OS executes. A number of instructions, exceptions and traps do trigger VM exit events. These events are handled by the VMM which eventually returns into the guest VM.

Threat #3: The guest manages to exit the VM isolation and to execute code in the VMM.

- **Impact**: VMM compromise, denial of service, access to attack surface of other components. The rest of the system would however remain unaffected, the compromission and/or denial of service is contained in the VMM. Guest privilege escalation to resources the VMM has capabilities on.
- **Likelihood**: Medium; the interface between the VMM and the guest VM is complex and therefore prone to bugs. However such bugs are deemed rare, and hardware extensions such as VT-x, EPT and IOMMU limit both the required complexity of the implementation and the potential impact.
- **Severity**: Medium.

4 User Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

4.1 Functional Requirements

4.1.1 Core Functions

- **REQ-001**: The VMM SHALL create a virtual environment emulating an x86-64 architecture on a x86-64 host.
- **REQ-002**: Each VMM instance SHALL host a single guest virtual machine.
- **REQ-003**: The VMM SHALL emulate a functional subset of a modern x86-64 virtual machine with ACPI, IO APIC(s), and at least one PCIe bus. The functional subset emulated by the VMM SHALL be sufficient to execute an unmodified guest OS such as Linux.
- **REQ-004**: The VMM SHALL support modern x86-64 Intel processors with the following virtualisation extensions: VT-x, EPT, APICv, VT-d.

4.1.2 Device Support

- **REQ-051**: The VMM SHALL allow configuring pass-through PCIe devices.
- **REQ-056**: The VMM SHALL implement a VirtIO Socket device.
- **REQ-057**: The VMM SHALL implement an emulated/virtual network device.
- **REQ-058**: The emulated network device SHALL allow point-to-point communication between two VMs.
- **REQ-059**: The VMM SHALL implement an emulated/virtual serial device.
- **REQ-060**: The emulated serial device SHALL allow point-to-point communication between two VMs.
- **REQ-061**: The emulated serial device SHALL allow multiplexing multiple ports from multiple VMs onto the same physical serial port. Special hot keys select which VM input goes to; output can be colour coded, or each line prefixed with the name of its VM.
- **REQ-062**: The emulated network interface exposed by the VMM to the guest SHOULD be supported by network drivers present in recent mainline Linux kernel. Should the VMM expose a network interface not supported by the Linux kernel, a high performance Linux driver for the interface SHALL be provided.

4.2 Non-functional Requirements

4.2.1 Performance

- **REQ-101**: The VMM SHALL make use of the following virtualisation extensions of the underlying hardware: VT-x, EPT, APICv, VT-d.

- **REQ-104**: The emulated network interface SHALL minimise the number of VM exits required per packet transmission. It SHOULD limit the number of VM exits to one per group of packets: a single memory mapped register access SHOULD trigger the transmission of all packets present in the packet descriptor ring.
- **REQ-105**: The emulated network interface SHALL minimise the number of interrupts. It SHOULD limit the number of interrupts to one per group of packets either sent or received: a single interrupt SHOULD be used to inform the guest that all packets scheduled for transmission have been sent and/or that new packets have been received.
- **REQ-106**: The emulated network interface SHALL minimise the number of VM exits required per packet reception. The interrupt handler SHOULD not be required to write to more than one memory mapped register to acknowledge the received packets.
- **REQ-107**: The emulated network interface SHALL present a scatter-gather DMA interface to the guest such that the guest need not present packets in contiguous guest-physical memory.
- **REQ-108**: The emulated network interface SHALL support checksum offloading and TCP segmentation offload (TSO).
- **REQ-109**: The VMM SHALL perform memory copies between locations that have the same 64-bit word alignment and the same cache alignment whenever it would improve performance.
- **REQ-110**: The VMM SHALL use SSE4.2 instructions (such as `load_si128` and `store_si128`) for memory copies whenever it would improve performance.

4.2.2 Security

- **REQ-201**: The VMM SHALL **not** attempt to support processors lacking the virtualisation extensions VT-x, EPT, APICv, VT-d.

4.2.3 Maintainability

- **REQ-401**: The VMM SHALL be developed in the C language.
- **REQ-402**: The VMM source code SHALL follow a consistent coding style and coding standard.
- **REQ-403**: The VMM source code SHALL include documentation of all global variables and functions, including function arguments, preferably in Doxygen format.
- **REQ-404**: If changes to the guest OS are required by the VMM, for instance to support certain hypercalls, such changes SHALL be adequately documented and included in the upstream kernel project (e.g. Linux).

4.2.4 Scalability

- **REQ-501**: There SHALL be no limit on the number of VMMs that can run concurrently, within the constraints of hardware resources.
- **REQ-502**: There SHALL be no limit on the number of pass-through devices that can be attached to a VMM, within the constraints of hardware resources.
- **REQ-503**: There SHALL be no limit on the number of emulated devices that can be attached to a VMM, within the constraints of hardware resources.
- **REQ-504**: The VMM SHALL support multi-core hosts that are supported by the seL4 microkernel.

