UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Trianqles: Porting Qubes to seL4

by

Adam Felizzi

Thesis submitted as a requirement for the degree of

Comp Sci & Eng (Honours)

Submitted: November 2017 Student ID: z3463674

Supervisor: Ihor Kuz Topic ID: 3289

Abstract

Qubes is a security-oriented desktop operating system architecture, aiming to provide security through compartmentalization and isolation. This approach focuses on separating applications and operating system services across separate virtual machines (VMs), preventing exploited applications from compromising the entire operating system.

An integral part of Qubes system architecture is its use of the Xen hypervisor to provide all of its underlying virtualization. Qubes has currently reported 27 CVE's (Common Vulnerabilities and Exposures) relating to Xen in 2017 alone [Pro17a], and multiple more since its inception in 2010, a substantial portion of which directly compromises the security of Qubes users. Thus Qubes security guarantees are dependent on a relatively heavyweight and unsecure hypervisor to enforce its isolation. Porting Qubes to a more secure architecture becomes increasingly justifiable, motivating a project to move Qubes to seL4, a formally verified microkernel to provide stronger guarantees for achieving isolation and security.

This thesis aims to start laying the necessary implementation framework for achieving a port of Qubes on seL4. This is demonstrated by porting a viable subset of the Qubes architecture to run on seL4.

Acknowledgements

I would like to thank my supervisor Ihor Kuz for giving me valuable guidance and feedback throughout the past year. It was a privilege to receive Ihors mentorship throughout the thesis as he helped me keep on track. I would like to thank Kent and Anna for sharing their technical knowledge and giving valuable advice when it came to dealing and managing thesis studies. Thank you to my assessor Gernot for taking the time to review my work and for giving me valuable feedback from thesis part A. I would lastly like to generally thank the people at Data61 for welcoming me, giving me the resources and opportunities to learn and grow throughout the course of my thesis, it was a privilege.

Abbreviations

OS Operating System

IPC Inter-process Communication

VM Virtual Machine

VMM Virtual Machine Monitor

PV Paravirtualisation

HVM Hardware Virtual Machine

API Application Programming Interface

TCB Trusted Computing Base

GUI Graphical User Interface

Contents

Chapter 1: Introduction

The primary goal of this thesis is to evaluate the viability of seL4 as a virtualisation platform for Qubes. Qubes is a security-focused desktop operating system (OS) architecture which leverages virtualisation as a means to isolate monolithic operating system services. The thesis seeks to enable Qubes on seL4 through utilising existing seL4 virtualisation services, and in doing so, we:

- Assess the viability of seL4 as a virtualisation platform for supporting the Qubes framework
- Propose solutions that would enable Qubes to run on seL4
- Identify the challenges, requirements and road-blocks to support a full Qubes port to seL4.

Qubes is an open-source desktop operating system architecture developed by Invisible Things Labs, designed with a focus on security. Key to Qubes approach towards security is to minimise the trusted computing base (TCB) of traditional monolithic desktop operating systems by moving untrusted system services (e.g. device drivers and applications) into isolated domains. To achieve this Qubes takes advantage of isolation mechanisms provided through modern virtualisation systems (e.g. hypervisors) and hardware technologies, such as Intel VT-x, VT-d and Intel Trusted Execution Technology (TXT). Through virtualisation, Qubes presents a compartmentalised OS, where dedicated virtual machines (VMs) are used to run untrusted applications, sandboxing their potential bugs and vulnerabilities that could otherwise compromise the entire system in a traditional monolithic OS. Currently underpinning Qubes' virtualisation is the Xen hypervisor.

Critical to enforcing Qubes' isolation is the virtualisation service it runs on, making it one of the most security critical elements in the system. Exploiting a bug in the hypervisor could potentially result in an attacker compromising the entire system. Further motivating this problem is that such vulnerabilities have unfortunately been found in Xen. Qubes has currently reported 27 CVE's (Common Vulnerabilities and Exposures) relating to Xen in 2017 alone [Pro17a], and multiple more since its inception in 2010. A large majority of these vulnerabilities break the fundamental isolation mechanisms Qubes relies on which significantly impacts the security of a Qubes user.

The security concerns introduced by Xen motivates the goals of this thesis. By porting Qubes to seL4, we ask:

- Would seL4 provide stronger security guarantees?
- What engineering would be required to enable Qubes on seL4?

seL4 is a high performance microkernel which is noted for its comprehensive formal verification of implementation correctness [KEH+09] [KAE+14]. As a microkernel, seL4 provides a small core subset of machine abstracts for controlling access to physical and virtual address spaces, threads, scheduling, page tables and interprocess communication (IPC). Thus seL4's microkernel architecture provides the necessary minimal base to build a virtual machine monitor (VMM) solution, which in combination with its formal verification provides for a convincing alternative to Xen when investigating security benefits.

Qubes however relies on a comprehensive set of virtualisation toolstacks and services supported by Xen. This presents the challenge of engineering. Through this thesis I aim to deconstruct Qubes, identify the

various services and toolstacks Qubes relies on and look to implement them in an seL4-based environment.

The result of this thesis was not the full port of Qubes to seL4, rather the focus throughout the thesis was on porting a minimal viable subset of the Qubes architecture, to serve as both a stepping stone and proof-of-concept for a full Qubes port in the future. The final product, which will be discussed in proceeding sections, is completed in a static environment which is not representative of a complete dynamic Qubes system. Contributions through this thesis focus on:

- Identifying and understanding the Qubes code base, the inner workings of its various services and its dependencies.
- An implementation of a core set of services that facilitate inter-VM communication mechanisms and virtualization API's, such that it can support the Qubes code base.

Though the implementation is within a static environment we will explore designs that allow the discussed system to be easily migrated to a more dynamic solution in the future.

This thesis presents a discussion of relevant background information in Chapter 2. Chapter 3 will expand on background information relating to Qubes. We will investigate existing approaches and related works in Chapter 4. Chapter 5 introduces my approach to porting Qubes and Chapter 6 subsequently analyses the specific implementation details behind my designs. Chapter 7 evaluates my implementation and future work based from the implementation will be outlined in Chapter 8. Lastly Chapter 9 concludes my thesis.

Chapter 2: Background

To understand the background of Qubes and its underlying motivations, we will first identify what a monolithic operating system (OS) is, unpack key design elements and fundamental security issues that suggest an alternative OS design. The primary concepts of virtualization and the state of hypervisor solutions with regards to both Xen and seL4 are addressed.

2.1 Monolithic Operating Systems

At a high level, an OS plays two distinct roles [TB14]. The first of which being the OS acts as an extended machine. In doing so, the OS provides a clean representation of system resources of which programmers can utilize to develop applications. These abstractions allow applications to become hardware agnostic, enabling them to run in a diverse set of environments, irrespective of the underlying hardware resources. The second role of the OS is that it acts as a resource manager, organizing the underlying hardware to provide a controlled and shared interface to the various programs utilizing them.

In this shared environment it becomes necessary to protect applications from potentially malicious or incorrect behaviours of other applications running on the same system. In the case of incorrect or malicious behaviour it becomes critical for the OS to enforce system wide policies, which introduces a variety of unique OS designs such as Qubes.

2.1.1 Monolithic Operating System Design

A monolithic kernel is an OS architecture where a large majority of the system components reside in the kernel, executing in a privileged mode. This includes device drivers, file systems, IPC systems, schedulers and memory management. In a traditional monolithic kernel design, this composition allows privileged components to call any other part of the system, which without restriction can often make understanding the system complicated and cumbersome. Structure is however defined between the privileged kernel level and a unprivileged user level from which applications are run. Additionally applications are able to interact with the abstractions provided by the OS through an extensive system call interface. Performing a system call will usually initiate a trap instruction that results in transferring control to the underlying OS to handle the privileged operation.

Monolithic kernel designs are seen to be used in many modern OSes including Unix, Linux, BSD and Windows.

2.1.2 Security vulnerabilities

> *"Can we rely on a big, fat, and buggy kernel that has hundreds of drivers inside, networking stacks, and so forth to enforce strong isolation?... People who regularly release kernel exploits for popular OSes (Linux being no exception) seem to be yelling: NO!"*
>
> — Joanna Rutkowska, *Founder of Qubes, Interview, Qubes OS: An Operating System Designed For Security, 2011 [Dan11]*

Today monolithic OSes, such as Windows or Linux, are being deployed on an increasing number of different hardware platforms with varying configurations. To accommodate for such support, the number of services and components within the kernel also increases. This directly impacts the size and complexity of the OS code base. In 2017, the Linux 4.12 kernel has 20+ million lines of code, averaging 795 line additions per hour during its 63 day development [KH17]. Hence a monolithic OS consists of a large trusted computing base resulting in a complex attack service. The growth and size of such a code base introduces multiple security concerns.

An increasing number of new components and drivers introduces a potential set of new bugs and faults. In a monolithic system, faults in a privileged mode can often make it difficult for the kernel to recover, often compromising the entire system. This is increasingly problematic however as that there is little effective isolation between the various drivers and components within the kernel. An exploit of a kernel bug can in effect compromise the security of other components in the system and the applications using them. If a bug is exploited through a user application or core system service (e.g. Ethernet driver), the OS is unable to guarantee protection over other applications and user data from being compromised.

Finally as the monolithic kernel code base grows so does its complexity, which makes it increasingly difficult to verify and reason that the kernel is fault-free. Traditional approaches to dealing with found bugs and exploits are reactive whereby developers release security patches on a bug-by-bug basis. This particularly doesn't scale well as it does not assure protection against future vulnerabilities.

2.2 Virtualisation

Virtualisation refers to the method of running software, such as an OS, within a virtual environment that emulates native hardware. This technology allows a single computer to host multiple virtual machines each of which host different operating systems. Advantageous to this approach is that a failure of a single virtual machine won't impact other virtual machines running on the same computer. This introduces a fault tolerance model that is both cost effective and easy to maintain [TB14].

Virtualisation generally works by establishing a hypervisor (or virtual machine monitor) that runs the guest OS in user mode. The guest OS faults whenever it tries to execute a privileged instruction, allowing the hypervisor to emulate the sensitive instruction. There are two general approaches to implementing hypervisors, being Type 1 and Type 2 hypervisors.

2.2.1 Type 1 Hypervisor

A type 1 hypervisor runs on bare metal. In a sense, a type 1 hypervisor becomes the host OS being the only program that runs in kernel mode. Guest virtual machines run on top of the hypervisor as a user process in user mode. Therefore when the guest OS executes a sensitive instruction that traps into the

hypervisor, the hypervisor inspects the instruction and emulates it on behalf of the guest. In placing the hypervisor in kernel mode it becomes the most privileged software in the system. In comparison to a monolithic kernel, the complexity and code size of a hypervisor is an order of magnitude smaller than a full operating system, thus having a reduced attack surface and potentially less bugs.

2.2.2 Type 2 Hypervisor

In contrast to the privileged operation of a type 1 host, a type 2 hypervisor runs as a user program within a host OS. A type 2 hypervisor is able to host a virtual machine, interpreting the guest OS machine instruction set. Popular type 2 hypervisors such as VMware & Virtualbox run as ordinary user programs on top of a host operating system such as Windows and Linux. When starting guest operating system, these hypervisors often use a form of binary translation, executing the guest OS binary but translating sensitive instructions to call internal hypervisor procedures that emulate the sensitive instruction.

2.3 Paravirtualisation

The discussed methods of trapping and emulating sensitive instructions and performing binary translations in a fully virtualised environment can however have negative impacts on the performance of the virtual machines. Support for full virtualisation wasn't originally part of the x86 architecture, where executing certain instructions with insufficient privileges would silently fail without ever causing a trap. Solutions to this problem often involved dramatic changes that would introduce performance inefficiencies. Examples including hypervisor solutions [BDF+03] dramatically rewriting portions of the binary to emulate non-trapping instructions and all updates to system structures (page tables). Processing and emulating these sensitive instructions introduces extra cycles, significantly impacting the performance of the guest OS if it frequently accesses these instructions with the assumption that it is the most privileged process.

To deal with these problems, paravirtualisation techniques were developed. In a paravirtualised system, the guest OS is aware that it is being run on a hypervisor and explicitly makes calls to the host OS/VMM to perform privileged operations. The VMM often defines a custom interface with which the guest OS is able to access. This interface can be designed around to reduce the total number of entries and exits performed between the VMM and guest OS.

2.4 Virtualisation for Security

Virtualisation provides a possible solution to isolating the processes found in a monolithic operating system, enabling multiple operating systems to be instantiated and run on top of a hypervisor. This functionality is leveraged by Qubes to create isolation containers, where applications and system services are executed within separate user defined virtual machines (VMs). Advantageous to this approach is a simplified interface that exists between a VM and a hypervisor where the number of services and interfaces provided by a hypervisor are orders of magnitude smaller than a traditional OS (e.g. hypervisors don't provide services such as file-systems and networking). This effectively reduces the overall trusted computing base, providing a smaller attack surface and enforcing stronger isolation between processes. Further motivating this approach is the maturity of hardware support for

virtualisation on modern desktop computers such as Intel VT-d, providing effective I/O sandboxing mechanisms for hardware, e.g. isolating a networking card code prone to compromise.

2.5 Xen

The Xen hypervisor is an open source type 1 (bare metal) hypervisor that makes it possible to run multiple operating systems in parallel on a single machine. Xen provides support for two different types of virtualisation, being Paravirtualisation (PV) and Hardware-assisted Virtualisation (HVM), both of which can be run at the same time on the single hypervisor. Xen PV enabled kernels exist for Linux, NetBSD, FreeBSD and OpenSolaris [Pro17d], being configured with PV drivers.

Xen also supports paravirtualisation in a HVM guest to by-pass disk and network I/O emulation. The Xen project also introduced a new virtualisation mode called PVH in version 4.5, which involves a kernel using PV drivers for boot and I/O, whilst using HW virtualisation extensions for the remaining parts of the system.

The standard Xen configuration involves a control domain VM, known as dom0. Dom0, typically configured as a Linux guest (though support for NetBSD and OpenSolaris exist), is the first VM to boot in a Xen system. Dom0 typically contains special privileges to access hardware directly, handles system I/O and is able to interact with other guest VMs (domU's). In addition dom0 contains a control stack that enables it to start, stop, destroy and configure other guest VMs on the hypervisor.

The Xen hypervisor provides all the virtualization facilities for Qubes.

2.5.1 Xen Paravirtualisation

Paravirtualisation in Xen is used to support [Pro17c]:

- Disk and Network drivers
- Interrupts and timers
- Emulated Motherboard and Legacy Boot
- Privileged instructions and Page Tables

In the Linux kernel, paravirtualisation support is added through a combination of PV frontend and backend I/O drivers and paravirt operation extensions (PVOPS). PVOPS are a specific piece of Linux kernel infrastructure that allows the kernel to be optimised for low level operations within the virtualisation stack. PV frontend and backend I/O drivers act in a client-server fashion whereby a VM with passthrough hardware access will host a backend driver that communicates to another guest VM hosting a frontend driver. The frontend driver presents a virtual interface within the guest OS for I/O devices such as block and network interfaces. This is also known as a split-driver configuration. The split driver model is illustrated in Figure 2.1, involving a guest OS interacting with various other domains that have passthrough access to hardware.

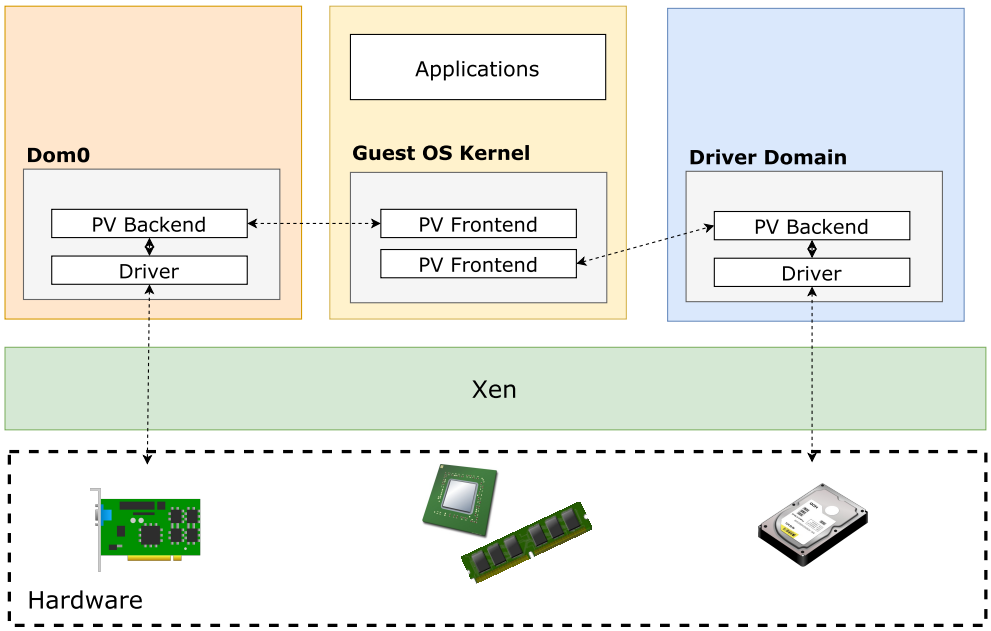

Figure 2.1: Overview of the Xen Paravirtualisation

2.6 seL4

2.6.1 Microkernel

In contrast to monolithic operating systems, microkernels look to reduce the amount of functionality in the kernel. This involves reducing the kernel to a minimal core subset of functionality and machine abstractions needed to implement an OS. A reduced code base that operates in kernel mode makes for a strong case as its reduces the potential attack surface and the opportunity for bugs.

Additional OS functionality, such as file systems and device drivers are then implemented as user processes running on top of the microkernel. This adds a layer of fault tolerance such that when a driver crashes it significantly reduces the chance that the entire system will crash. When a driver faults, the fault occurs at the user level which the kernel can then capture and deal with appropriately without impacting the other processes running on the same system.

Message passing communication between processes is then achieved through an Inter-process Communication mechanism (IPC). In addition, the microkernel implements a core subset of functionality such as scheduling, memory management and access control.

seL4 is a high performance microkernel, noted for its comprehensive formal verification of implementation correctness. The implementation of seL4's formal specification means that if initialised and used properly, according to the proof and hardware assumptions, the microkernel enforces the strong security properties: integrity, confidentiality, and availability [KAE+14]. There are also further proofs that extends seL4's formal specification and functional correctness to the binary [KAE+14].

As a microkernel, seL4 provides a minimal set of machine abstractions for controlling access to physical and virtual address spaces, threads, scheduling, page tables and a mechanism for inter process

communication (IPC).

2.6.2 Capabilities

User land access to kernel services and objects on seL4 is mediated through a capability-based access control system. In seL4, every kernel managed object has a capability associated with it that in turn specifies the rights of the user application when using the object. Each process stores its capabilities in its own capability space (cspace). When a process performs an action on a given object it will pass its capability which seL4 will verify before performing the action. Through its capabilities, seL4 provides a strong security model that prevents resources from being accessed without the necessary capability.

Some of the objects and resources you can create on seL4 include threads, page directory objects, notification endpoints, synchronous endpoints, memory (RAM) frames and device memory.

2.6.3 CAmkES

CAmkES, a Component Architecture for microkernel-based Embedded Systems is a software development and runtime framework that enables for the rapid and reliable development of static multi-server operating systems based on the seL4 microkernel. A CAmkES system typically consists of a series of software components, interconnected though statically defined connections. The CAmkES framework provides a language to describe components and connections in the system, a toolstack to automatically generate necessary glue code for a complete, bootable system image and integration with the seL4 environment and build system.

It is important to note that CAmkES is used to define static multi-server systems whilst as we expand on Qubes in later sections, we will find that Qubes is a dynamic system. In the context of this thesis, CAmkES has mature and up-to-date VMM support whilst also providing the convenience of rapidly prototyping a Qubes-like system. This allows the focus of the thesis to move towards Qubes specific systems, however the need for a dynamic platform as future work is a topic that will be identified and expanded on in future sections.

2.6.4 CAmkES specification

A CAmkES specification defines the components, interfaces and connectors in the multi-server operating system. The composition of these objects are described through a language provided in the CAmkES framework. Some of the CAmkES terminology will be expanded upon in the following subsections.

Interfaces

An interface is an exposed interaction point of a component. There are different types of interface types that a component can interact with, being procedures, events and port type interfaces. A procedure interface can be seen as a series of function/procedure prototypes which software components in the system will either implement or use. An event is an asynchronous signal interface similar to that of an interrupt. Lastly a port represents a shared memory interface between two components.

Components

A component is a single software item that is configured to implement or use a series of interfaces defined in its specification. In the context of this thesis, a component is a C program. Components can be in either an active or passive state. In an active state a component has a thread of control, containing a main function. In contrast, a passive state component lacks a thread of control.

In the context of seL4, a component has its own cspace that stores the capabilities of all the kernel objects it will use. In addition, active components will have their own virtual address spaces, page directories and thread control blocks (TCB).

Connectors

Connectors are a type of link between component instances. A common connector type includes a Remote Procedure Call (RPC) connector, allowing one component to invoke a procedure in other components. From the component programmers point of view, invoking a connection would simply involve calling a function however behind the scenes the CAmkES build system generates glue code to implement the specific connection.

2.6.5 VMM

Fundamental to developing a Qubes-based system on seL4 is the ability to instantiate VMs within the seL4 run-time environment. seL4 currently has a VM library implementation with mechanisms for constructing VMMs. seL4 has virtualisation support for x86 and ARM architectures, where seL4 runs as the hypervisor (in Ring-0 mode on x86 or hyp-mode for ARM), forwarding virtualisation events to a de-privileged VMM for emulation (running in Ring-3 mode on x86 or supervisor mode for ARM) [Kuz16]. Furthermore, VMs are supported in seL4 CAmkES. A CAmkES VM component consists of one VM per VMM, with support for communication mechanisms between VMs (dataports and events). Currently there is only 32-bit VM support on seL4.

VMM Library

seL4 has a library, libsel4vmm, that provides methods for constructing,initialising and managing VMMs. More specifically, this can be used to initialise the relevant VMM kernel objects, including a TCB, cspace, VCPU, Fault IPC endpoint, memory frames and extended page table (EPT) directory. The VMM library also provides mechanisms for booting the VMM, managing Direct Memory Access (DMA), catching VM exits and generating interrupts within the guest OS.

CAmkES Init Component

To use the VMM implementation in CAmkES on x86 a special initialisation component, Init, is used. Init is an active component required to perform the creation of kernel objects for the guest, initialise the necessary devices and setup a VM exit handler that waits for and handles VM exits. Implementation details discussed through this thesis will involve extending the VM exit handler to accommodate for necessary PV interfaces Qubes utilizes.

Cross VM Connectors

The CAmkES VM project also introduced mechanisms to connect guest Linux processes to regular CAmkES components. The connectors include:

- Dataports: mechanism to share memory between a CAmkES component and guest Linux process
- Consume Event: mechanism to wait for or poll for an event emitted by a CAmkES process
- Emit Event: mechanism to emit an event to a CAmkES process

The connectors are implemented through 3 kernel drivers in the guest Linux, in addition to Linux library syscall wrappers and utility programs to interact with the cross-vm mechanisms. These are compiled into the root file system image.

Chapter 3: Qubes

3.1 Overview of Qubes

Qubes OS is a security-oriented OS that is both free to use and open source. Qubes' approach towards security is through introducing a compartmentalised OS architecture. A compartmentalised approach to security focuses on separating the various services and applications that exist in a traditional monolithic kernel into separate isolated domains. In the case of Qubes, the seperate domains are VM's, typically hosting Linux guest operating systems. Each domain, also referred to as a "Qube", is isolated from each other preventing domains from being able to compromise others. An example application where this is useful can involve web browsing. In this scenario a domain is dedicated for visiting untrusted websites whilst another domain hosts user critical data e.g. online banking details. This way if the web browser application is compromised in the untrusted domain, the users online banking details will still be safe.

Qubes is currently supported on Xen with the ability to instantiate 64-bit Fedora, Debian, Archlinux and Whonix Linux VMs. Qubes is built to run on x86-64 processor architectures, utilizing specific hardware virtualisation support such as Intel VT-x, enabling multiple virtual machines to efficiently and safely utilize x86 processor resources.

The version of Qubes this thesis will focus on is release 3.2, however it is important to note that during the implementation of this thesis the Qubes team released version 4.0. The significant difference between these versions is that the Qubes development team decided to forego paravirtualisation support, opting to enforce hardware-based virtualisation (HVM) for all VM domains [Rut17]. A significant factor behind this decision is that a large majority of Xen security vulnerabilities that impacted Qubes throughout is development were due to bugs founds in Xen's paravirtualisation support.

3.2 Qubes 3.2 architecture

At a high level, the types of VMs the Qubes architecture relies upon can be divided into two general categories:

- **System VMs:** focused on providing system wide services e.g. networking and block device access.
- **AppVMs:** typically used to host user applications such as web browsers or email clients.

Illustrated in Figure 3.1, Service VMs include the Network VM and USB (Storage) VM. Interfacing Service VMs to hardware is an extensive use of Intel's VT-d extension, a virtualisation technology most important to Qubes' security model. Intel VT-d provides an IOMMU, allowing devices to perform direct memory accesses (DMA) to communicate with software. Intel VT-d is used to restrict devices in specific memory domains, preventing malicious devices from accessing regions of memory that would compromise system security. The components highlighted in blue make up the TCB of Qubes, being Xen and Domain 0. These components in particular manage the VMs and enforce system-wide policies.

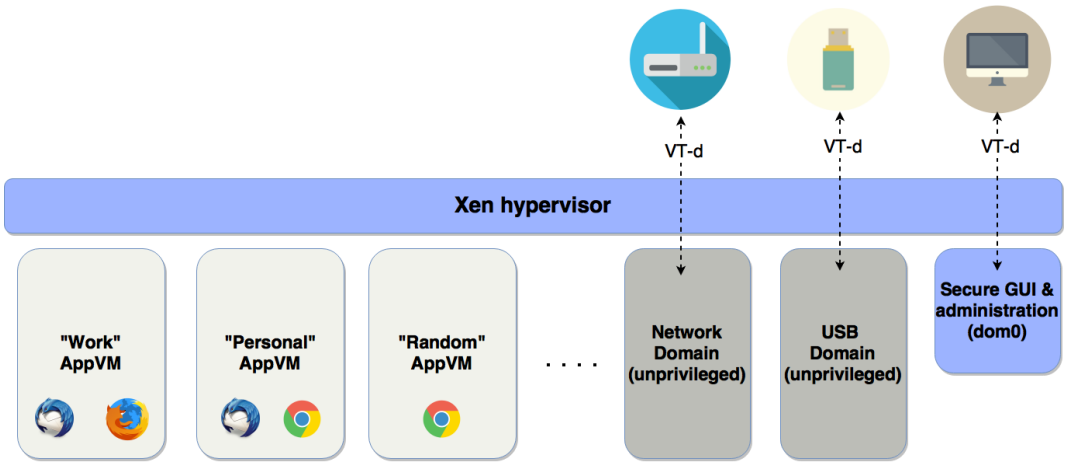

Figure 3.1: Overview of Qubes Architecture [RW10]

Further detail of the elements that exist within each of these VM's are illustrated in Figure 3.2. Unpacking the specifics of Qubes' architecture will be used as a means for identifying the critical components that would be essential to develop a Qubes-based system on seL4. Further subsections will identify each of these critical components and discuss their intended function.

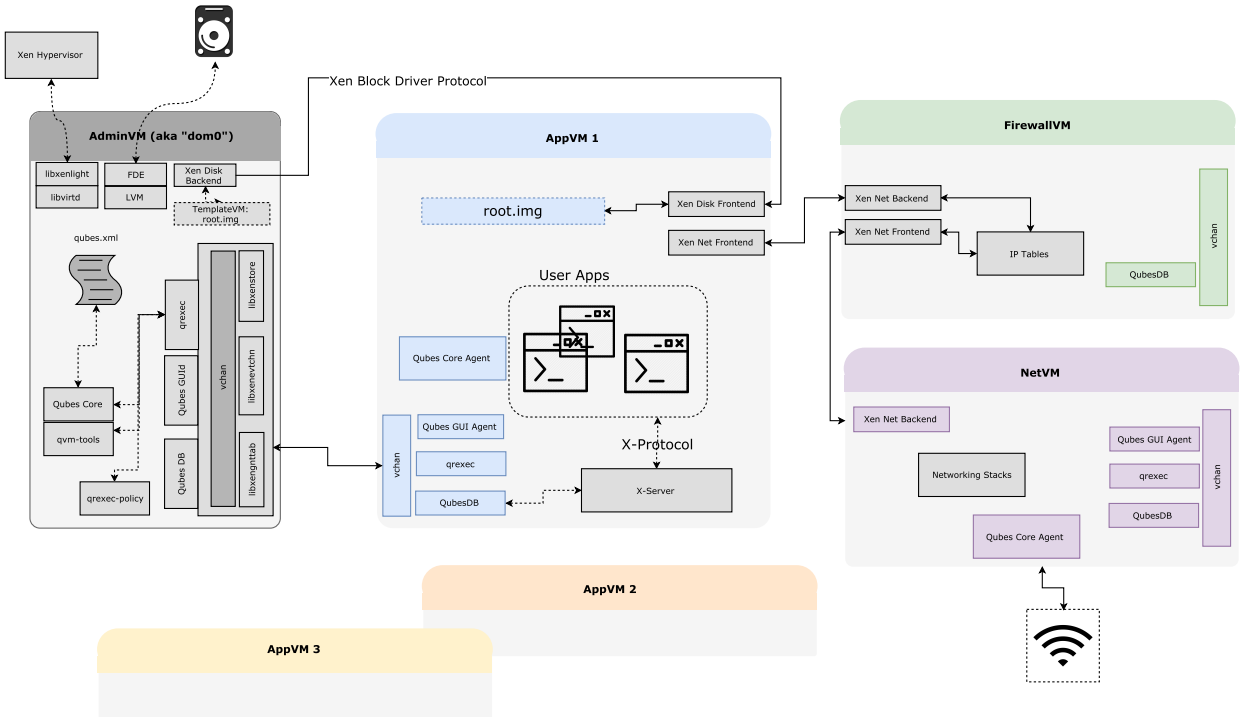

Figure 3.2: Qubes 3.2 Architecture

3.3 Xen Tools

Forming the basis of the Qubes architecture and its vchan infrastructure is its low-level usage of Xen toolstacks and libraries. The various libraries discussed form a paravirtualised interface, enabling the various domains in the system to communicate and share information with each other.

3.3.1 Event Channels (evtchn)

Event channels in Xen are the equivalent of interrupts, this being an asynchronous mechanism that enables Xen domains to become signaled in the event there may be data available for processing [Pro14]. Through Xen's event channel implementation, domains are free to dynamically create and close channels between one another, allowing for inter-domain signalling. There exist two implementations for event channels in Xen, the first being a shared memory+hypercall protocol in the guest OS kernelspace. The second being a userspace library implementation that interacts with a /dev/xen/evtchn device file. Interactions with the device file through a system call interface (e.g. open,read,write,close,ioctl) invokes the kernel level driver to carry out the event channel procedure.

Event Channels form the backbone of many Xen PV implementations, where through a combination of events and shared memory mechanisms allow for effective communication semantics between domains. For this thesis, event channels form the basis of vchans, a guest OS userspace protocol for sharing data between domains.

3.3.2 Grant Tables (gnttab)

Xen Grant Tables provides a generic mechanism for sharing memory between domains. Each domain in Xen holds a grant table. A grant table is a data structure which records the permissions of other domains with regards to accessing its pages [Pro13]. A domain is able to allocate one of its own pages and set up a grant reference, similar to a capability, which describes the permissions an unprivileged domain has over the page. The grant reference is an index into the granters grant table. Given the appropriate permissions, a grantee is then able to map the shared page into its address space.

Similar to event channels, grant tables underpin many of the split PV driver implementations as well as the shared memory mechanisms of vchans. It is also implemented through a hypercall protocol in the guest OS kernel space and exported to a device file for userspace library implementations.

3.3.3 Xenstore

The Xenstore is a filesystem-like database that enables applications and drivers in guest operating systems to share configurations and status information [Pro15]. Information is stored in key-value pairs with the ability to also create nested keys in a hierarchical structure. An example of the values that can exist in the xenstore can be seen in listing 3.1.

```
#xenstore-ls -f
/local = ""
/local/domain = ""
/local/domain/0 = ""
/local/domain/0/name = "Domain-0"
/local/domain/0/memory = ""
/local/domain/0/memory/target = "524288"
```

```
/local/domain/0/memory/static-max = "524288"
/local/domain/0/memory/freemem-slack = "1254331"
/vm = ""
/libxl = ""
```

Listing 3.1: Example Xenstore values

Each domain gets its own section in the store which it can then overlay with permissions for other domains to read and write to. Additionally applications are able to set watches for specific keys allowing them to recieve notifications when specific key values are created or changed. The stored information is meant for small values thus is not a suitable system for large data transfers. The xenstore is however convenient for sharing information such as grant table references and created event channels, allowing domains to create shared channels in a dynamic manner.

The implementation is again similar to event channels and grant tables, having both a kernel driver and a userspace library implementation. The xenstore data is hosted in dom0.

3.3.4 Libxenlight (libxl)

Libxenlight is a low level C library intended to provide a simple API to the Xen toolstack. Libxl provides various mechanisms including the ability to create, shutdown, reboot and pause guest VMs, hotplug and unplug devices to guest VMs and perform live migrations of VM. Thus libxl provides a valuable interface for VM management.

3.4 VChan Library

The key mechanism that allows for inter-VM communication is a VChan library. The vchan library is a Xen specific library exposing a socket-like (datagram-based) interface for VMs to perform inter-domain communication. The vchan library is implemented for userspace applications running in the guest operating systems, allowing for the applications to communicate across domains. The vchan implementation initially originated out of the Qubes project but was later adopted by Xen and integrated into the Xen 4.2 release [Pro11].

A vchan refers to a single channel created between a client and server domain, strictly being a 1-1 channel. Internally the vchan library uses grant tables and event channels to establish client/server communication. Abstracting over the shared memory and event semantics the library provides a set of asynchronous and synchronous mechanisms to read and write from a vchan.

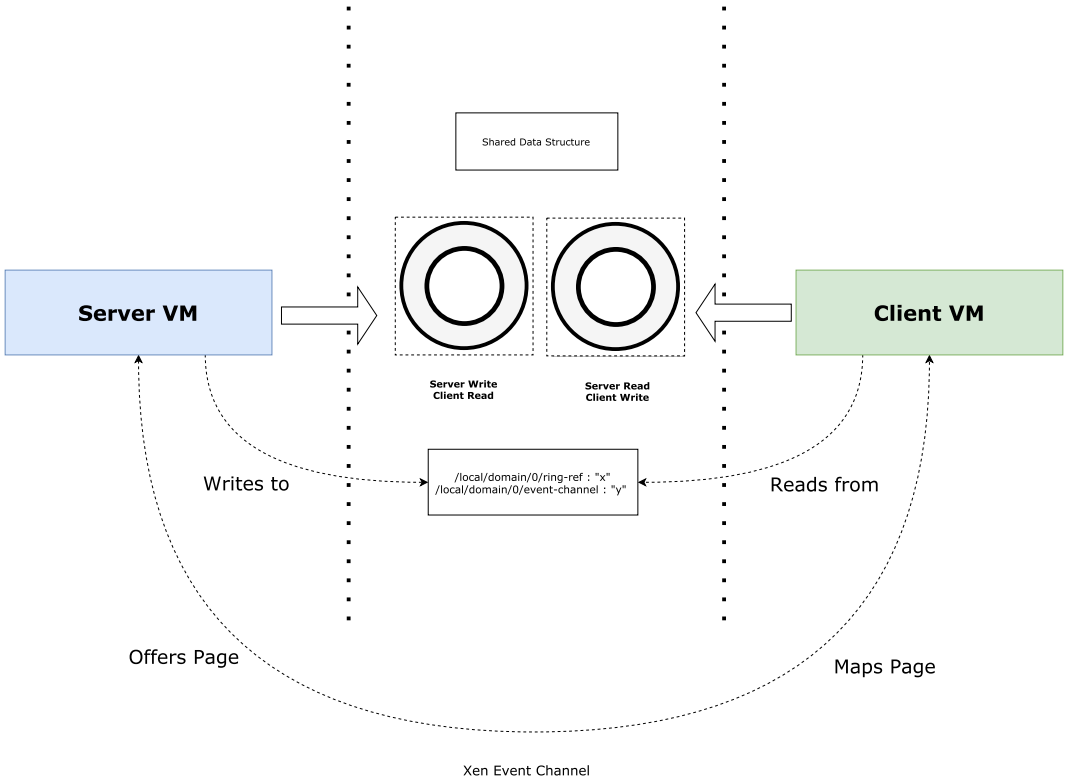

Figure 3.3: VChan Initialization [XZh16]

When creating a vchan between two domains, a server domain will allocate a shared memory region (through grant tables) to host two ring buffers (being a write buffer each for the server and client) and an event channel port to signal the client domain with. Both domains will read from each others write buffer. The server domains will offer the grant pages to the client VM who in turn will map the region into its own address space. To do so, the server will store the event channel port and the grant table reference into the xenstore. This being readable by the client domain who will then use the grant references to map the buffers into its own address space. The client will then proceed to listen on the created event channel. A highlevel overview of this process is seen in Figure 3.3.