4.2.5 Flexibility

- **REQ-701**: The VMM SHALL provide full CAmkES integration such that VM instances can be entirely configured using CAmkES, including PCIe pass-through devices and emulated devices.
- **REQ-702**: The VMM SHALL support running in a KVM/QEMU VM with nested virtualisation enabled to facilitate development and testing.

4.3 Implied Requirements

4.3.1 Manageability

- **REQ-M01**: The VMM SHALL provide a straightforward way to specify at build time which devices are passed through to which VM.
- **REQ-M02**: The VMM SHALL provide interfaces for starting, stopping, and restarting virtual machines.

5 Future Requirements

The requirements in this section cover potential future works.

5.1 Functional Requirements

5.1.1 Core Functions

- **REQ-005**: The VMM SHALL support modern x86-64 AMD processors with the following virtualisation extensions: AMD-V, EPT, AVIC, AMD-Vi.
- **REQ-006**: The VMM SHALL support the emulation of multiple vCPU cores in the guest virtual machine.
- **REQ-007**: The VMM SHALL implement a functional subset of the industry standard `paravirt-ops` (a.k.a. `pv-ops`) guest API to run paravirtualised guest OS. The functional subset emulated by the VMM SHALL be sufficient to execute an unmodified paravirtualised guest OS such as Linux.

5.1.2 Device Support

- **REQ-052**: The VMM SHALL allow configuring emulated VirtIO devices.
- **REQ-053**: The VMM SHALL implement a VirtIO Console/Serial device.
- **REQ-054**: The VMM SHALL implement a VirtIO Network device.
- **REQ-055**: The VMM SHALL implement a VirtIO Block device.

5.2 Non-functional Requirements

5.2.1 Performance

- **REQ-102**: The VirtIO Network and Block devices SHALL support zero-copy operation between the backend driver and the guest OS for devices that are not shared between VMs.

5.3 Implied Requirements

5.3.1 Manageability

- **REQ-MIO1**: Where PCIe devices have multiple functions (for example, network cards that provide virtual endpoints), the VMM SHALL enable individual functions to be passed to different VMs, with shared drivers for shared parts.
- **REQ-MIO3**: The VMM SHALL provide a way to access EFI runtime services, either passed through or emulated.

Acknowledgements

This effort is co-funded by a grant from NLnet Foundation[1]. Neutrality's time is co-funded by the Innosuisse – Swiss Innovation Agency and the European Union, under the Eurostars[2] programme as Project E!115764.

[1] https://nlnet.nl/

[2] https://ec.europa.eu/programmes/horizon2020/en/h2020-section/eurostars-programme/

Bibliography

Gernot Heiser, Kevin Elphinstone. 2016. "L4 Microkernels: The Lessons from 20 Years of Research and Deployment." *ACM Transactions on Computer Systems*.

Gerwin Klein, Gernot Heiser, Kevin Elphinstone. 2009. "SeL4: Formal Verification of an Os Kernel." In *ACM Symposium on Operating Systems Principles*, 207–20. ACM.

Jesse Millwood, Leonard Elliott, Robert VanVossen. 2020. "Performance Impacts from the seL4 Hypervisor." In *2020 Ndia Ground Vehicle Systems Engineering and Technology Symposium*.

A Appendix

Glossary

- **ACPI** – Advanced Configuration and Power Interface.
- **APIC** – Advanced Programmable Interrupt Controller.
- **CAmkES** – Component Architecture for microkernel-based Embedded Systems.
- **CSpace** – seL4 kernel object used to manage capabilities.
- **DMA** – Direct Memory Access.
- **ELF** – Executable and Linkable Format.
- **EPT** – Extended Page Tables, MMU extension to improve virtualisation.
- **IOMMU** – I/O Memory Management Unit.
- **IPC** – Inter-Process Communication.
- **ISA** – Instruction Set Architecture.
- **KVM** – Kernel-based Virtual Machine, a common Linux hypervisor.
- **MCS** – Mixed Criticality Systems, seL4 time and space partitioning extensions.
- **MMU** – Memory Management Unit, CPU technology.
- **OS** – Operating System.
- **PCIe** – Peripheral Component Interconnect (PCI) Express, the standard x86-64 expansion bus.
- **TCB** – Thread Control Block, seL4 kernel object used to manage execution threads.
- **TCB** – Trusted Computing Base.
- **VM** – Virtual Machine, the emulation of a computer system.
- **VMM** – Virtual Machine Monitor, also called Hypervisor.
- **VMX** – Virtual Machine Extensions, Intel CPU extension, also called VT-x.
- **VSpace** – seL4 kernel object used to manage address spaces.
- **VirtIO** – I/O virtualisation framework and paravirtualised device model.