3.5 Qrexec

The vchan library underpins Qubes' core RPC mechanism Qrexec. Qrexec enables dom0-to-VM and VM-to-VM remote execution. An example of an important use case includes using Qrexec to start an application in an AppVM. The key components making up the Qrexec system are demonstrated in Figure 3.4. This mainly consists of a qrexec-daemon (in dom0) for every AppVM. Every AppVM contains a qrexec-agent daemon to communicate with their corresponding qrexec-daemon in dom0. The qrexec-client and qrexec-client-vm components are programs to initiate an RPC call between two VMs.

The flow of a dom0-to-VM qrexec call involves the user running qrexec-client in dom0, specifying the target VM and procedure to execute in the VM. This triggers an IPC with the qrexec-daemon (within dom0) that manages the vchan server interface for the specific target VM. The qrexec-daemon subsequently forwards the command to the qrexec-agent in the target VM to be executed.

VM-to-VM RPC behaves similarly however all initial communication is mediated through dom0, where the qrexec-agent (invoked by qrexec-client-vm) forwards the command to its corresponding qrexec-daemon in dom0. The qrexec-daemon then translates the call to invoke dom0's qrexec-client, executing the command in a similar fashion as the dom0-to-VM RPC sequence. Illustrated in Figure 3.4, a VM will invoke an RPC service ("qubes.someRPC"), which corresponds to a policy file in dom0. The policy file defines what VMs are permitted to use the RPC service. Making dom0 the centralised controller in the sequence allows the administrative domain to enforce policies for VM-to-VM qrexec (e.g. preventing bash execution) adding an extra layer of security over the RPC mechanism. The administrative user can further install policy files in dom0 to enable or disable qrexec services between VM's.

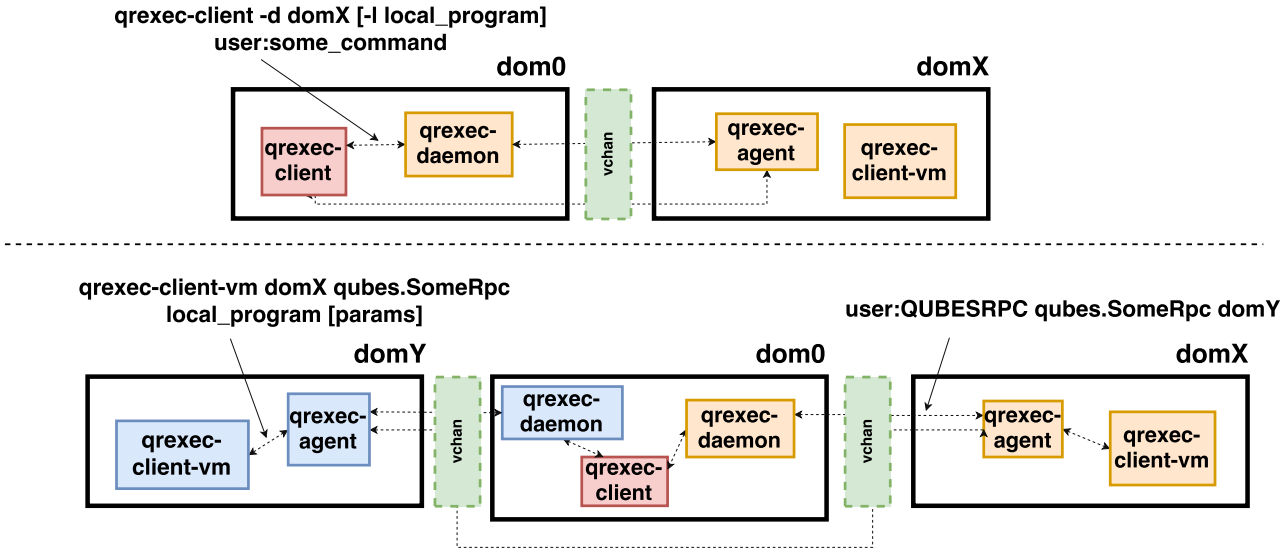

Figure 3.4: Qrexec configuration: Dom0-to-VM & VM-to-VM

3.6 Qubes VM Classes

3.6.1 Admin VM (dom0)

In Qubes OS, all administrative tools and utilities to manage the system exist within dom0. Given dom0's management facilities and hardware access privileges in a standard Xen configuration, dom0 becomes almost as privileged as the hypervisor. Being a security critical element, compromising dom0 would essentially compromise the entire system. With a design goal of Qubes OS being to minimize the overall TCB of the system, special care was given to limiting the amount of world-facing code running in dom0, decreasing the likelihood of an attack. Subsequently a significant portion of hardware facing code was stripped out of dom0 (e.g. no networking card access).

Significant programs that reside in the Admin VM include:

Block/Disk Backend Drivers (blkback): A PV backend driver with passthrough access to disk. The kernel and root file system images for the various VM's Qubes can create are stored on disk, managed by the Admin VM. The blkback driver allows the Admin VM to export virtual block device interfaces, which host the kernel and file system images, for other domains to boot, read and write from. More specifically, this mechanism allows us to configure the file systems of our

AppVMs. This configuration is illustrated in Figure 3.5. The illustrated configuration allows us to serve the root file system images as a read-only block device, in addition to a discardable per-VM copy-on-write image to allow for a RW root file system. We can then share the root file system between multiple AppVMs, discarding the copy-on-write image once the VM is destroyed.

Qubes Admin Modules: A python library that is responsible for managing the Qubes framework. The python modules provides mechanisms for creating, starting and configuring VMs and book keeping all the VMs the user has created. The admin framework maintains a "qubes.xml" file in dom0 to track all the VMs the user has created, its various configuration settings and the location of its kernel and filesystem images.

Libvirt API: This is a generic virtualisation API that is designed to be compatible with a variety of different hypervisors and virtualisation solutions. The Qubes admin module utilises the libvirt libraries to create and destroy VMs within the system.

qrexec-daemon and policies: A daemon that manages the RPC interface with another VM in the system. The admin VM also contains a series of policy files that define the restrictions of VM-VM RPC services.

qvm-tools: A series of python based command line utilities that invoke the various mechanisms within the Qubes Admin modules. This involves creating, starting, and destroying VMs and running executing services in other VMs.

GUI subsystem: involves running an X Server, a Window Manager (e.g. xfce, i3) and a special GUI daemon that catches the X11 graphical content from AppVMs to be rendered to the display. Accordingly, the GUI subsystem needs direct access to the graphics device as well as input devices, such as keyboard and mouse. The interface to the devices is mediated with Intel VT-d.

Qubes DB An additional key-value database, similar to Xenstore, used to store and share Qubes configuration information. This is implemented using vchans.

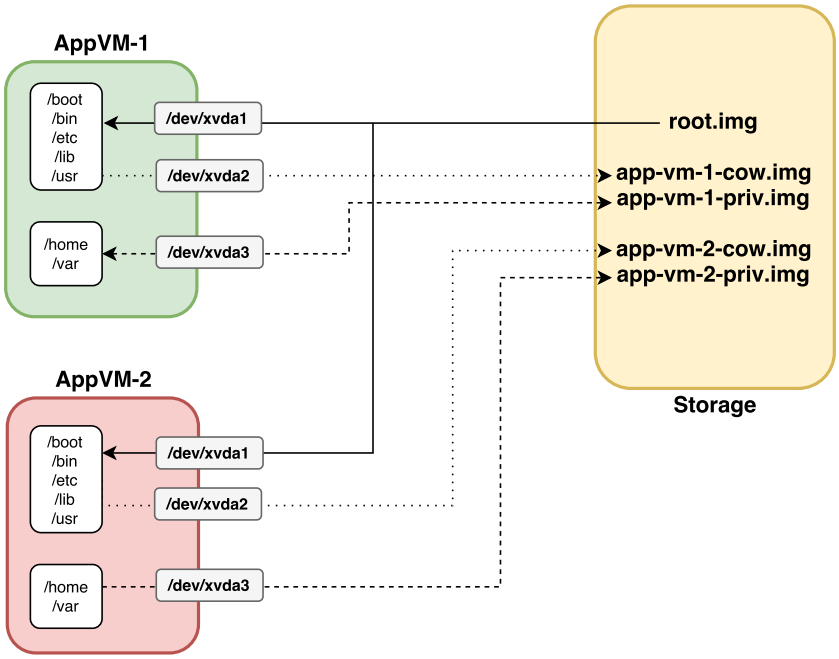

Figure 3.5: Storage Configuration

3.6.2 Service VM

The Service VMs are used as a means of moving core system services that would typically exist in the one monolithic kernel out into separate VMs. These VMs typically contain code with physical hardware access, delegated with the responsibility of exposing the hardware as virtual device interfaces to the other VMs. Core service VMs in Qubes are the NetVM and USBVM, with an overview of their configuration seen in Figure 3.6. The benefit of this compartmentalized scheme is that it effectively sandboxes potential bugs and vulnerabilities found in the mentioned system services (e.g. device drivers), preventing an attacker from compromising the entire system if exploited.

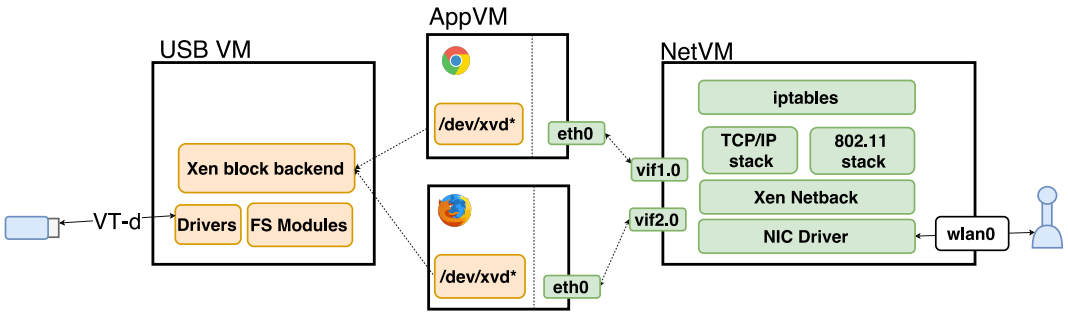

Figure 3.6: Service VM Configuration

Net VM

The NetVM has direct access to the networking hardware, granted via Intel VT-d, and the associated drivers and protocol stacks that make up the networking subsystem. Furthermore, Xen Netfront and Netback drivers are used to facilitate a virtual networking interface to AppVMs. Components such as iptable configurations can be further brought out into separate VMs, forming a FirewallVM. FirewallVMs look similar to a NetVMs in that they host backend network drivers that AppVMs interface with, whilst they also use a frontend network driver to communicate with a NetVM that has hardware access.

USB VM

The USB domain is similar to the NetVM in that it has physical block access to USB controllers. It also exposes a virtual interface for other VMs to access, through a front and backend driver configuration.

3.6.3 AppVM

The AppVMs in Qubes are used to host user applications. These are typically Linux VMs, although Qubes can support Window-based VMs. The configuration of an AppVM is relatively minimal compared to its Service VM counterparts. Each AppVM consists of a virtualised X server and a special GUI agent daemon to send frame buffers back to the secure GUI subsystem in dom0. The interface between these VMs is minimal, using Xen Shared Memory as an efficient zero-copy mechanism to send graphical content to dom0. An overview of this configuration is shown in Figure 3.7. On start-up of an AppVM, the root file system images of each VM is served by dom0 as a read-only block device, in addition to a discardable per-VM copy-on-write image to allow for a RW root file system. Additionally, VM-specific private data

(e.g. /home) is kept on a per-VM block device backed by dom0. Figure 3.5 depicts an overview of this configuration.

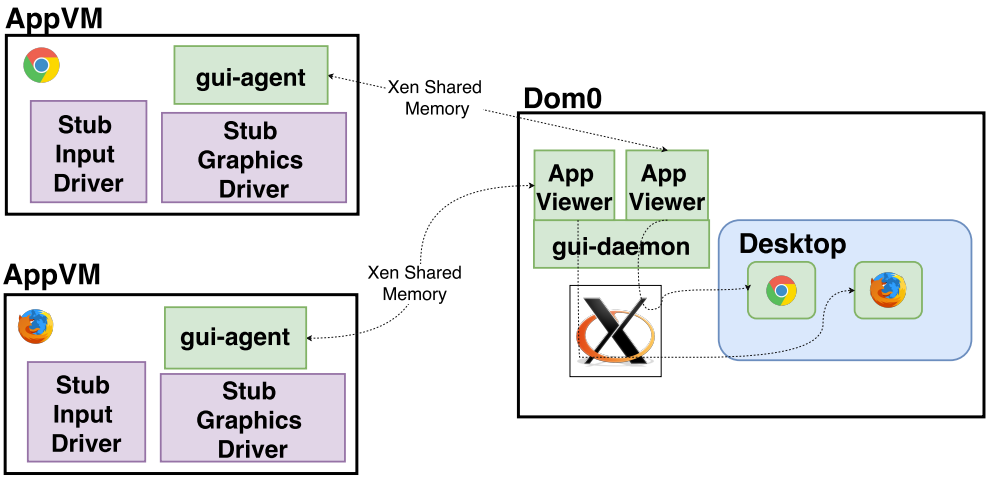

Figure 3.7: AppVM GUI Configuration

3.6.4 Template VM

AppVMs are based on pre-defined configurations known as TemplateVMs. A TemplateVM in Qubes defines a kernel and root filesystem image which AppVMs are able to derive off. This being beneficial as it dramatically saves disk space, only needing to maintain a private image for AppVM specific data.

3.6.5 Additional VM Classes

In addition to the VM classes mentioned there also exist Firewall VMs, VPN/Proxy VMs and Disposable VMs. These VMs are considered as extensions to the basic VM types already discussed in that they introduce additional attributes to the base NetVM and AppVM classes. Supporting these VM classes was out of the scope of this thesis and hence are not a primary focus of design and implementation discussion.

3.7 Hypervisor Abstraction Layer (HAL)

With the third release of Qubes (R3), Invisible Things Labs introduced the Hypervisor Abstraction Layer (HAL) [Rut13]. The goal of HAL is to abstract and modularise the core Qubes services from the hypervisor. This effectively removes hard coded Xen dependencies by decoupling Qubes from Xen. The aim of HAL is to make Qubes relatively easier to port to an alternative hypervisor architecture. To port Qubes to an alternative VMM, the HAL backend requires two core parts:

- **A libvirt driver:** Libvirt is a virtualisation API providing a general framework for interacting with hypervisors. Through the API, a user can manage VM instances and hardware devices without having to rely on specific hypervisor control stacks. Libvirt functions as a driver within dom0 with

which Qubes heavily uses to start and stop VMs. In Xen, the libvirt driver depends on libxl, in a sense being a wrapper around its functionality.

- **A vchan library:** A library to facilitate inter-VM communication. When moving to an alternative VMM, a port of the library is necessary to utilize the given inter-domain communication mechanisms exposed by the VMM API.

Chapter 4: Related Work

4.1 Application Security for Operating Systems

The goal to secure desktop operating systems through sandboxing applications has been explored in various works, reflecting similar themes found in Qubes' design.

4.1.1 Virtics

The Virtics project [PJ10], introduces an operating system primitive that involves instantiating unprivileged Linux VMs (HVM) using KVM to run applications when dealing with untrusted data. Similar to Qubes' concept of AppVMs, each Linux VM runs a virtualised X server whose graphical content gets mapped to the display by a trusted Linux host. A key point of difference in Virtics is that the deployment of VMs are transparent to the user, being automatically created when a user works with a separate application or document. In contrast, Qubes users need to be proactive when creating VMs for different applications. The application of Virtics was demonstrated through armored versions of a PDF and web browser application where VMs would be created for separate documents and tabs. The project was reported to be in "daily use for over a year" when the technical report was published however there hasn't been any further discussion on the project since, thus the current state of the project is unknown.

4.1.2 Genode

An active and regularly updated project is the Genode OS framework [Fes16], which further explores the concepts of creating secure sandboxed environments for operating system applications and processes to execute within. The Genode framework organises the processes in the system into a hierarchical fashion, where a parent process is able to create a sandboxed environment, being granted with specific access rights and resources to fulfill its specific purpose. A parent can only give the child process the resources it has permissions to use, which in turn applies to the child when it creates it own processes. This in turn creates a structured tree of processes where each parent maintains full control over its children and their relationship to other resources in the system. To bridge compatibility to existing applications, the Genode framework also supports virtualisation, enabling VMs to run within their own secure sandboxed subsystems. With ready to use drivers, GUI and networking stacks, Genode and Qubes share similar design goals in building secure compartmentalized operating systems.

Further, Genode is supported on a variety of L4 family kernels including seL4. In particular, Genode has recently made significant strides to supporting its framework on seL4 with its release of the Genode OS Framework 17.08 [Lab17b]. This added support to ARM and 64-bit x86 architectures and multiprocessor support. Given Genode's current support for seL4 combined with a framework that reflects similar themes in Qubes' design, using Genode to support Qubes is also a feasible solution that aligns with the goals of this thesis. Qubes on Genode is a project Genode developers have expressed interest in [Lab17a], however Genode's current support for seL4 does not include VMM support (based on release 16.08 notes [Lab16]). Nevertheless future work based from this thesis should still be applicable to a Genode-based system as its support for seL4 continues to mature.

4.1.3 Dune

Dune [BBM+12] explores an alternative strategy for application security, leveraging hardware virtualisation features, such as ring protection, page tables and tagged TLBs, to create sandboxes for user-level program execution. Belay introduces a Dune kernel module, running within an existing host OS kernel that initialises and mediates safe access to VT-x and privileged hardware features for user programs to utilize. Thus similar to the effect of Qubes isolating applications in VMs, Dune enforces security through sandboxing processes in privilege ring protection modes. This approach allows Dune to effectively filter and restrict the behaviour of an untrusted binary by catching unsafe operations as exceptions. Compared to VM based sandboxing solutions, Dunes approach can be seen as advantageous where its kernel module offers a simplified and efficient solution to isolating processes, avoiding the overheads of creating VMs to run applications. Secondly a process using a Dune module doesn't need to be separated from its host OS, this avoiding the process from being decoupled from the core services of its host e.g. file system and devices. Whilst beneficial, the Dune kernel module relies on running within Linux, keeping the monolithic kernel as the TCB of the system. In contrast, Qubes' overall design seeks to break away from making a monolithic kernel the TCB of the system. Thus a sandboxed Dune application may still have access to a multitude of APIs found within a monolithic kernel, presenting a large attack surface to potentially compromise the system.

4.1.4 Bromium

Sandboxing applications on the host OS is similarly achieved through the Bromium microvisor [Bro17], a commercial type-2 hypervisor solution derived from the Xen code base and developed upon by initial Xen developers. Developed to run within the Windows operating system, the microvisor platform is able to instantiate specialised micro-VMs to host specific user tasks e.g a web browser tab or PDF document. Therefore, if a program was compromised by any malicious process, it would effectively be contained within its sandboxed environment, preventing it from compromising the rest of the system.

4.2 Compartmentalization and Isolation

Operating system compartmentalisation is a significant focus of Qubes, moving system services that make up a significant portion of the attack surface for a monolithic kernel out to isolated unprivileged domains, with the key goal being to reduce the OS TCB to a minimum. The idea of compartmentalising OS system services can be seen in the Xoar project [CNZ+11]. Incorporating the modularity and isolation principles from microkernel based architectures, Xoar introduced a modified version of the Xen platform by compartmentalizing the control VM, dom0, into single purpose service VMs. Service VMs included domains dedicated to managing physical block and network devices, exposed as virtual devices to guest VMs. This idea is similarly carried over in Qubes' architecture, for example, delegating network card access to a NetVM.

4.3 Past work

The idea of porting and developing a Qubes system on seL4 has been a discussed and explored topic for numerous years. A series of projects have been undertaken to look into making various steps towards achieving a full Qubes port. Notable investigations into the area involve the development of a vchan

library and demonstration of a basic qrexec RPC protocol from 2014 and 2016 Taste of Research projects (Muir [Mui14] and Tugai [Tug17]). The vchan implementation has since been absorbed into the CAmkES VM project. The implementation of vchan however was limited, such that a vchan connection and size was statically defined in the CAmkES specification. This was unfortunately unsuitable to support higher level Qubes code that would dynamically create and close vchans with other VM's. In addition the qrexec port had since been lost. The following design and implementation sections will describe a new approach to achieving vchans and its ability to support the Qubes code base with little to no modifications of its code.

Chapter 5: Design

To a significant extent, the primary hurdles of porting Qubes to seL4 comes down to engineering. There are a variety of risks and roadblocks that will be identified in further sections that made a full port of the Qubes architecture unfeasible in the time frame of this thesis. Rather we focus on implementing and porting a series of identified sub-systems in Qubes' architecture that can together make up a minimal viable representation of a Qubes OS. The components that make up a minimal viable Qubes will be identified in the following subsections.

5.1 Risk and Roadblocks

Prior to breaking down the design for porting Qubes to seL4 it is important to identify the various risks and roadblocks of the project that motivated the various design choices of the implementation. The following roadblocks that exist which prevent a full unmodified version Qubes to run on seL4 include:

- **64-bit virtualisation support:** Currently only 32-bit VMs are supported on seL4. An unmodified Qubes is built only to support 64-bit VMs. Developing support for 64-bit VMs is an involved task, where a 64-bit VM requires a greater range of general-purpose CPU registers and larger address range support. Due to the complexity of the task, developing 64-bit virtualisation deviated from the main aim of the thesis and thus was out of scope during the implementation. Instead the focus was taking individual components out of Qubes' build system and porting them to run in a 32 bit Linux VM system.
- **Dynamic VM management:** At present there is no dynamic system for managing VMs on seL4. Qubes on Xen is a dynamic system where VMs can be instantiated and destroyed when needed. A user desktop operating system is a highly dynamic ecosystem where users at any point can start and stop applications on request, thus dynamically managing VMs is an important requirement for Qubes. Developing a dynamic system for VMs is however a significant task which again deviates from the main focus of the thesis. The implementation rather focused on running Qubes components in a static CAmkES system.

Finally, it is important to realise that Qubes is a complex system, being continuously developed and iterated upon since its inception in 2010. Qubes is a fully featured desktop operating system with a huge number of services, applications, utilities and UI widgets that make it a complete package. These components are tightly coupled and interwoven in a complex build system. Successfully taking components out of the Qubes build system to run in an alternative environment required careful attention to dependencies that where either provided, revised or removed.

5.2 Designing a Minimal Viable Qubes (MVQ)

Given the complexities of the Qubes architecture, the focus of this thesis was not to achieve a full port of Qubes to seL4. The goals were rather to determine the feasbility of a Qubes system on seL4 and to start

making incremental progress towards acheiving a Qubes-based system. The end result being to create a minimal viable version of Qubes that demonstrated these goals.

To achieve minimal viable Qubes, a key focus was to identify a subset of systems that would make up a usable Qubes-based system. In identifying these system, an important first step was to first assess whether it was feasible to take the chosen components out of the Qubes build system and run them in a minimal environment. Furthermore it was important to learn if this was even possible in a native Xen system. Thus the first milestone of the thesis was to develop a minimal viable Qubes on Xen. The Xen MVQ served importance in influencing the final design of the seL4-based implementation with the benefits being two fold. Firstly it helped identify all the required dependencies of a component when detaching it from its complete build. Secondly, with all the dependencies and source files identified, it simplified the migration of an MVQ build to seL4 and ultimately contributed to a stronger understanding of Qubes inner-workings. Discussion of the Xen MVQ is intended to identify the basic design building blocks where further discussion as to how the components work in an seL4-based system will be found in the implementation section of the report.

A refinement of the Qubes 3.2 architecture (shown in Figure 3.2), produced the Xen MVQ design illustrated in Figure 5.1.

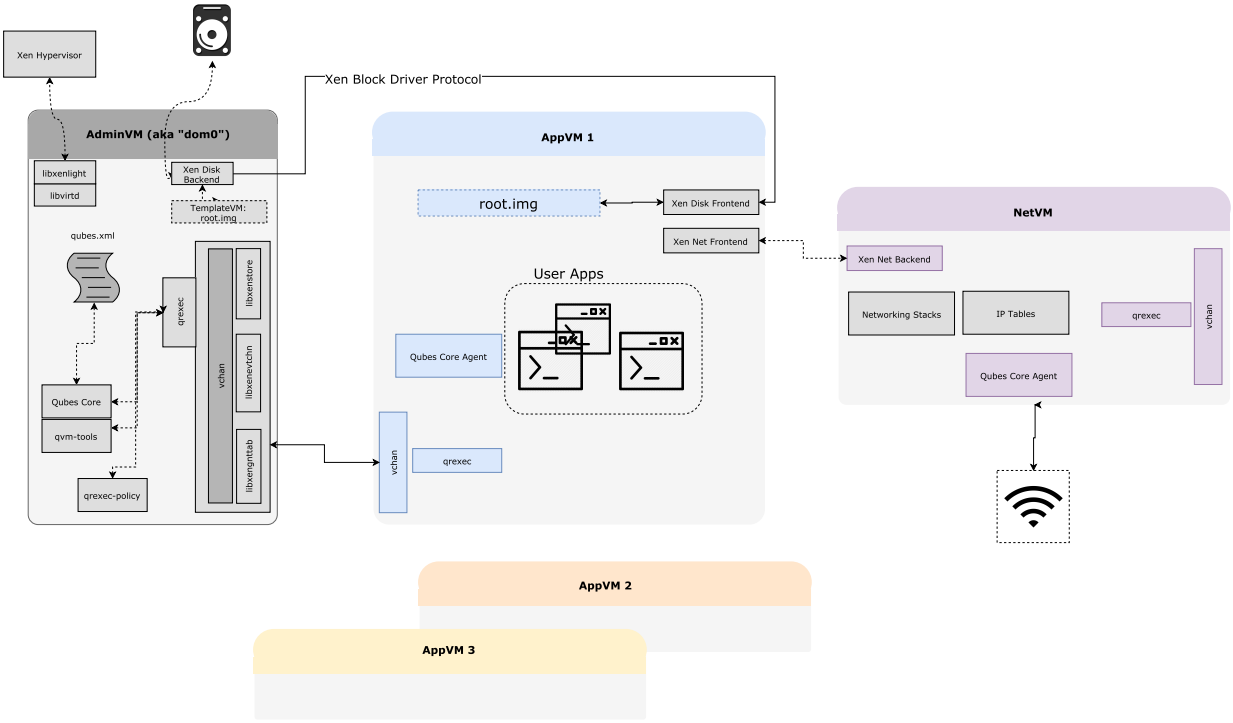

Figure 5.1: Overview of the Xen MVQ Architecture

The components being integrated in the MVQ build are taken from the Qubes 3.2 release. At the time of development the 4.0 code base wasn't available, however inspection of the 4.0 code base post-release reveals the changes between releases aren't too significant with regards to the components we are using in the MVQ. When developing the MVQ, some of major components omitted from the full Qubes system where the GUI infrastructure and the Firewall, USB, Proxy and Disposable VM types.

The primary components in the Xen MVQ where:

Xen Hypervisor: This of course being the fundamental building block of the MVQ. An x86 64-bit build of Xen 4.9.0 release was used for the MVQ (based on the "RELEASE-4.9.0" tag from Xen git project [Pro17b]). Domain 0 hosted a Ubuntu 16.04 LTS 64-bit Linux OS (PV).

VChan Library: VChans are a critical component in Qubes, forming a backbone for RPC mechanisms and VM management in Qubes. In the MVQ, VChans are necessary to support the Qrexec framework. In turn the Qubes Admin stack invokes the Qrexec binaries to execute programs/commands in other VMs. Thus a working vchan library is a base dependency to have established in an MVQ build. Since vchans have been integrated into the upstream Xen toolstack, getting vchans to work is a fairly trivial exercise as the libraries are already installed in the Xen-based system that is being used. The VChan library are implemented in C with library dependencies to Xen tools (xenstore, gnttab, evtchns).

Qrexec: The qrexec components were identified critical to have in the MVQ to faciliate RPC functionality between VMs. The only dependency required to build and use the Qrexec programs were the vchan libraries. With the establishment of a working vchan library, building and using Qrexec was again a fairly simple exercise. Care had to be taken however to ensure the binaries where installed at the right locations in the Admin and AppVM file systems. The qrexec binaries are rarely exposed to the user, rather they are utilized by the Qubes Admin modules, expecting the binaries to exist in specific locations. The qrexec programs are implemented in C.

Libvirt Installation: Libvirt is extensively supported on the Xen platform and it was possible to recycle it for the Xen MVQ. The Libvirt 3.7.0 API was built against the Xen 4.9 libraries and then installed into the MVQ system. Important to the libvirt installation was the installation of its python library bindings which the Qubes Admin code base uses to manage VMs.

Qubes Admin Modules: The Qubes admin code base is implemented as python library modules. A basic Qubes VM is represented as a python class. The various Qubes VM types (e.g. AppVM, NetVM) are also represented as python classes that extend the basic VM class with additional attributes. When a user creates a new VM, the python class for that VM is instantiated. Furthermore the state, name and configuration attributes are recorded in a central xml file (qubes.xml) so that the class can be re-instantiated at a later point. The python classes make extensive use of the libvirt python binding to start, pause, resume and shutdown the VMs on Xen. When integrating the python code base into the MVQ, careful modifications needed to be made to remove usages of unsupported features e.g usages of GUI code and unsupported VM types. The final implementation of the modules were installed with in the admin VMs system python libraries folder (e.g. /usr/lib/python2.7/).

qvm-tools: A subset of the qvm-tools where chosen to be integrated into the MVQ. The qvm-tools represent an interface that allows the user to invoke the mechanisms in the Qubes admin stack. The key tools chosen are qvm-create (creates a VM), qvm-start (starts a VM), qvm-run (run a command on a specified VM), qvm-kill (kills a VM), qvm-remove (removes a created VM from the qubes.xml file) and qvm-ls (lists all the created and running VMs).

AppVMs: A sample set of AppVM kernel and filesystem images I created for the MVQ. Creation of the AppVM images was a manual process that involved creating new Fedora Linux VM installations that were manually configured to have the qrexec binaries and Xen VChan libraries installed. In addition, initialisation (systemd) scripts needed to be installed to ensure the Xen PV drivers where loaded (evtchn, gnttab and xenstore) and the qrexec-agent daemon was started on boot.

TemplateVMs: The newly created AppVM kernel and filesystem images formed the basic images that make up a TemplateVM. These were manually installed in the appropriate AdminVM library folders (/var/lib/qubes/vm-templates).

5.3 Minimal Viable Qubes on seL4 & CAmkES

When moving to the seL4-based implementation of the MVQ, we want to achieve the following:

- Minimal to no modification of higher level Qubes code
- Code reusability for future iterations of Qubes on seL4

As discussed, Qubes is a complex system. The first goal is particularly important in order to reduce the engineering cost of bringing Qubes to seL4. Thus a significant focus of the implementation involved targeting the components below the Hypervisor Abstraction Layer such that higher level Qubes code would not be need to be modified.

On the issue of code reusability, the MVQ implementation on seL4 involved using a static system on CAmkES with a set of predefined pooled VMs as components. An important future iteration of Qubes will ultimately invole moving to a more dynamic solution. Despite the static restrictions of a CAmkES base, my implementation focused on hiding the static nature of the system from a VMs perspective, in a sense mocking a dynamic system. This would mean little to no modification of the code within the VMs for later iterations of Qubes.

5.3.1 Porting VChans

The first component to focus on in the hypervisor abstaction layer was the vchan library. The CAmkES VM project had a vchan implementation from previous Qubes projects [Mui14], however using this implementation to support higher level Qubes code proved problematic. The main limitation of the implementation was that a single, fixed buffer size vchan connection between two VMs was statically defined within the CAmkES spec. The creation of vchans in Qubes is however a dynamic process where a Qubes VM can start up and close multiple connections (of varying buffer sizes) with the same VM on demand, this being particularly exercised by the qrexec framework. Our focus is on keeping the behaviour of vchans as close to Xen implementation as possible, whilst also maintaining the minimal modification approach. This was done by reusing the Xen vchan implementation but supplying my own library implementations of the lower level vchan dependencies, being xenstore, event channels and grant tables.

Xen Toolstack Equivalents

VChans depend on the implementation of three libraries: xenstore, event channels and grant tables. Below we see the approach to supplementing these dependencies on seL4.

Event Channels: The basic functionality of event channels involves a VM allocating a channel, designated for a specific domain (using its domain id). In return, the allocator gets back a port number,

identifying the event channel. The only other domain that can use the event channel is the designated domain. The recievee will open the event channel with the identifying port number. The VMs require a user level library implementation to use the event channels and a Linux kernel driver to perform hypercalls. The managing of event channels ports and permissions on Xen is an internal hypervisor implementation. For an seL4 based solution we require a seperate seL4 userland process (CAmkES component) to manage the event channel bindings. We also require extensions to the VMM implementation to handle a event channel hypercalls. Significant focus was spent on implementing the kernel driver, VMM handers and the event channel component. The userland Linux library implementation was recycled from the Xen project meaning the vchan libraries use of event channels did not need to be modified.

seL4store: The seL4store is a re-implementation of xenstore. The xenstore implementation uses dom0 to host all the key-value data. Keeping the xenstore in dom0 however increases the TCB of the system, thus my implementation looks to take the xenstore out of dom0 and run it as a CAmkES component. Aside from loosely basing my Linux library API on Xen's version, the seL4store implementation was developed from the ground up. The implementation involves a CAmkES component, Linux userland library, kernel driver and VMM hypercall handling mechanisms.

Share-Allocator: My equivalent to grant tables is the share-allocator component. Given the static nature of a CAmkES system, all kernel objects and capabilities required by the system are pre-defined by a generated CapDL spec. Thus dynamically allocating untyped memory becomes problematic for my seL4 based implementation. Given this limitation, my approach involves defining a large pool of memory (dataport), shared between all VM components. This is managed by a share-allocator CAmkES component, similar in nature to a frame table implementation managing physical memory. Through a Linux userland library and kernel driver, guest VMs make calls to the share-allocator component to allocate pages of memory. In return they recieve an index corresponding to an offset into the shared pool, representing the contingious shared memory region they allocated. Allocation is at a page-level (4K) granularity.

5.3.2 Libvirt and VM Manager

The second item to focus on in the hypervisor abstraction layer is the Libvirt layer. Unfortunately, due initial difficulties cross-compiling the libvirt platform for my 32 bit Linux platform and time restrictions during the thesis, a libvirt driver port was not feasible. Rather the approach taken was to emulate the python libvirt API in-order to satisfy the admin module code-base. A 'libvirt' python library was implemented, providing an identical API that the admin module code was able to import and use.

The python libvirt implementation invoked my vm-manager implementation. The vm-manager abstraction is in similar nature to libxl on Xen. A userland Linux library provided methods to create and free a VM and also get the callers domain ID. Similar to seL4store and event channels, a CAmkES component was used to manage available VMs (being spooled), and a Linux kernel driver was implemented to carry out the hypercall procedure.

5.3.3 Porting Qrexec and Qubes Admin Modules

The intention of the above design was to avoid modifying qrexec and the python admin modules. Qrexec was able to be built against my new library headers and the admin modules was able to simply import my libvirt python library.

5.3.4 Ethernet Component

To supplement the functionality of a NetVM, an existing implementation of an ethernet driver component in CAmkES was used. This work being recycled from from the CamkES VM project involves an eth1000 driver implementation running as a seperate component on seL4. VMs interact with the component through a virtio interface, a paravirtualisation standard for network and disk drivers. This could be seen advantageous over a using another VM to provide network access, having a smaller code base and memory footprint. Disadvantageous to this approach however is that the configuration become less portable when moving to different ethernet hardware.

5.3.5 Fileserver

A late design decision during the implementation of the thesis was to use a VM as a file server, having passthrough access to disk and USB hardware. This was necessary as compiling multiple Linux kernel and rootfile systems images into the boot image consumed a significant amount of memory at runtime. Thus a minimal file server kernel and rootfile system image is compiled into the boot image and booted at run time. Once booted, it accesses the remaining kernel and rootfile system images stored on USB, serving the images to the other VMM components.

5.3.6 MVQ Architecture

In summary of the discussed items, an overview of the architecture is illustrated in Figure 5.2. The demonstrated configuration shows one Admin VM and two AppVMs and the various CAmkES components in the system.

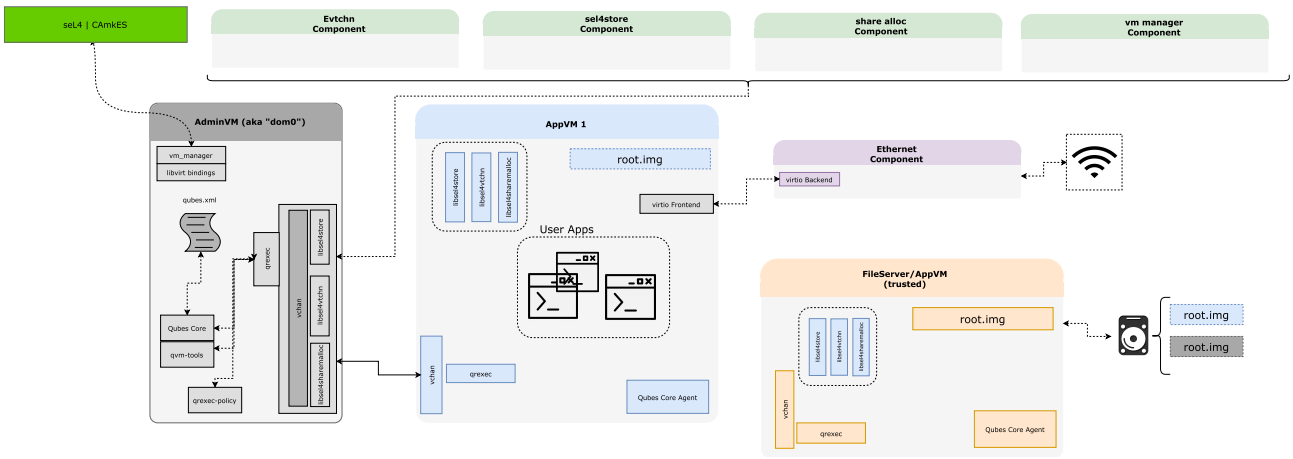

Figure 5.2: Overview of the seL4 MVQ Architecture

Chapter 6: Implementation

In this section we focus on the implementation details of the various features and components that make up the MVQ. The structure of the discussion will look at the implementation from a bottom-up view to help identify the lower level dependencies that provide the hypervisor abstraction layer.

6.1 MVQ Configuration

The seL4 CAmkES-VM project was used as a base for the project. Implementation focused on the x86 32-bit architecture, with x86 32-bit guest Linux VM's.

6.1.1 Guest VM Images

Admin VM (dom0): I used a Tiny Core Linux (TCL) distribution guest image for the Admin VM. TCL is a minimal Linux distribution noted for its small image size, making it suitable for the MVQ build. In addition, TCL provides a small set of community driven tools and libraries. This was useful for installing a python interpreter, being a major requirement of the Admin VM for the admin modules and tools. Additionally Tugai [Tug17] established an automated TCL image building process within the seL4 CAmkES VM project, being used to build the TCL images within the project.

AppVM: The configuration of the AppVMs were relatively minimal (e.g not requiring a python interpreter). Hence our AppVMs use a Buildroot-based Linux image, being already supported in the CAmkES VM project. Buildroot is a tool that allows developers generate embedded Linux systems. This compiled a small Linux image suitable for the MVQ build.

6.2 seL4store

The seL4store is a system-wide database equivalent to the xenstore. The implementation consists of three distinct areas:

6.2.1 seL4store CAmkES component

The seL4store component is an active component that implements an RPC interface. The component implements 5 main functions, being write,read, set key permissions and set read watch. The key-values in the store are overlayed with access control rights. When writing a key to the store, the permissions of the key default to read and write access to the writer domain. The writer can then extend those permissions to other domains (through the set key permissions interface). A domain does not have permission to write a sub-key value if it doesn't have permission to write in the parent key. The use of these permissions are illustrated in Figure 6.1.

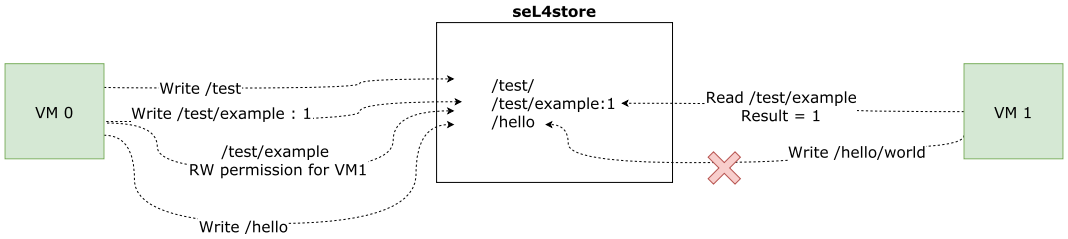

Figure 6.1: seL4store Access Control

Lastly, a domain can register a watch on a specific key. This means a domain can register to receive a notification whenever a specific key is written or changed. If the waiting domain has read permissions on the key, then the domains will receive a notification. A hashtable implementation is used to store the watch requests. When a key write occurs, the hashtable is checked to see if a domain is waiting on the specific key. The component will notify the VM domain by emitting a CAmkES event (seL4_Signal) to the corresponding VMM component.

6.2.2 Cross-VM RPCs

Cross-VM RPCs is an extension to the Cross-VM Connector mechanisms. In addition to the dataports and events, the Cross-VM RPC mechanism exports an RPC interface to guest Linux processes, allowing them to communicate with CAmkES components. This was implemented to simplify interaction with the RPC connections to the seL4store, where the seL4store library API implementation involves invoking the CrossVM RPC mechanisms. The Cross-VM RPC mechanism is implemented through a combination of a Linux kernel driver, VMM hypercall handlers and syscall library wrappers. The system flow of a Cross VM RPC call is demonstrated in Figure 6.2.

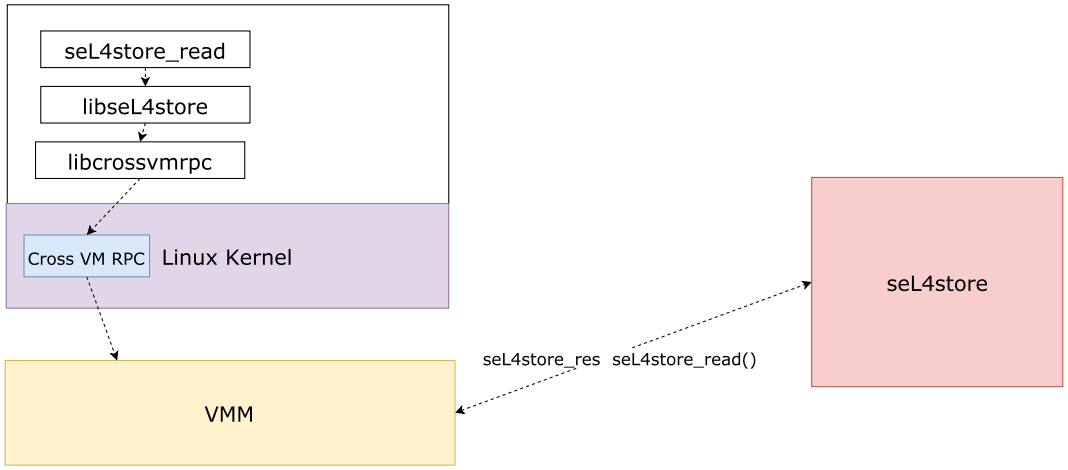

Figure 6.2: seL4store using Cross VM RPCs

The implementation of Cross-VM RPCs is complicated by the fact the RPC function definition needs to be understood by the Linux process and the VMM. Hence a second layer of marshalling occurs (in addition to the VMM to component RPC), where the syscall library wrapper marshalls the RPC data

and the VMM handler identifies the RPC, appropriately unmarshalls the data and invokes the CAmkES RPC connector. To achieve this the function definitions were shared between the VMM and guest Linux OS through a common header file. The marshalling and unmarshalling functions were automatically generated at compile time through C Preproccesor macros. The marshalling process is as follows:

- The cross-vm rpc library packs the arguments into an allocated buffer which then performs an ioctl to the RPC kernel driver
- The driver copies the user memory buffer to kernel space and then hypercalls into the VMM. The hypercall arguments include the guest physical address of the parameters buffer, a guest physical address for a return value buffer, and an identification number of the RPC.
- The VMM identifies the RPC and unpacks the memory according to the parameter definition in the shared header file.
- On return, the VMM populates a return buffer allocated by the Linux process with RPC return data.

6.2.3 Library API

The Cross-VM RPC library generates methods for the RPCs defined in a shared header file. The seL4store library simply wraps around the generated RPC functions to implement the read, write, set key permissions and set watch functions. The seL4store library header can be seen in Appendix 9.

6.3 Event Channels

Event channels provide an asynchronous signalling mechanism between VMs. Despite being in similar nature to the consume and emit Cross-VM connectors, event channels provide a more dynamic interface with which VMs can open and close various event ports between VMs. This means that a VM can have multiple event ports open with another VM but can selectively listen, block and poll on individual ports. This being particularly useful if a VM hosts multiple vchan connections. Similar to seL4store, the implementation of event channels is achieved through several distinct areas, these being through a library syscall interface, kernel driver, VMM handler and CAmkES component.

6.3.1 Library API

The syscall API is implemented to match the Xen evtchn implementation. The key methods to interact with event channels from a Linux process is summarised in Listing 6.1. The event channel implementation is heavily based around the use of a Linux file descriptor. The library implementation involves opening a file descriptor (fd) to the event channel driver, being a Linux character device (exported as "/dev/evtchn"). With the opened fd, a user will be able to bind event channel ports to the specific descriptor enabling them to interact with the event channel through the various API methods.

```
int seL4evtchn_open(int *fd);
int seL4evtchn_close(int fd);
int seL4evtchn_bind_interdomain(int fd, uint32_t domid, uint32_t remote_port);
int seL4evtchn_bind_unbound_port(int fd, uint32_t domid);
int seL4evtchn_notify(int fd, int port);
```

```
int seL4evtchn_unbind(int fd, int port);
int seL4evtchn_unmask(int fd, evtchn_port_t port);
int seL4evtchn_pending(int fd);
```

Listing 6.1: Event Channels API

Important methods worth to elaborate upon include:

evtchn_bind_unbound_port: Allocate a new event channel port to be used with a specific domain ('domid'). The returned event channel (e.g. 2) will only permit the allocator and destination domain to bind to it. The library performs an ioctl syscall on the opened fd.

evtchn_bind_interdomain: Bind the passed fd to the event channel port ('remote_port'). Only the domain the allocated the channel or it the channel was assigned to can bind to the event channel. The library performs an ioctl syscall on the opened fd.

evtchn_pending: With a bound fd, the library can check (poll) if any events have occurred. Note that multiple event channels can be binded to the one fd. Invoking evtchn_pending will return a port that an event occurred. Multiple pending calls need to be made to drain the fd if there are multiple pending events. The events are registered/returned in FIFO (first-in-first-out) order. The library performs an read syscall on the opened fd.

evtchn_unmask: With a binded fd, the user can unmask an event channel on an fd, making it available to receive events on the given channel. Whenever an event occurs on a specific, subsequent events on the same port are disabled. Calling unmask re-opens the port for future signals.

6.3.2 Kernel Driver

The event channel driver is based on the Xen evtchn implementation, in order to match the asynchronous behaviour the vchan library expects. As mentioned prior, the event channel kernel driver is exported as kernel character device (/dev/evtchn). The event channel driver facilitates three main functions, handling library syscalls, handling event channel interrupts from the VMM and performing hypercalls.

Syscall Interface

Through opening an fd, the driver is available to handle various syscalls, being ioctl, read, write, open and close (release). Important for handling these syscall methods enables the kernel to cover important operations regarding the initialisation, binding of and closing of event channels. When opening an event channel, the private data field of the Linux file descriptor struct (struct file *filp) is populated with a event channel specific data-structure. The data-structure keeps track of all the event channels binded to the fd, a notification ring (for tracking incoming events) and a wait queue for waiting processes (when no events have occurred).

When a fd managed by the kernel driver is released, the driver is able to handle the necessary cleaning up of the fd specific data structures and also the unbinding of the event channel port, enabling other domains to recycle the port.

Interrupt Handling

When an event occurs, the VMM injects an interrupt (pre-defined IRQ number) into the guest OS. In addition, the VMM will populate a shared memory region (only between the driver and VMM) with the incoming event channel port. I developed an IRQ handler within the driver to inspect the shared memory region and update the appropriate event channel, corresponding to a specific data-structure. The specific data structure maintains a global state of all fd-evtchn bindings, the interrupt path will go through each fd binding and populate a ring buffer associated with the fd for the incoming event channel port.The mentioned ring buffer is a standard producer-consumer FIFO queue, specific to each fd. The interrupt being the producer and a process calling evtchn_pending being the consumer. If any process was waiting on the event channel port at the time of the interrupt, it will also be resumed.

Hypercall Interface

To allocate or free an event channel port and/or emit an event, the driver will hypercall into the VMM. Upon initialisation, our driver allocates a shared memory region, and passes the guest OS physical address of the shared region to the VMM (through a hypercall). This allowing the VMM to map in the shared memory region into its own virtual address space. Subsequent hypercalls from driver to VMM populate the shared memory region as a means of passing the event channel command and its associated arguments.

6.3.3 VMM Handler

The VMM Init component was extended to contain a handler for event channel related VMM calls. The interface captures the VM exits invoked by the event channel driver and then subsequently uses the shared memory region to decode the specific event channel request and its arguments. The VMM component is linked to the CAmkES event channel component through RPC connectors. The VMM component performs an RPC to the CAmkES component to carry out event channel creation, destruction and event emits.

In addition the VMM handler sets up a callback for incoming CAmkES events (seL4_Signal). The callback forwards the event to the event channel driver by generating an interrupt (pre-defined IRQ value) into the guest OS.

6.3.4 Event Channel CAmkES Component

The CAmkES event channel component is an active component, implementing the RPC methods the VMM Init component interfaces with. The component maintains all global state regarding the set of allocated event channel ports, this including information regarding what domains the channel is binded between.

In addition, the component has an event connector to all VMM components in the system, allowing the component to generate CAmkES events (seL4_Signal). This is used to emit an event on a port. A VMM will thus go through the event channel component to notify another VMM. When going through the component to emit an event, the process performs a lookup of the event channel, translating the port into a destination VM. The event channel will then populate a pre-allocated shared memory region between itself and the destination VMM with a bitmask and emit the CAmkES event. The bitmask has the bit that corresponds to the event port set (e.g. 3rd bit translates to port 3).

An immediate concern of this solution is the introduced inefficiencies from the extra context switch. The reason behind this design is due to the centralised state of the event channel component and the static bindings of event connectors in the CAmkES configuration.

An example flow of an event channel being used in the system is illustrated in Figure 6.3. In our example, event channel port 3 is bound between our two VMs. Using a bound file descriptor, a VM can emit an event on port 3, travelling through the event channel driver and CAmkES component. The CAmkES component then generates an seL4_Signal to the destination VMM. The destination VMM generates the appropriate interrupt signal to the kernel driver. The driver populates a notification ring buffer of an fd that is bound to the event channel port 3. The event is then returned to the destination user process when it next reads from the file descriptor.

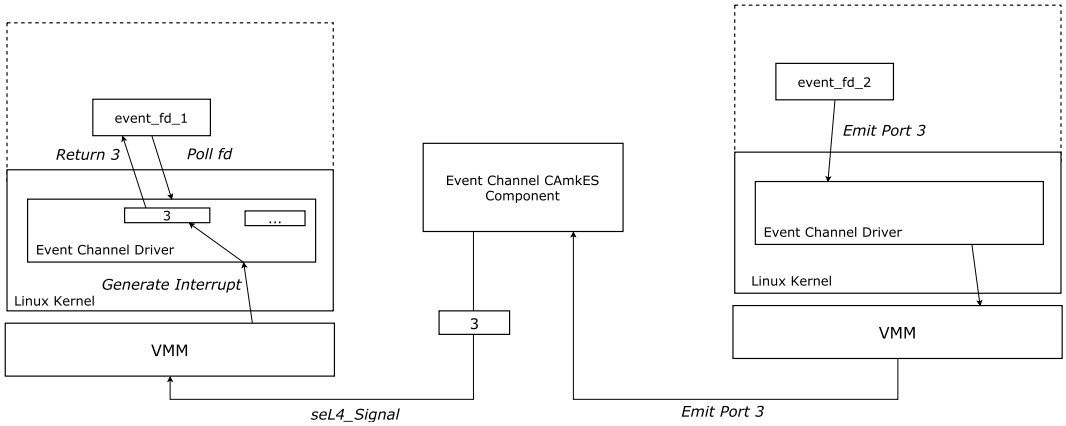

Figure 6.3: Event Channel Flow

6.3.5 Future Work & Improvements

The event channel component could be significantly improved when moving to a more dynamic system. Improvements would involve dynamically allocating notification endpoints between VMMs on the creation of an event channel, foregoing the extra context switch on event emits. To make a more reasonable evalution compared to a Xen implementation (in Chapter 7), we will see a variant of this system that added direct event connectors between each VMM Init component, avoiding the need to passthrough the CAmkES component for emitting events. This however breaks the centralised consistent view the CAmkES component holds on allocated event channels, allowing VMM components to notify other VMMs on ports it hasn't allocated.

Additionally, optimisations could be made between the VMM handler and Linux driver interface. The VMM will generate an interrupt per event channel emit. Improvements such as batching emits would avoid the need to continually switch between the VMM and guest OS. I was however unable to investigate this optimisation during the timeframe of this thesis.

6.4 Share-Allocator

The share allocator is an equivalent implementation to grant tables in Xen. The share-allocator provides a coordinated mechanism for VMs to dynamically allocate shared regions of memory. The approach

involves defining a large pool of memory (dataport), shared between all VM components. The share-allocator CAmkES component provides an interface to manage the shared dataport region, allowing VMs to allocate sections of contiguous memory. The intention behind the share-allocator was to provide a mechanism for our vchan library to allocate buffers of varying sizes to communicate over. Similar to our seL4store and event channels implementation, the implementation of the share-allocator is achieved through several distinct areas, these being through a library syscall interface, kernel driver, VMM handler and CAmkES component.

It is important to note, our implementation doesn't accurately emulate the behaviour of Xen grant tables, rather its behaviour is more unique due to the static nature of the environment it operates. There is opportunity to rework this implementation to better suit later iterations of Qubes on seL4. We significantly base the share-allocator off the existing Cross-VM dataport connector implementation. We make modifications that allow for mapping separate segments of the shared memory space at a time (as opposed to the whole dataport region). We also layer permissions over the shared dataport region, preventing processes from mapping regions it doesn't have permission to access.

6.4.1 CAmkES component

The CAmkES component is responsible for managing the shared data region. We manage the region at 4K page boundaries, effectively creating a one-dimensional page table. Each page entry tracks the domain that allocated the page, and in addition other domains that have permission to read and write to the page.

The component is active, implementing an RPC connector for other VMMs to interact with. The component involves supporting allocation of shared pages, freeing pages, setting permissions on pages and checking if we have access rights to a given set of pages.

6.4.2 Kernel Driver

We require the kernel driver to manage the shared memory region and map the shared region into the processes address space. Upon initialisation, the driver allocates a region of kernel memory that intends to be backed by the physical frames of the shared region. When a process needs to map the shared memory into its virtual address space, it invokes an mmap syscall to create a new mapping. Our driver has an mmap syscall handler that in turn backs the processes mmap memory segment at the offset which aligns with the memory region allocated by the kernel upon initialisation. Whether the mmap region is actually backed by the physical frames of the shared memory segment depends if the VM domain has been granted access to the corresponding pages.We verify this by hypercalling into the VMM who in turn performs an RPC to the CAmkES component that verifies the rights of the domain. If the rights are valid the VMM handler will back the appropriate pages in the kernels memory region with the frames that correspond to the shared dataport memory region. The relationship between the kernel memory buffer and shared memory region is illustrated in Figure 6.4

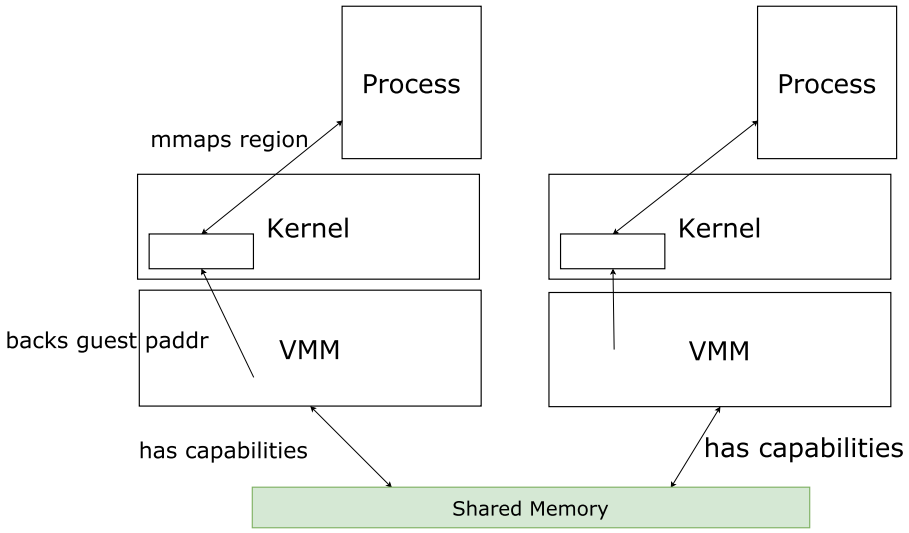

Figure 6.4: Share Allocator Memory Structure

6.4.3 VMM handler

Relying on the VMM handler to perform the mapping elevates the MVQ TCB to include the VMM component. We generally however don't assume that the VMM is ever malicious, rather the guest OS can be. Without the VMM enforcing permission checks over the shared dataport region we allow a malicious guest OS to gain access to the entire shared region, which it can use as a channel to listen and tamper with other inter-domain communication.

To allow the VMM to dynamically map in the frames we use the 'seL4SharedDataWithCaps' template in CAmkES. This is a dataport connector which exposes the frame capabilities to the VMM, enabling us to map pages into the guest OS vspace.

6.4.4 Library API

To interact with the various allocation mechanisms we implement a library which the Linux user process can call. A process can allocate shared page, free shared pages and also choose to share the pages it has allocated with other domains. The API is shown in Listing 6.2. Our library involves the use of the Cross-VM RPC driver to invoke the RPC allocation methods implemented in the CAmkES component and the share-allocator kernel driver to mmap the shared region into the process virtual address space. Note the share-allocator driver is necessary to get the VMM to back the mmap region with the physical frames.

```
int alloc_share_init(int *fd);
void alloc_share_close(int fd);
int alloc_share_pages(int share_malloc_fd, size_t *offset, size_t size);
void alloc_free_share_pages(int share_malloc_fd, size_t offset, size_t size);
int alloc_share(int share_malloc_fd, int to_domain, size_t offset, size_t size);
```

Listing 6.2: Share-Allocator API

6.4.5 Future Work & Improvements

The behaviour we discussed revolves around having a static pre-defined shared memory pool. Improvements we could make for future iterations of the MVQ, when moving out of a CAmkES based system, would involve the VMM allocating untyped memory into frames on demand. Furthermore the implementation we explored doesn't apply fine-grained permissions. It is either R/W or no permissions at all. Extending the implementation to support this should be trivial however it was not explored in the timeframe of this thesis.

6.5 Porting VChans & Qrexec

Given our substitutes for xenstore, evtchns and gnttabs, porting the Xen vchan library to seL4 involves modifying the library calls with our own. A majority of the logic was left unchanged, however modifications had to made to the initialisation of the shared memory buffer due to differences with our API against gnttab. In addition to using the Xen vchan library, the Qubes project also implements a minimal vchan library wrapper around the Xen vchan library. This has an identical API to the Xen vchan library where most functions directly call through to the Xen implementation. Modifications are added to server and client initialisation methods where a client sets a store read watch in the case the client attempts to establish a connection before the server has started. We require small modifications to make read watch usage consistent with our libraries. Lastly modifications were made to log printing macro usage and C include paths to match our build system.

Qrexec requires minimal changes. Modifications made reflect different include paths to match our build system.

The changes involved for porting vchans and qrexec is show in Table 6.1.

Table 6.1: Porting vchan and qrexec: LoC Differences

Program/Library	**LoC Xen**	**LoC seL4**	**Same**	**Modified**	**Added**	**Removed**
libvchan	616	762	395	186	181	35
qubes libvchan	262	157	86	61	10	51
Qrexec Library	1098	1098	1084	14	0	0
Qrexec Programs	2728	2727	2698	25	4	5

6.6 vm-manager

The vm-manager in our MVQ is a simple management toolstack, similar in nature to libxl. We use the vm-manager API to implement our libvirt python wrapper. The vm-manager is implemented in the same manner as prior components, involving a syscall library, vm-manager kernel driver and CAmkES component. Through a combination of these components, the functionality we implement includes:

Get Domain ID: This returns the domain ID the calling process exists in. The VM's are statically assigned domain id attributes in the CAmkES specification. A special template connector was implemented to export this variable to be accessed by the VMM component.

Create New Domain: This allows the user to 'create' a new VM. In our static MVQ system, the VMs are pooled. Our CAmkES vm-manager component maintains knowledge of the free and used VMs. On invocation, our vm-manager component finds a free VM and returns its domain-id. VMs are created with a domain name and a UUID (universally unique identifier) value.

Lookup Domain Name: Returns the domain id that corresponds with a given name. The vm-manager component maintains state of all created VMs and their given names and UUID values.

Lookup Domain UUID: Returns the domain id that corresponds with a given UUID value.

Free Domain: Free's the domain id, allowing it to be recycled for future domain creation calls.

6.7 Porting Qubes Admin Modules & qvm-tools

The porting effort surrounding the qubes admin modules and qvm-tools in our Xen MVQ was able to directly translate to our seL4 MVQ implementation. The modifications made during the development of our Xen MVQ involved removing code relating to the use of the Qubes GUI and various unsupported vm classes. The modification summary can be seen in Table 6.2, measured with the use of the CLOC (Count Lines of Code) tool [Dan15].

Table 6.2: Porting admin modules and qvm-tools: LoC Differences

Program/Library	LoC Xen	LoC seL4	Same	Modified	Added	Removed
Admin Modules	2758	1736	1331	155	250	1272
qvm-tools	1292	1019	935	46	38	311

To integrate the qubes admin modules and qvm-tools into our seL4 MVQ, we make modifications to the TCL image generation script to include the new python libraries and tools into the rootfile system image.

6.8 Linux File Server

A slight divergence from focusing on the development of the MVQ was the need to implement a Linux fileserver. The CAmkES VM project involved the implementation of a file server that ran as a separate active component. The file server managed a cpio archive, compiled into the boot image, compacting the Linux kernel and file system images used by the VMMs. The file server implements a POSIX based interface with which a VMM connects to, through an RPC connector, to open and read the images required to boot the guest OS.

Problematic to compiling the Linux file system and kernel images into the seL4 boot image was that it consumed a large amount of memory, limiting the amount of memory we could give to a guest OS once booted. Hence we introduce a Linux based file server with passthrough access to USB. This allowing VMMs to read their images off disk. This is achieved by extending an implementation of the file server RPC interface to a VMM Init component. In addition, the file server OS starts a file server kernel driver upon boot. When the VMM recieves an I/O request, we generate a pre-defined interrupt into the guest OS, passing the I/O request arguments through a shared memory buffer between the VMM and driver. The driver handles the interrupt, parses the arguments and performs the I/O request. The returned file I/O data is placed into the shared buffer between the VMM and driver, subsequently processed by the

VMM and transferred to another shared buffer between the server VMM and client VMM. A general overview of this architecture is illustrated in Figure 6.5.

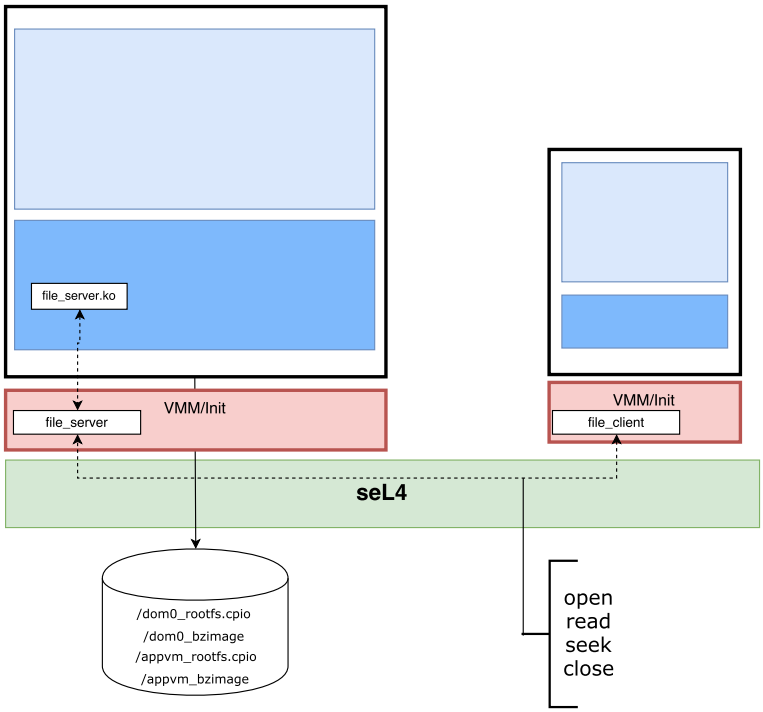

Figure 6.5: Linux Fileserver

6.8.1 Future Work & Improvements

Our fileserver implementation could be further optimized. For simplicity, file I/O requests are handled in a synchronous fashion, where the kernel driver can only handle one I/O request at a time. This could be improved by implementing mechanisms to queue I/O requests, avoiding the cost of blocking and continually entering and exiting the VM.

A useful extension to this framework would be implementing paravirtualized block device support. This would allow the file server VM to export virtual block device interfaces to other VMs, which they could boot and read directly from. The implementation would require substantial engineering however the current file server implementation could serve as a base to build off.

Chapter 7: Evaluation

7.1 Benchmark Configuration

7.1.1 Hardware Setup

Benchmarking experiments were carried out to investigate the performance difference between Xen and seL4 for event channels and vchans. The experiments were performed on the same x86_64 platform. The test machine details are shown in Table 7.1.

Spec	**Value**
Arch	x86
ISA	x86_64
SoC	haswell
CPU	i7-4770
Cores	4
RAM	16GB
Max CLK	3.4Ghz

Table 7.1: Testing Hardware

7.1.2 System Setup and Testing

The systems we will evaluate and their configurations are described below:

seL4 MVQ: The seL4 MVQ is the system we implemented. This being a CAmkES based system, built with all the discussed components (event channels, share-allocator etc). The kernel is compiled with with user debugging and kernel prints disabled and VTx support enabled. The MVQ has been setup to only start two VMs, the Admin VM (32-bit TCL OS) and an AppVM (32-bit buildroot OS). The event channel implementation was modified to allow VMM components to directly notify other VMMs rather than going through the event channel component. We do this to better represent a dynamic configuration that is comparable to Xen.

Xen: We use a x86 64-bit build of Xen 4.9.0 release (based on the "RELEASE-4.9.0" tag from Xen git project [Pro17b]). Domain 0 on Xen hosts a Ubuntu 16.04 LTS 64-bit Linux OS (PV). A 64-bit Fedora OS (PV) is started as a domU to carry out the test.

Both configurations are given the same benchmarking programs. The event channel and vchan APIs between seL4 and Xen are identical with the exception of the method names e.g. libseL4vchan_read vs libxenvchan_read. In addition we disable network activity on both platforms in order to minimize noise in the system.

We perform our measurements in the guest OSes, using the 'clock_gettime' library to retrieve monotonically increasing clock values, independent of system time.

7.2 Benchmarking Event Channels

7.2.1 Benchmarking Program

To benchmark event channels, a simple ping-pong application was implemented. A server sends a client domain a notification which the client immediately responds to by sending an event back. The server measures the round-trip time (RTT) of the process in microseconds. The benchmark performs 50 ping-pong rounds (plus 5 rounds warmup) in a given test. We measure the average RTT of a ping-pong round. We run 10 tests and present the mean and standard deviation across those 10 tests. Changing the iteration and test values gave litte difference, with minimal variance between the measured times.

7.2.2 Benchmarking Results

The results of the ping-pong benchmark can be seen in Figure 7.1.

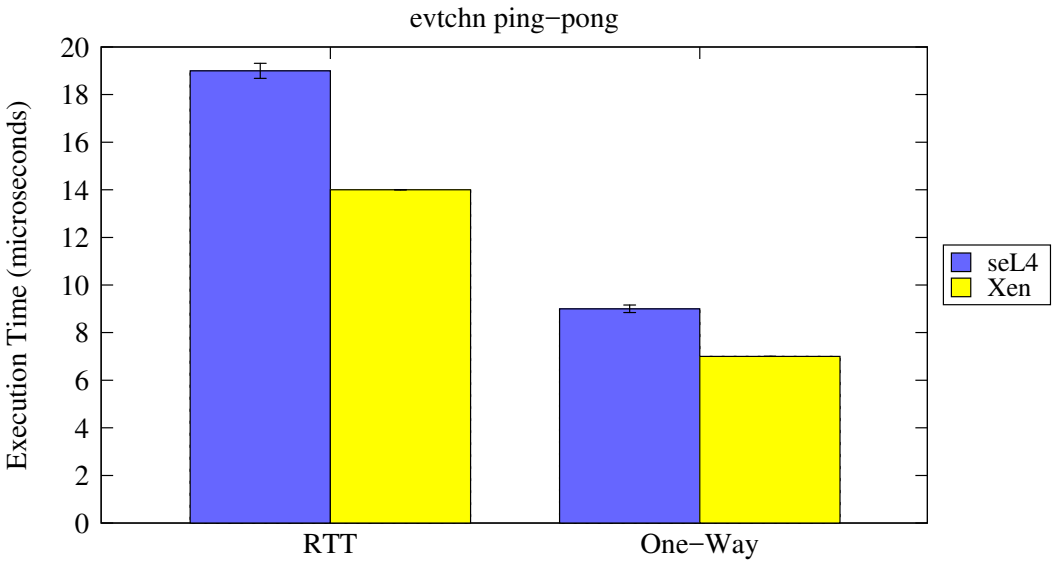

Figure 7.1: Round Trip Latency of Event Channels

The benchmark shows an average increase of 5 microseconds in a RTT measurement over the Xen event channel implementation. The seL4 one-way latency being roughly 9.5 microseconds. We assume one way latency is half the round trip time as the path between the VMMs are symetrical when emitting an event. Definitive reasons as to why there is extra latency was unable to be identified when benchmarking. The extra latency may be possibly introduced by going through the Init component when sending and recieving an event. The event path being from the guest OS to its Init component to the destination Init component into the destination guest OS.

7.3 Benchmarking VChans

7.3.1 Benchmarking Program

To benchmark vchans, we extend our ping-pong application to write and recieve increasing message buffer sizes between a server and client. We establish a ring buffer of a fixed size to send the messages and measure the overall, accumulative time it took to complete the read and write operations. VChans will internally segment write buffers into multiple messages if the message size is greater than the ring buffer size. We expect to see increasing reading and writing times for increasing message sizes.

Similar to the event channel ping-pong benchmark, for every message, we perform 50 ping-pong rounds (plus 5 rounds warmup) in a given test. The server writes a message buffer (containing random data) which is in turn read by the client and written back to the server. We perform benchmarking runs prior to measurement to validate vchans are working correctly and the server is recieving the correct data back from the client. We measure the average time it took to write the entire message and read the entire message on the server end .

7.3.2 Benchmarking Results

The resulting read and write performance of the vchan benchmark can be seen in Figures 7.2 and 7.3.

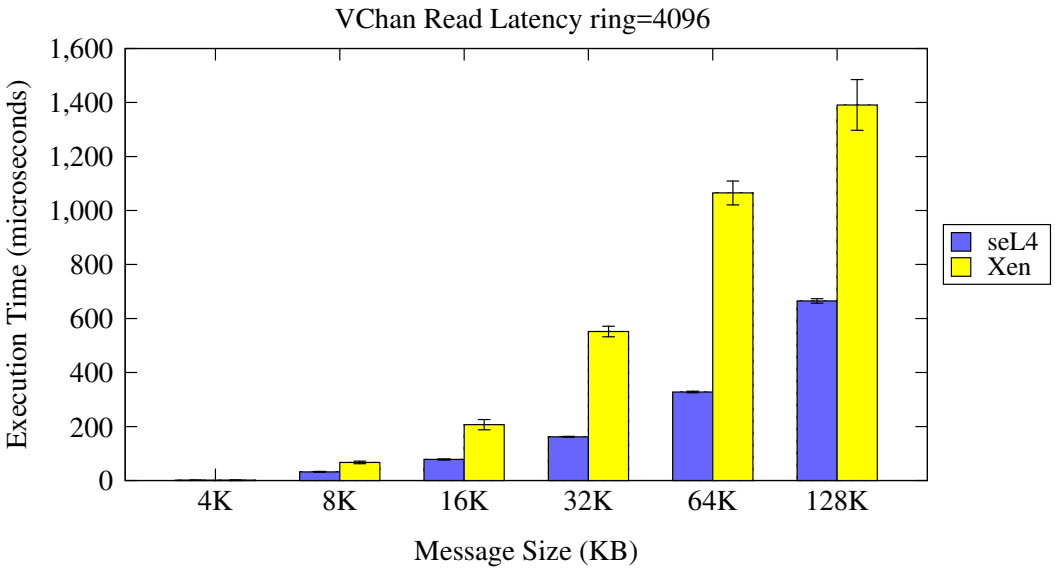

Figure 7.2: VChan Read Benchmark

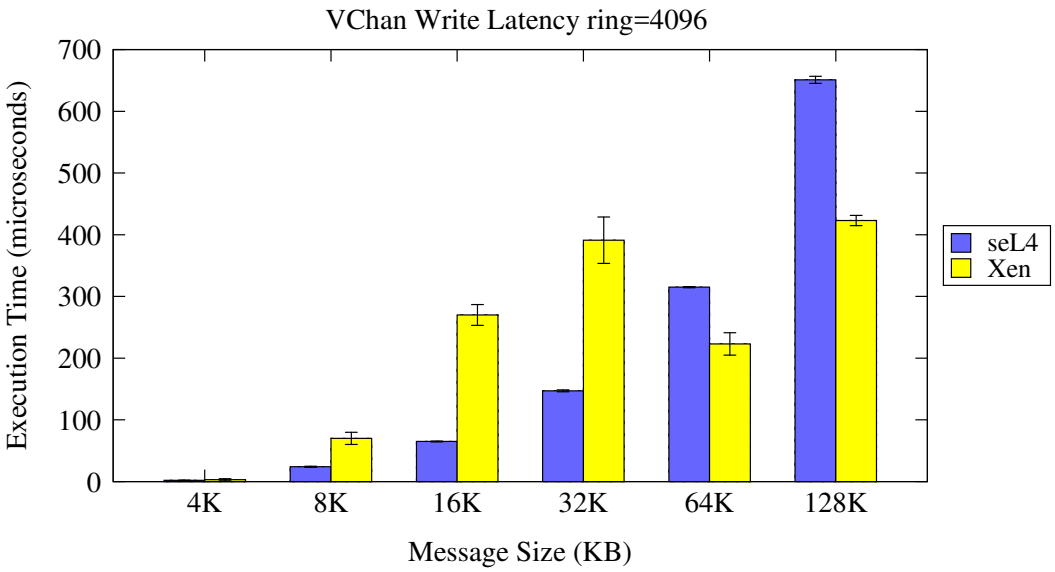

Figure 7.3: VChan Write Benchmark

The read and write performance for vchan on seL4 follows the trend of increasing read and write times for increasing message sizes. This is also the same for Xen with regards to vchan reads. Write performance however is not consistent on Xen. Xen has less latency when writing larger buffers, whilst also having signicantly higher latencies for smaller workloads. For 128K message sizes, seL4 shows an average increase of 220 microseconds. The experiment is consistent and repeatable whereby Xens latency consistently drops when handling >=64K messages. The reason behind these results could not be determined. Xen could possibly have grant table optimizations for larger mappings, however this could not be confirmed.

Chapter 8: Roadmap

Qubes is large framework and there are a lot more items to implement before being able to support a full Qubes port. We will discuss the future work here, laying out a roadmap to bring us closer to a full Qubes port.

Dynamic VMs on seL4: As discussed in earlier sections, the next step would involve moving the MVQ build out of CAmkES environment. The items implemented rely on statically defined connections and kernel objects. A first step would involve breaking away from the static definitions, providing the ability to create the necessary kernel objects out of untyped memory on demand. The ability to dynamically start and shutdown VMs would be the next necessary iteration.

64-bit VMs: Identified as an earlier road-block, Qubes requires 64-bit VM support whilst we only have 32-bit VM support on seL4. This limitation lead us to take a bottom-up approach, taking individual items in the build system and building them as 32-bit targets. Supporting 64-bit VMs would allow the seL4 implementation to better merge with the Qubes architecture and build system.

Paravirtualised Drivers: Split driver models are used extensively in Qubes on Xen as a mechanism to export virtual interfaces to hardware in other domains. This being the key mechanism behind service vm's. Further work can look into developing more split driver models for various hardware devices e.g. block devices, ethernet card. Additional approaches can look into using unikernels to provide virtual device interfaces, reducing the memory footprint in comparison to using a Linux kernel for hardware access.

Libvirt Driver: A proper libvirt driver port is the next major requirement in fulfilling Qubes' hypervisor abstraction layer. This would involve extending the libvirt driver framework to support seL4's virtualisation API e.g. vm-manager.

Secure GUI: A significant subsystem omitted from the developed MVQ were the secure GUI daemons. This being necessary to create a streamlined desktop experience. Modifications need to be made to the guest images such that they contain the appropriate X-org libraries and drivers as well as a port of the Qubes GUI infrastructure. Further work would also involve developing graphics passthrough in order to render the desktop on a screen.

In summary, there are multiple tasks and opportunities that would be beneficial to support a future full port of Qubes on seL4.

Chapter 9: Conclusion

The goal of this thesis was to assess the viability of seL4 as a virtualisation platform for Qubes. To reach this goal we sought to port a subset of the Qubes architecture to prove we could support the Qubes framework on seL4. Our approach involved targeting lower level toolstacks and virtualisation API's that existed in the Xen ecosystem, providing a means to emulate behaviours the Qubes framework expects on a virtualisation platform.

Through this thesis, we explored background information regarding virtualisation and mechanisms to reduce the TCB of monolithic operating systems. We explored the Qubes architecture and identified the key sub-systems that power the system. Through the related work, we further investigated alternative solutions in the same space as Qubes. We presented a design of an approach to porting the Qubes architecture to seL4, an implementation to execute our desired design and improvements that can be made to improve the future iterations of Qubes on seL4.

Taking our approach, we've demonstrated that we can support the Qubes framework. This work laying a basic foundation to further develop Qubes upon as we move to a more complete port of Qubes to seL4. Importantly, through unpacking the Qubes architecture we've identified the requirement and roadblocks of the Qubes framework and where development needs to focus to better support a more complete port of Qubes on seL4. The next step would involve breaking the MVQ implementation away from the static CAmkES base, moving to a more dynamic system.

Lastly we evaluated the basic performance of Qubes and Xen components as we implemented them on seL4. This offering a basic framework and starting point which can be further analysed, improved and expanded on in future iterations.

Bibliography

[BBM+12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, USA, October 2012. USENIX.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating System Review*, 37(5):164–177, October 2003.

[Bro17] Bromium. Our technology. https://www.bromium.com/platform/our-technology.html, 2017. Accessed: 2017-05-18.

[CNZ+11] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, New York, NY, USA, October 2011. ACM.

[Dan11] Alan Dang. Qubes OS: An operating system designed for security. http://www.tomshardware.com/reviews/qubes-os-joanna-rutkowska-windows,3009.html, August 2011. Accessed: 2017-05-18.

[Dan15] Al Danial. Cloc. http://cloc.sourceforge.net/, 2015. Accessed: 2017-10-24.

[Fes16] Norman Feske. *GENODE: Operating System Framework 16.05*. Genode Labs, Dresden, 2016.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, October 2009. ACM.

[KH17] Greg Kroah-Hartman. Linux 4.12 is big, really big, like bigger than you thought big. https://plus.google.com/photos/photo/111049168280159033135/6438923160458543554?icm=false&iso=true, 2017. Accessed: 2017-10-24.

[Kuz16] Ihor Kuz. Virtualisation on seL4. https://sel4.systems/Community/Devdays/Workshop2016/ihor-sel4ws2016.pdf, 2016. Accessed: 2017-05-18.

[Lab16] Genode Labs. Release notes for the Genode OS Framework 16.08. https://www.genode.org/documentation/release-notes/16.08, 2016. Accessed: 2017-05-18.

[Lab17a] Genode Labs. Future challenges of the Genode project. https://genode.org/about/challenges, 2017. Accessed: 2017-05-18.

[Lab17b] Genode Labs. Release notes for the Genode OS Framework 17.08. https://genode.org/documentation/release-notes/17.08, 2017. Accessed: 2017-10-19.

[Mui14] A.J. Muir. Taste of research: Qubes on seL4. Technical report, NICTA, 2014.

[PJ10] Matt Piotrowski and Anthony D. Joseph. Virtics: A system for privilege separation of legacy desktop applications. Technical Report UCB/EECS-2010-70, EECS Department, University of California, Berkeley, May 2010.

[Pro11] Xen Project. http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=README;h=c9bf699be6bb938e3b263fbad9db1d5cdc8a881e;hb=834f076e77c4a88df2f87de276debb12048f7eca, 2011. Accessed: 2017-10-24.

[Pro13] Xen Project. Grant table. https://wiki.xen.org/wiki/Grant_Table, 2013. Accessed: 2017-10-24.

[Pro14] Xen Project. Event channel internals. https://wiki.xen.org/wiki/Event_Channel_Internals, 2014. Accessed: 2017-10-24.

[Pro15] Xen Project. Xenstore. https://wiki.xen.org/wiki/XenStore, 2015. Accessed: 2017-10-24.

[Pro17a] Qubes OS Project. Qubes security bulletins. https://www.qubes-os.org/security/bulletins/, May 2017. Accessed: 2017-05-18.

[Pro17b] Xen Project. http://xenbits.xen.org/gitweb/?p=xen.git;a=commit;h=c30bf55594a53fae8aae08aabf16fc192faad7da, 2017. Accessed: 2017-10-24.

[Pro17c] Xen Project. Paravirtualization (pv). https://wiki.xen.org/wiki/Paravirtualization_(PV), 2017. Accessed: 2017-10-24.

[Pro17d] Xen Project. Xen project software overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview, 2017. Accessed: 2017-10-24.

[Rut13] Joanna Rutkowska. Introducing Qubes Odyssey Framework. https://blog.invisiblethings.org/2013/03/21/introducing-qubes-odyssey-framework.html, 2013. Accessed: 2017-05-18.

[Rut17] Joanna Rutkowska. Qubes Os 4.0-rc1 has been released. https://www.qubes-os.org/news/2017/07/31/qubes-40-rc1/, 2017. Accessed: 2017-10-24.

[RW10] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS architecture. Technical report, January 2010.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[Tug17] G. Tugai. Qubes on seL4. Technical report, Data61, 2017.

[XZh16] XZhang. Xen vchan port. https://www.cs.uic.edu/~xzhang/vchan/, 2016. Accessed: 2017-10-24.

Appendix 1 - Library Interfaces

```
#ifndef __EVTCHN_H
#define __EVTCHN_H

#include <inttypes.h>
#include "evtchn_types.h"

int seL4evtchn_open(int *fd);
int seL4evtchn_close(int fd);
int seL4evtchn_bind_interdomain(int fd, uint32_t domid, uint32_t remote_port);
int seL4evtchn_bind_unbound_port(int fd, uint32_t domid);
int seL4evtchn_notify(int fd, int port);
int seL4evtchn_unbind(int fd, int port);
int seL4evtchn_unmask(int fd, evtchn_port_t port);
int seL4evtchn_pending(int fd);

#endif
```

Listing 1: evtchn Interface

```
#ifndef __SEL4_STORE_H
#define __SEL4_STORE_H

#include <stdint.h>

#define SEL4STORE_PERM_NONE 0
#define SEL4STORE_PERM_READ 1
#define SEL4STORE_PERM_WRITE 1
#define SEL4STORE_PERM_OWNER 1

#define SEL4STORE_KEY_EXISTS 9

int seL4store_write(int seL4store_fd, char * key, char * value, uint32_t value_length);

int seL4store_read(int seL4store_fd, char * key, uint32_t length, char **res);

int seL4store_set_key_perm(int seL4store_fd, char * key, int read, int write, int owner,
    unsigned int to_domain);

int seL4store_read_watch(int seL4store_fd, char * key);

int seL4store_init(int *fd);

void seL4store_close(int fd);

#endif
```

Listing 2: seL4Store Interface

```
#ifndef __VM_MANAGER_H
#define __VM_MANAGER_H

int get_domain_id(int fd);
int create_new_domain(int fd, char *name, char *uuid);
int lookup_domain_name(int fd, char *name, char **uuid);
int lookup_domain_uuid(int fd, char *uuid);
```

```
void free_domain(int fd, int domain_id, char *uuid);

#endif
```

Listing 3: vm-manager Interface

```
#ifndef __SHARE_MALLOC_H__
#define __SHARE_MALLOC_H__

int alloc_share_init(int *fd);
void alloc_share_close(int fd);
int alloc_share_pages(int share_malloc_fd, size_t *offset, size_t size);
void alloc_free_share_pages(int share_malloc_fd, size_t offset, size_t size);
int share_pages(int share_malloc_fd, int to_domain, size_t offset, size_t size);

#endif /* __SHARE_MALLOC_H__ */
```

Listing 4: Share-Alloc Interface

```
struct libseL4vchan_ring {
        /* Pointer into the shared page. Offsets into buffer. */
        struct ring_shared* shr;
        /* ring data; may be its own shared page(s) depending on order */
        void* buffer;
        /**
         * The size of the ring is (1 << order); offsets wrap around when they
         * exceed this. This copy is required because we can't trust the order
         * in the shared page to remain constant.
         */
        int order;
};

/**
 * struct libseL4vchan: control structure passed to all library calls
 */
struct libseL4vchan {

        /* Pointer to shared ring page */
        struct vchan_interface *ring;

    uint32_t ring_ref;

    /* event channel interface */
        int event_fd;
    int dataport_fd;
    int share_mem_fd;

    uint32_t event_port;
        /* informative flags: are we acting as server? */
        int is_server:1;
        /* true if server remains active when client closes (allows reconnection) */
        int server_persist:1;
        /* true if operations should block instead of returning 0 */
        int blocking:1;
        /* communication rings */
        struct libseL4vchan_ring read, write;
};

/**
 * Set up a vchan, including granting pages
 * @param domain The peer domain that will be connecting
 * @param seL4store_path Base seL4store path for storing ring/event data
 * @param send_min The minimum size (in bytes) of the send ring (left)
 * @param recv_min The minimum size (in bytes) of the receive ring (right)
```

```
 * @return The structure, or NULL in case of an error
 */
struct libseL4vchan *libseL4vchan_server_init(int domain, const char* seL4store_path,
                                              size_t read_min, size_t write_min);
/**
 * Connect to an existing vchan. Note: you can reconnect to an existing vchan
 * safely, however no locking is performed, so you must prevent multiple clients
 * from connecting to a single server.
 *
 * @param domain The peer domain to connect to
 * @param seL4store_path Base seL4store path for storing ring/event data
 * @return The structure, or NULL in case of an error
 */
struct libseL4vchan *libseL4vchan_client_init(int domain, const char* seL4store_path);
/**
 * Close a vchan. This deallocates the vchan and attempts to free its
 * resources. The other side is notified of the close, but can still read any
 * data pending prior to the close.
 */
void libseL4vchan_close(struct libseL4vchan *ctrl);

/**
 * Packet-based receive: always reads exactly $size bytes.
 * @param ctrl The vchan control structure
 * @param data Buffer for data that was read
 * @param size Size of the buffer and amount of data to read
 * @return -1 on error, 0 if nonblocking and insufficient data is available, or $size
 */
int libseL4vchan_recv(struct libseL4vchan *ctrl, void *data, size_t size);
/**
 * Stream-based receive: reads as much data as possible.
 * @param ctrl The vchan control structure
 * @param data Buffer for data that was read
 * @param size Size of the buffer
 * @return -1 on error, otherwise the amount of data read (which may be zero if
 *         the vchan is nonblocking)
 */
int libseL4vchan_read(struct libseL4vchan *ctrl, void *data, size_t size);
/**
 * Packet-based send: send entire buffer if possible
 * @param ctrl The vchan control structure
 * @param data Buffer for data to send
 * @param size Size of the buffer and amount of data to send
 * @return -1 on error, 0 if nonblocking and insufficient space is available, or $size
 */
int libseL4vchan_send(struct libseL4vchan *ctrl, const void *data, size_t size);
/**
 * Stream-based send: send as much data as possible.
 * @param ctrl The vchan control structure
 * @param data Buffer for data to send
 * @param size Size of the buffer
 * @return -1 on error, otherwise the amount of data sent (which may be zero if
 *         the vchan is nonblocking)
 */
int libseL4vchan_write(struct libseL4vchan *ctrl, const void *data, size_t size);
/**
 * Waits for reads or writes to unblock, or for a close
 */
int libseL4vchan_wait(struct libseL4vchan *ctrl);
/**
 * Returns the event file descriptor for this vchan. When this FD is readable,
 * libseL4vchan_wait() will not block, and the state of the vchan has changed since
 * the last invocation of libseL4vchan_wait().
 */
int libseL4vchan_fd_for_select(struct libseL4vchan *ctrl);
/**
```

```
 * Query the state of the vchan shared page:
 *  return 0 when one side has called libseL4vchan_close() or crashed
 *  return 1 when both sides are open
 *  return 2 [server only] when no client has yet connected
 */
int libseL4vchan_is_open(struct libseL4vchan* ctrl);
/** Amount of data ready to read, in bytes */
int libseL4vchan_data_ready(struct libseL4vchan *ctrl);
/** Amount of data it is possible to send without blocking */
int libseL4vchan_buffer_space(struct libseL4vchan *ctrl);
```

Listing 5: VChan Interface

Appendix 2 - CAmkES Specification

```
#include <autoconf.h>
#include <configurations/vm.h>
import <std_connector.camkes>;

#define FILE_SERVER_VM_NUM 2

procedure CrossRPC {
    include "seL4_store_wire.h";
    seL4_store_retmsg_t vm_rpc(in seL4_store_sockmsg_t msg);
}

procedure ShareMallocRPC {
    include "share_malloc_common.h";
    share_request_res_t vm_rpc(in share_request_t request);
}

procedure EventRequestRPC {
    include "evtchn_wire.h";
    evtchn_ret_wire_t vm_rpc(in int cmd, in evtchn_wire_t request);
}

procedure VmManagerRPC {
    include "vm_manager_common.h";
    int vm_rpc(in vm_manager_request_args_t req_args);
}

procedure Lock {
    void lock(void);
    void unlock(void);
}

component Init0 {
    VM_INIT_DEF(ADMIN_VM)

    uses Ethdriver ethdriver;

    consumes seL4EventNotification dom_notify;
    consumes seL4StoreWatchNotification watch_notify;

    uses DomInfoInterface dom_info;
    has mutex cross_vm_event_mutex;
    uses CrossRPC seL4store_request;
    uses ShareMallocRPC share_malloc_request;
    uses EventRequestRPC event_request;
    uses VmManagerRPC vm_manager_request;

    dataport Buf(4096) seL4store_request_buf;
    dataport Buf(1474560) vchan_buf;
    dataport Buf(4096) event_port_status;
    dataport Buf(4096) vm_manager_request_buf;

    has mutex evtchn;
    provides Lock evt;
}

component Init1 {
```

```
    VM_INIT_DEF(USER_VM)

    uses Ethdriver ethdriver;

    consumes seL4EventNotification dom_notify;
    consumes seL4StoreWatchNotification watch_notify;

    uses DomInfoInterface dom_info;
    has mutex cross_vm_event_mutex;

    uses CrossRPC seL4store_request;
    uses ShareMallocRPC share_malloc_request;
    uses EventRequestRPC event_request;
    uses VmManagerRPC vm_manager_request;

    dataport Buf(4096) seL4store_request_buf;
    dataport Buf(1474560) vchan_buf;
    dataport Buf(4096) event_port_status;
    dataport Buf(4096) vm_manager_request_buf;

    has mutex evtchn;
    provides Lock evt;
}


component Init2 {
    VM_INIT_DEF(FILE_SERVER_VM)

    uses Ethdriver ethdriver;

    consumes seL4EventNotification dom_notify;
    consumes seL4StoreWatchNotification watch_notify;

    uses DomInfoInterface dom_info;
    has mutex cross_vm_event_mutex;

    uses CrossRPC seL4store_request;
    uses ShareMallocRPC share_malloc_request;
    uses EventRequestRPC event_request;
    uses VmManagerRPC vm_manager_request;

    dataport Buf(4096) seL4store_request_buf;
    dataport Buf(1474560) vchan_buf;
    dataport Buf(4096) event_port_status;
    dataport Buf(4096) vm_manager_request_buf;

    has mutex evtchn;
    provides Lock evt;
}

component SEL4Store {
    control;
    provides CrossRPC request_in;
    emits seL4StoreWatchNotification watch_notify;

    dataport Buf(4096) seL4store_request_buf0;
    dataport Buf(4096) seL4store_request_buf1;
    dataport Buf(4096) seL4store_request_buf2;

    has mutex store;
}

component ShareMalloc {
    control;
    dataport Buf(1474560) vchan_buf;
    provides ShareMallocRPC share_malloc_request_in;
```

```
    has mutex share;
}

component EventChannel {
    control;
    provides EventRequestRPC event_request;
    emits seL4EventNotification dom_notify;
    dataport Buf(4096) event_port_status;
    has mutex event;

    uses Lock evt_port0;
    uses Lock evt_port1;
    uses Lock evt_port2;
}

component VmManager {
    control;
    provides DomInfoInterface dom_info;
    provides VmManagerRPC manager_request_in;

    dataport Buf(4096) manager_request_buf0;
    dataport Buf(4096) manager_request_buf1;
    dataport Buf(4096) manager_request_buf2;

    has mutex manager;
}

component VM {

    provides Ethdriver ethdriver_interface;

    composition {
        VM_COMPOSITION_DEF()
        VM_PER_VM_COMP_DEF(0, ADMIN_VM, FILE_SERVER_VM_NUM)
        VM_PER_VM_COMP_DEF(1, USER_VM, FILE_SERVER_VM_NUM)
        VM_PER_VM_COMP_DEF(2, FILE_SERVER_VM, FILE_SERVER_VM_NUM)

        component SEL4Store seL4_store;
        component ShareMalloc share_malloc;
        component EventChannel event_channel;
        component VmManager vm_manager;

        /* When vchan is set up, everyone will be able to notify everyone else*/
        /* Need to modify consume and emit init extensions to handle domain notifications*/
        /* Might need to find a way to compact these into one interface/component */
        connection seL4RPCCall event_request(from vm0.event_request, from vm1.event_request,
                                             from vm2.event_request, to event_channel.
                                                 event_request);
        connection seL4DomNotification dom_notify(from event_channel.dom_notify, to vm0.
            dom_notify, to vm1.dom_notify,
                                                  to vm2.dom_notify);
        connection seL4DomNotification watch_notify(from seL4_store.watch_notify, to vm0.
            watch_notify, to vm1.watch_notify,
                                                    to vm2.watch_notify);

        connection seL4RPCCall request_rpc(from vm0.seL4store_request, from vm1.
            seL4store_request,
                                           from vm2.seL4store_request, to seL4_store.
                                               request_in);
        connection seL4SharedDataWithCaps request_rpc_buf0(from seL4_store.
            seL4store_request_buf0,
                                           to vm0.seL4store_request_buf);
        connection seL4SharedDataWithCaps request_rpc_buf1(from seL4_store.
            seL4store_request_buf1,
                                           to vm1.seL4store_request_buf);
        connection seL4SharedDataWithCaps request_rpc_buf2(from seL4_store.
```

```
            seL4store_request_buf2,
                                            to vm2.seL4store_request_buf);

        connection seL4RPCCall request_malloc(from vm0.share_malloc_request, from vm1.
            share_malloc_request,
                                            from vm2.share_malloc_request, to
                                                share_malloc.share_malloc_request_in);
        connection seL4VChanDataWithCaps vchan_buf(from share_malloc.vchan_buf, to vm0.
            vchan_buf, to vm1.vchan_buf,
                                            to vm2.vchan_buf);


        connection seL4RPCCall request_manager(from vm0.vm_manager_request, from vm1.
            vm_manager_request,
                                            from vm2.vm_manager_request, to vm_manager.
                                                manager_request_in);

        connection seL4SharedDataWithCaps manager_rpc_buf0(from vm_manager.
            manager_request_buf0,
                                            to vm0.vm_manager_request_buf);
        connection seL4SharedDataWithCaps manager_rpc_buf1(from vm_manager.
            manager_request_buf1,
                                            to vm1.vm_manager_request_buf);
        connection seL4SharedDataWithCaps manager_rpc_buf2(from vm_manager.
            manager_request_buf2,
                                            to vm2.vm_manager_request_buf);

        connection seL4DomInfo conn_dom_info(from vm0.dom_info,from vm1.dom_info,
                                            from vm2.dom_info, to vm_manager.dom_info);

        connection seL4SharedData evtchn_port_status(from event_channel.event_port_status, to
             vm0.event_port_status, to vm1.event_port_status,
                                                    to vm2.event_port_status);
        connection seL4RPCCall evtchn_port_lock0(from event_channel.evt_port0, to vm0.evt);
        connection seL4RPCCall evtchn_port_lock1(from event_channel.evt_port1, to vm1.evt);
        connection seL4RPCCall evtchn_port_lock2(from event_channel.evt_port2, to vm2.evt);


        // Ethernet driver that we share to Linux
        component Ethdriver82574 ethdriver;
        component HWEthDriver82574 HWEthdriver;

        // Hardware resources for the ethernet driver
        connection seL4HardwareMMIO ethdrivermmio(from ethdriver.EthDriver, to HWEthdriver.
            mmio);
        connection seL4HardwareInterrupt hwethirq(from HWEthdriver.irq, to ethdriver.irq);
        // Connect vm1 ethernet to the ethdriver
        connection seL4Ethdriver ethdriver_con1(from vm1.ethdriver, to ethdriver.client);
        connection seL4Ethdriver ethdriver_con2(from vm2.ethdriver, to ethdriver.client);

        // For ssh access
        connection seL4Ethdriver ethdriver_con0(from vm0.ethdriver, to ethdriver.client);

        //connection seL4DomNotification direct_notify_1(from vm0.direct_dom_notify_out, to
            vm2.direct_dom_notify_in);
        //connection seL4DomNotification direct_notify_2(from vm2.direct_dom_notify_out, to
            vm0.direct_dom_notify_in);

        export ethdriver.client -> ethdriver_interface;


    }

    configuration {
        VM_CONFIGURATION_DEF()
        VM_PER_VM_CONFIG_DEF(0)
```

```
        VM_PER_VM_CONFIG_DEF(1)
        VM_PER_VM_CONFIG_DEF(2)

        vm0.simple_untyped24_pool = 6;
        vm0.guest_ram_mb = 150;
        vm0.heap_size = 0x10000;
        vm0.kernel_cmdline = DOM0_VM_GUEST_CMDLINE;
        vm0.kernel_image = DOM0_FILE_SERVER_KERNEL_IMAGE;
        vm0.kernel_relocs = DOM0_FILE_SERVER_KERNEL_IMAGE;
        vm0.initrd_image = DOM0_FILE_SERVER_ROOTFS;
        vm0.iospace_domain = 0x0f;
        vm0_config.ram = [ [ 0x20800000, 23 ], [ 0x21000000, 24 ], [ 0x22000000, 25 ], [ 0
            x24000000, 26]];
        vm0_config.pci_devices_iospace = 1;


        vm1.simple_untyped24_pool = 6;
        vm1.guest_ram_mb = 130;
        vm1.heap_size = 0x10000;
        vm1.kernel_cmdline = VM_GUEST_CMDLINE;
        vm1.kernel_image = APPVM_FILE_SERVER_KERNEL_IMAGE;
        vm1.kernel_relocs = APPVM_FILE_SERVER_KERNEL_IMAGE;
        vm1.initrd_image = APPVM_FILE_SERVER_ROOTFS;
        vm1.iospace_domain = 0x10;
        vm1_config.ram = [ [ 0x30800000, 23 ], [ 0x31000000, 24 ], [ 0x32000000, 25 ], [ 0
            x34000000, 26]];
        vm1_config.pci_devices_iospace = 1;

        vm2.simple_untyped24_pool = 2;
        vm2.guest_ram_mb = 80;
        vm2.heap_size = 0x10000;
        vm2.kernel_cmdline = VM_GUEST_CMDLINE;
        vm2.kernel_image = KERNEL_IMAGE;
        vm2.kernel_relocs = KERNEL_IMAGE;
        vm2.initrd_image = ROOTFS;
        vm2.iospace_domain = 0x11;
        vm2_config.ram = [ [ 0x40800000, 23 ], [ 0x41000000, 24 ], [0x42000000, 25 ], [ 0
            x44000000, 26]];
        vm2_config.pci_devices_iospace = 1;

        /* seL4store request buffer */
        vm0.seL4store_request_buf_id = 1;
        vm0.seL4store_request_buf_size = 4096;
        vm1.seL4store_request_buf_id = 1;
        vm1.seL4store_request_buf_size = 4096;
        vm2.seL4store_request_buf_id = 1;
        vm2.seL4store_request_buf_size = 4096;

        /* vchan buffer */
        vm0.vchan_buf_id = 2;
        vm0.vchan_buf_size = 1474560;
        vm1.vchan_buf_id = 2;
        vm1.vchan_buf_size = 1474560;
        vm2.vchan_buf_id = 2;
        vm2.vchan_buf_size = 1474560;
        share_malloc.vchan_buf_id = 2;
        share_malloc.vchan_buf_size = 1474560;

        vm0.vm_manager_request_buf_id = 3;
        vm0.vm_manager_request_buf_size = 4096;
        vm1.vm_manager_request_buf_id = 3;
        vm1.vm_manager_request_buf_size = 4096;
        vm2.vm_manager_request_buf_id = 3;
        vm2.vm_manager_request_buf_size = 4096;

        /* VM domain information */
```

```
vm0.domain_num = 0;
vm1.domain_num = 1;
vm2.domain_num = 2;

seL4_store.heap_size = 0x30000;
share_malloc.heap_size = 0x10000;
vm_manager.heap_size = 0x8000;

vm2_config.pci_devices = [
    {"name" : "USB2d",
        "bus":0, "dev":0x1d, "fun":0,
        "irq" : "USB2d",
        "memory": [
            {"paddr":0xf7f37000, "size":0x400, "page_bits":12},
        ],
    },
];

vm2_config.irqs = [
    {"name":"USB2d","source":23,"level_trig":1,"active_low":1,"dest":14},
];


ethdriver.simple = true;
ethdriver.cnode_size_bits = 12;
ethdriver.iospaces = "0x12:0x0:0x19:0";
ethdriver.iospace_id = 0x12;
ethdriver.pci_bdf = "0:19.0";
ethdriver.simple_untyped20_pool = 2;
ethdriver.heap_size = 0x10000;
ethdriver.dma_pool = 0x200000;

HWEthdriver.mmio_paddr = 0xf7f00000;
HWEthdriver.mmio_size = 0x20000;
HWEthdriver.irq_irq_type = "pci";
HWEthdriver.irq_irq_ioapic = 0;
HWEthdriver.irq_irq_ioapic_pin = 20;
HWEthdriver.irq_irq_vector = 20;


vm1.ethdriver_attributes = "1";
vm1.ethdriver_global_endpoint = "vm1";
vm1.ethdriver_badge = "134479872";
vm1.ethdriver_mac = [6, 0, 0, 11, 12, 13];
vm1_config.init_cons = [
    {"init":"make_virtio_net", "badge":134479872, "irq":"virtio_net_notify"},
];


vm2.ethdriver_attributes = "2";
vm2.ethdriver_global_endpoint = "vm2";
vm2.ethdriver_badge = "134479872";
vm2.ethdriver_mac = [6, 0, 0, 12, 13, 14];
vm2_config.init_cons = [
    {"init":"make_virtio_net", "badge":134479872, "irq":"virtio_net_notify"},
];

//For ssh access
vm0.ethdriver_attributes = "3";
vm0.ethdriver_global_endpoint = "vm0";
vm0.ethdriver_badge = "134479872";
vm0.ethdriver_mac = [6, 0, 0, 13, 14, 15];
vm0_config.init_cons = [
    {"init":"make_virtio_net", "badge":134479872, "irq":"virtio_net_notify"},
];
```

```
    }
}
```

Appendix 3 - Lines of Code

Program/Library	**LoC**
VMM Handler	310
Linux Driver	649
Linux Library	125
CAmkES Component	196
Total	**1280**

Table 1: Event Channel Code Base

Program/Library	**LoC**
VMM Handler	-
Linux Driver	-
Linux Library	270
CAmkES Component	992
Total	**1262**

Table 2: seL4 Store Code Base

Program/Library	**LoC**
VMM Handler	234
Linux Driver	183
Linux Library	178
CAmkES Component	395
Total	**990**

Table 3: Share Allocator Code Base

Program/Library	**LoC**
VMM Handler	160
Linux Driver	303
Linux Library	142
CAmkES Component	296
Total	**901**

Table 4: vm-manager Code Base

Program/Library	**LoC**
VMM Handler	129
Linux Driver	144
Linux Library	137
CAmkES Component	-
Total	**410**

Table 5: Cross-VM RPC Code Base